ABSTRACT

MIRHOSSEINI, SAMIM. Addressing CS-Ed Course Material Preparation and Delivery Frictions Through ClassOps. (Under the direction of Chris Parnin and John-Paul Ore).

Computer science instructors typically have many responsibilities, such as creating material, delivering lectures, clarifying student questions, and grading student deliverables, while the demand for computer science education has been increasing. Handling all of these responsibilities is challenging in itself. However, it is made worse when instructors have to spend significant time grappling with hidden obstacles and the hidden work to resolve them. While many previous studies in the CS education community have focused on improving student experience, in this work, we aim to study instructor challenges, potential practices for addressing them, and their effect on the students. The thesis of this dissertation is ***ClassOps can improve educational material preparation and delivery by simultaneously addressing several needs, such as computing environment, automation, and authoring interface***.

This research aims to make teaching computer science less challenging for instructors and more streamlined for students. The first study found evidence for a potential lack of quality, inconsistency, and failure in an existing form of educational material, online tutorials, when analyzed through the lens of executability. We also offered possible strategies and ideas for authors and toolsmiths to improve the quality of future tutorials. In the second study, we gained more insights from the perspective of educational material authors (i.e., instructors). We reported several pain points, workarounds, and remaining desires of CS instructors and discussed preliminary ideas that may help address them in the future. In the third study, we conduct a more in-depth study on one of the pain points regarding helping students with their computing environment issues. We found the root causes of common issues students experienced in their computing environments. We inspired the design of OPUNIT, a tool for environment verification that proved effective in helping students discover and fix issues and improve confidence. Finally, we studied a potential solution to another pain point regarding course material preparation and delivery. We experimented with a computational notebook design, DOCABLE, to report its effects on the experience of instructors delivering the course and the students' learning experience. To guide the design of future tools in this space, we formalize findings from instructor and student experience as a set of properties that we deem essential in ClassOps tools in this domain.

© Copyright 2023 by Samim Mirhosseini

All Rights Reserved

Addressing CS-Ed Course Material Preparation and Delivery Frictions Through ClassOps

by
Samim Mirhosseini

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2023

APPROVED BY:

Chris Parnin Co-chair of Advisory Committee	John-Paul Ore Co-chair of Advisory Committee
Sarah Heckman	Shiyan Jiang

Austin Henley

DEDICATION

With profound appreciation, I dedicate this dissertation to my incredible parents, whose support has propelled me forward.

BIOGRAPHY

Samim Mirhosseini was born in Alborz, Iran. He became interested and started learning computer programming at age 15. He always knew he wanted to do something in engineering and computer science, so he pursued a Bachelor of Science in Computer Science at North Carolina State University. He experienced academic research during his undergraduate studies and became interested in research. As his interest in software engineering research grew, he started his work towards a Ph.D. in Computer Science at North Carolina State University and joined Alt-Code lab to work under the guidance of Dr. Chris Parnin. His research interests include empirical studies in computer science education and automation.

ACKNOWLEDGEMENTS

I would like to thank Dr. Chris Parnin for his support and mentorship over the several years of my higher education, helping me develop academic, industry, and communication-related skills. Most of this work would not have been possible without his guidance. Additionally, I would like to thank my Ph.D. committee, Dr. John-Paul Ore, Dr. Sarah Heckman, Dr. Shiyan Jiang, and Dr. Austin Henley, for their critical questions and feedback that helped guide this work and enhanced its quality. Finally, I thank my friends from Alt-Code lab and SE-lab. Thank you Nischal, Eric, Mahnaz, Kai, Chris Brown and the rest of my peers, for always providing me with your great feedback, inspiration, and encouragement throughout this journey.

TABLE OF CONTENTS

List of Tables . . . . **viii**

List of Figures . . . . **ix**

Chapter 1 Introduction **1**
1.1 Chapter 1: What Is the Significance of Online Tutorials Quality? 2
1.2 Chapter 2: Why Is It Important To Study Instructors? 2
1.3 Chapter 3 and 4: What Defines a ClassOps? 3
1.4 Novel Contributions 3

Chapter 2 Evaluating Quality of Tutorials Through Executability **5**
2.1 Study Rationale 5
2.2 Background 6
2.3 Methodology 8
2.3.1 Research Questions 8
2.3.2 Data Collection 8
2.3.3 Execution Harness 10
2.3.4 Baseline: Naive Execution 10
2.3.5 Human-Annotation-Based Execution 12
2.4 Naive Execution Results 14
2.4.1 Executability Rates 14
2.4.2 Errors and Exit Codes 15
2.4.3 Code Block Types 15
2.4.4 Summary 18
2.5 Human-annotation-based execution results 18
2.5.1 Annotations 18
2.5.2 Executability Rates 19
2.5.3 What Execution Failures Remain? 19
2.5.4 Summary 26
2.6 Limitations 27
2.7 Related Work 27
2.8 Discussion 28
2.8.1 Implication I—Provide Accessible Alternatives for Tutorial Takers . . 29
2.8.2 Implication II—Invest in Automated Tutorial Testing 31
2.8.3 Implication III—Provide Self-Contained Tutorials. 32
2.8.4 Implication IV—Be Intentional on Whether Instructions are Human-Readable or Machine-Readable 32
2.9 Conclusion 34

Chapter 3 Investigating Instructor Pain Points and Desires **35**

3.1 Study Rationale . . . 35
3.2 Methodology . . . 36
3.2.1 Interviews . . . 36
3.2.2 Interview Analysis . . . 37
3.3 Findings . . . 38
3.3.1 RQ1: What did instructors wish they could change? . . . 38
3.3.2 RQ2: What are current attempts of addressing pain points? . . . 43
3.4 Limitations . . . 45
3.5 Related Work . . . 45
3.6 Discussion . . . 46
3.7 Conclusion . . . 49

Chapter 4 Investigating Issues Related to Computing Environments . . . 51
4.1 Study Rationale . . . 51
4.2 Background . . . 52
4.3 Properties . . . 55
4.3.1 Availability . . . 55
4.3.2 Reachability . . . 56
4.3.3 Identifiable . . . 58
4.3.4 Capability . . . 59
4.4 Opunit . . . 60
4.4.1 Using OPUNIT . . . 60
4.4.2 Checks . . . 61
4.4.3 Environments . . . 61
4.4.4 Report . . . 62
4.5 Experiences . . . 62
4.5.1 Supporting Initial Course Setup . . . 64
4.5.2 Using OPUNIT for Workshops . . . 64
4.5.3 Student Feedback . . . 64
4.6 Future Directions . . . 67
4.7 Conclusion . . . 68

Chapter 5 Investigating Needs for Creating and Deliveing Course Content . . . 69
5.1 Study Rationale . . . 69
5.2 Background and Related Work . . . 70
5.3 The Notebook Systems: DOCABLE . . . 72
5.3.1 Content . . . 72
5.3.2 Authoring Experience . . . 75
5.3.3 Computing Environment . . . 77
5.3.4 Execution Mechanism . . . 77
5.4 Methodology . . . 78
5.4.1 Participants . . . 79

5.4.2 Analysis . . . 81
5.4.3 Limitations . . . 81
5.5 Findings . . . 81
5.5.1 How Did Instructors Use DOCABLE? . . . 82
5.5.2 Instructors' Perspective . . . 82
5.5.3 Students' Perspective . . . 84
5.5.4 Remaining Needs . . . 86
5.6 Discussion: Essential Properties . . . 87
5.7 Conclusion . . . 90

Chapter 6 Conclusion . . . 91

References . . . 93

LIST OF TABLES

Table 2.1 Executability of tutorials in our different experiments. Naive: running all the code blocks of the tutorials, Naive++: running naive, but updating `apt` and `apt-get` commands to use `-y` option. 14
Table 2.2 Executed code blocks and corresponding execution status (exit codes) in naive++ approach. Non-zero exit codes indicate errors. 16

Table 4.1 Follow-up Survey Responses 66

Table 5.1 Participants (ID), their titles, institution size, courses they teach, years of teaching experience (Exp), and their DOCABLE adoption in this study. 80
Table 5.2 Usability Survey Responses 85

LIST OF FIGURES

Figure 2.1 an excerpt of the technical software tutorial, "*How To Set Up a Node.js Application for Production on Ubuntu 16.04*"[1]. You can see the full instructions, and even give it a try. 6

Figure 2.2 Tutorial fragment with code block. We extract the command '`sudo add-apt-repository ppa:certbot/certbot`' for running in our execution harness. 9

Figure 2.3 Human-annotation execution harness can generate a report based on the original tutorial, which enables inspection of tutorial execution. In this example, the '`apt-get install`' command did not execute successfully. 11

Figure 2.4 An example `steps.yml` file 13

Figure 2.5 An example of a presentation code block. 18

Figure 2.6 We observed a problem in the Linuxize tutorial for installing the package "steam", which omits an explicit apt-get update command. 20

Figure 2.7 An example of a inconsistent file content block, which includes omitted content indicated by "..." and a placeholder value indicated by `"your-server-ip"` 21

Figure 2.8 An example of a contradictory instruction which asks to install `0.9.3` version of a tool and then checks if `0.9.2` has been installed. 23

Figure 2.9 An example of a volatile instruction, which includes the date and time a file was modified when tutorial author created these instructions. . 24

Figure 2.10 The download link referenced in this tutorial is no longer available. . 25

Figure 2.11 An example of an inaccessible resource used in a tutorial. 26

Figure 2.12 Using `/etc/hosts` file as an alternative for requiring a domain name in the instructions. 30

Figure 2.13 Sample of annotations used by tutorials at LEARNK8S for rendering code blocks 31

Figure 4.1 OPUNIT's verify command to test pipelines workshop 54

Figure 4.2 Examples of students' broken JSON files shown in red color 56

Figure 4.3 Result of running an OPUNIT profile 63

Figure 5.1 A snapshot of a notebook on DOCABLE demonstrating: (A) a file editor that's editable by the reader and saved on the file system, (B) a live executable file which is also editable by the reader and allows execution with the help of the run button. In this screenshot, (C) `usingPersonClass.js` has been executed and the output is displayed. 73

Figure 5.2 A snapshot of a notebook on DOCABLE demonstrating coding exercises that provide reader with an executable file editor where they can implement and run their answers. 74
Figure 5.3 A snapshot of a notebook on DOCABLE demonstrating (A) a file editor that's editable by the reader and saved on the file system, (B) a live terminal connected to the reader's isolated cloud environment. 75
Figure 5.4 DOCABLE + menu for adding various components to the notebook, and AI content and exercise generation features marked in red. 76
Figure 5.5 A code editor cell, from author's view, showing options for setting the language (A), script path on the file-system (B), and a gear icon to optionally customize the default execution behavior (C). 76
Figure 5.6 DOCABLE notebook's computing environment settings 78

CHAPTER

1 INTRODUCTION

In computer science education, the main components that play a role in a high-level view are the instructors, students, and the educational content utilized in the learning process. These components interact dynamically in a cycle to construct an educational experience. Several studies have emphasized the significance of these components to ensure that learning is occurring. For instance, Sidelinger [88] framed this as instructors and students working together towards an common outcome. In contrast, instructors have a crucial role in connecting the students with course content. Furthermore, the research has shown quality and purposefulness of activities can significantly affect the motivation in learning [24]. Therefore, it is crucial to focus on optimizing the quality of content and the collaboration among the three components to enhance the overall effectiveness of education.

As a general theme in this dissertation, we introduce *ClassOps* as a movement. Inspired by Bass et al.'s definition of DevOps [3], we define ClassOps as "*a set of tools and practices intended to reduce the frictions of teaching and running class operations, while ensuring high-quality learning experience*". ClassOps achieves these goals by promoting better integration and faster cycles between the components involved in CS education: instructors, students,

the educational material, and how it is delivered to students.

> **Thesis Statement:**
> ClassOps can improve educational material preparation and delivery by simultaneously addressing several needs, such as computing environment, automation, and authoring interface.

1.1 Chapter 1: What Is the Significance of Online Tutorials Quality?

Online software tutorials as a type of educational content serve a diverse range of learners, including self-guided individuals seeking to expand their knowledge, university students searching for supplementary materials, and professionals from the industry aiming to improve their skills. These tutorials also cover a broad spectrum of topics ranging from computer science fundamentals to more specialized topics such as DevOps and security. The accessibility and availability of these online tutorials make them a popular choice; for example, as mentioned in chapter 2, over 400 million readers have reached just one of the online tutorial sources, DigitalOcean.

Because of this widespread readership and the significant number of readers accessing them, a lack of quality can impact a more significant number of learners. Additionally, understanding these online tutorials' weaknesses is essential to our research to identify the best practices and design principles to guide the development of more effective and engaging online educational content and the associated tooling needed for maintaining them.

1.2 Chapter 2: Why Is It Important To Study Instructors?

Computer science instructors play a crucial role in the quality of training and educational experience that students receive. Instructors typically have many responsibilities such as creating material, delivering lectures, clarifying student questions, and grading student deliverables. Additionally, growing demands for meeting computer science enrollment has strained instructors with high teaching loads, as insufficient teaching faculty are

available [17], across large and small institutes. Handling all of these responsibilities and addressing scale can be challenging in itself, but is exacerbated when instructors spend significant time grappling with hidden obstacles and invisible work to resolve them. In a recent study [58] we found five categories of such challenges that instructors wished they could change—for example, answering student questions, and course material preparation.

Many of the previous research studies have analyzed interventions for improving student learning and focused on the resulting experience from the perspective of students [64, 38, 15], while few studies have focused on the perspective of an instructor. For example study by Lau et al. [51], which focused on challenges specific to scaling four data-science courses to larger size. Although analyzing experience of students can give us many useful insights, we believe it is just as important to understand what instructors experience, identify opportunities for addressing pain points related to their hidden obstacles and invisible work.

1.3 Chapter 3 and 4: What Defines a ClassOps?

The first two chapters of this dissertation investigate educational materials and their authors (i.e., the instructors) to uncover their weaknesses, challenges, and needs. In the following two chapters, we study and guide a more concrete discussion of what ClassOps tools and practices need to support to address existing challenges. Chapter 4 focuses on automating instructor feedback as a tool to support the computing environment challenges experienced by students, and chapter 5 formalizes properties needed to support course content creation and delivery of instructors while improving the learning experience of their students.

1.4 Novel Contributions

1. An empirical analysis of online software tutorial executability as a measure of quality (Chapter 2);

2. Possible strategies for improving the quality of online tutorials, validated by feedback from major stakeholders in technical documentation (Chapter 2);

3. A qualitative investigation of computer science instructor challenges, pain points, and current workarounds (Chapter 3);

4. A list of unsatisfied instructor desires, a discussion on the consequences of not solving them, and guidelines to alleviate them in future research (Chapter 3);
5. The root causes for the most common issues students experience related to their computing environments (Chapter 4);
6. A suggested ClassOps tool for environment verification inspired by our discovered root causes, and evaluation of this tool through student experiences (Chapter 4);
7. A qualitative evaluation of a computational notebooks design in addressing instructor needs and improving learning experience (Chapter 5);
8. Formalized essential properties for computer science education tools to guide design in this domain (Chapter 5);

CHAPTER

2

EVALUATING QUALITY OF TUTORIALS THROUGH EXECUTABILITY

This chapter is based on material originally published in FSE 2020 [57].

2.1 Study Rationale

This chapter details an empirical study of **Evaluating the Executability of Software Tutorials** [57]. Over the years, executability has been consistently used in literature to measure the quality [98, 39, 73] and the goal of this study is to gain a better understanding of online software tutorial quality. Software tutorial is an important modality of CS-Ed material, given their popularity (over 409 million views) and the skills they cover.

This study satisfies part of thesis related to **educational material**, its associated barriers for the tutorial creators, and how it **can be improved** with support of tools and practices in the future.

[1] `https://www.digitalocean.com/community/tutorials/how-to-set-up-a-node-js-application-for-production-on-ubuntu-16-04`

Hello World Code

First, create and open your Node.js application for editing. For this tutorial, we will use `nano` to edit a sample application called `hello.js`:

```
$ cd ~
$ nano hello.js
```

Insert the following code into the file. If you want to, you may replace the highlighted port, `8080`, in both locations (be sure to use a non-admin port, i.e. 1024 or greater):

hello.js

```
#!/usr/bin/env nodejs
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(8080, 'localhost');
console.log('Server running at http://localhost:8080/');
```

Now save and exit.

Figure 2.1: an excerpt of the technical software tutorial, "*How To Set Up a Node.js Application for Production on Ubuntu 16.04*"[1]. You can see the full instructions, and even give it a try.

2.2 Background

Many software tutorials, include step-by-step instructions for installing, configuring, and using software tools, which are essential in the software development process. For example, software tutorials hosted by DigitalOcean have been viewed over 409 million times, including tutorials such as "*How To Secure Nginx with Let's Encrypt*" and "*How To Set Up a Node.js Application for Production on Ubuntu 16.04*" as shown in Figure 2.1. Tutorials are featured prominently in developer-related search results [91], and serve a variety of purposes, such as integrating with instructional materials for classrooms [37], supporting documentation efforts [70], and training underrepresented and professional developers through community-driven [48] or paid workshops [2].

[2]https://learnk8s.io/

The ideal tutorial, as described in DigitalOcean's guidelines for tutorial creators [94], should follow several principles. First, tutorials should be accessible for all tutorial takers, being "*as clear and detailed as possible without making assumptions about the reader's background knowledge.*" Furthermore, tutorials should be executable from start-to-finish: "*We explicitly include every command a reader needs to go from their first SSH connection on a brand new server to the final, working setup.*" Finally, tutorials are not merely scripts, but should explain and impart knowledge: "*We also provide readers with all of the explanations and background information they need to understand the tutorial. The goal is for our readers to learn, not just copy and paste.*"

Unfortunately, tutorials can fall far from this ideal. Despite their importance, software tutorials require considerable effort to produce, test, and maintain in order to ensure a high-quality learning experience. Tutorial creators face several barriers: supporting different levels and environments of tutorial takers [77], preventing instructions from becoming stale as tools or environments quickly evolve [52], and overcoming the *expert blind spot effect* [65], when tutorial creators do not anticipate steps where novice tutorial takers may have difficulty [62]. As a result, undiscovered issues [49] in software tutorials can lead to frustrating and ineffective learning experiences [46, 60].

To systematically understand what issues software tutorials contain—and what tutorial creators can do to avoid them—we investigate software tutorials through the lens of *executability*, measuring to what extent we can follow step-by-step instructions to their "final, working setup". To this end, we conduct an empirical study of 663 tutorials, first by measuring executability with a naive execution strategy (as one would "just copy and paste"), and finally with a more sophisticated strategy using human-annotation for interpreting and executing instructions. Through a qualitative analysis, we identified several issues in tutorials that limited executability, and we validated these issues with 6 informants, who were expert stakeholders in technical documentation.

Our findings show that with a naive execution strategy, we achieve overall executability rate of only 26%. Even with a more sophisticated strategy using annotated tutorials, executability rate only increases to 52.3%—and even more concerning no tutorial successfully executed all steps to its "final, working setup". Our qualitative analysis revealed several issues, such as inaccessible resources, missing steps, inconsistency in handling file content, and documentation rot, which detract from the usability and value of the tutorials. Our informants, generally agreed with significance of the results and illustrated scenarios where these problems have occurred; however, informants had a mixed consensus on the severity

of some problems and how to best address them. Finally, we provide design implications for technical writers, toolsmiths, and software engineering researchers for improving tutorials, such as providing accessible alternatives for tutorial takers, and investing in automated tutorial testing to ensure continuous quality of software tutorials.

2.3 Methodology

To explore why software tutorials produce execution failures, we conducted a mixed-methods study through an empirical study on tutorials collected from various online sources, and through a qualitative analysis of tutorials. We do so through the following research questions:

2.3.1 Research Questions

- **RQ1: Can tutorials be naively executed. If not, why?** Can the average tutorial be run to completion naively, or will it result in failure? What failures occur when there is limited knowledge on how to follow instructions?

- **RQ2: Can tutorials be executed with limited human-annotation?** What extra human interpretation is needed to interpret and execute more instructions? What barriers remain that prevent fully automated testing of a software tutorial?

2.3.2 Data Collection

We selected three popular sources of software tutorials: Vultr[3], DigitalOcean[4], and Linuxize[5]. All tutorials were related to installing software tools, security, and configuring and operating virtual computing environments, topics frequently important in continuous deployment processes in software engineering [71]. We used a webscraping script to crawl and download all tutorials for the Ubuntu operating system hosted on the sources, yielding a total of 780 tutorials.

We organized our collected data into target platform specified by the tutorial (e.g. Ubuntu 18.04), and then removed duplicate tutorials with the same content but targeting

[3] `https://www.vultr.com/docs/category/ubuntu/`
[4] `https://www.digitalocean.com/community/tutorials`
[5] https://linuxize.com/tags/ubuntu/

different platforms. We also excluded tutorials that targeted a deprecated Linux distro (i.e. Ubuntu 12.10) for which we could not find a stable base image, and tutorials that were primarily focused on using GUI interfaces (e.g. "*How To Set Up Continuous Integration Pipelines with Drone on Ubuntu 16.04*" [6]). After filtering, our collected dataset of 780 tutorials was reduced to 663 tutorials. In summary, our filtered dataset included 339 tutorials from Vultr, 224 tutorials from DigitalOcean and 100 tutorials from Linuxize.

We then drew a stratified random sample of tutorials in our dataset (6%), in order to facilitate qualitative analysis, as done in Kim and Ko [46], who inspected a sample of 30 tutorials. Because our tutorial sources were not uniformly represented in the dataset, we determined a statistically representative sample size for each source. To do so, we used a proportionate stratified random sampling [43] by considering each source as a strata. We used a relaxed confidence interval ($80\% \pm 10\%$) to calculate the sample size of each strata, allowing us to target diversity over representativeness [63]. This yielded a total of 40 tutorials containing 787 content blocks—20 tutorials from Vultr, 14 tutorials from DigitalOcean, and 6 tutorials from Linuxize.

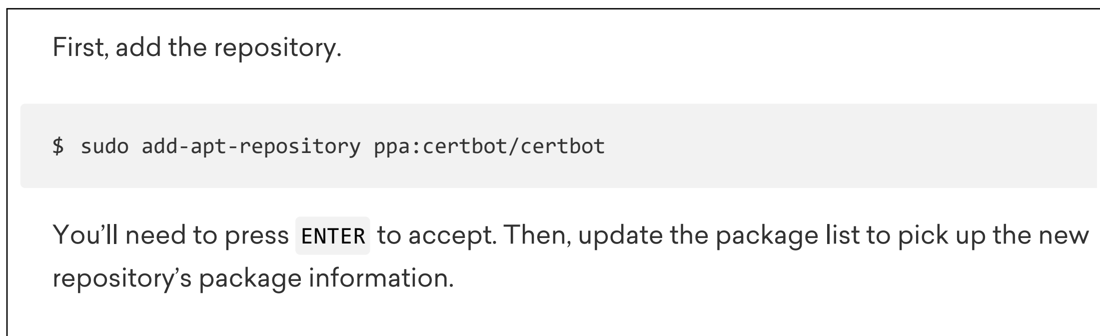


Figure 2.2: Tutorial fragment with code block. We extract the command '`sudo add-apt-repository ppa:certbot/certbot`' for running in our execution harness.

[6] `https://www.digitalocean.com/community/tutorials/how-to-set-up-continuous-integration-pipelines-with-drone-on-ubuntu-16-04`

2.3.3 Execution Harness

To create our execution harness, we initialize a new virtual machine environment with 4GB of RAM and 2 CPU cores. The virtual machine base image is selected to match the operating system tag associated with the tutorial article. Tutorial instructions are executed on the headless virtual machines through an SSH connection. Commands are instrumented to log output, failures, and exit status of the operation. A new execution harness is created for every tutorial.

2.3.4 Baseline: Naive Execution

To answer RQ1, we implement a baseline technique, we call *naive execution* which simply executes text within a content block. The technique closely mirrors previous studies on executability, where no interpretation is performed on the code snippets on Stack Overflow [98] or gists [39] when measuring executability rates.

We designed a custom CSS selector to extract code blocks from each source of tutorials (see Figure 2.2). A code block is defined in the HTML specification as an element that "represents a fragment of computer code", and is marked by the `<code></code>` tag. After extraction, we execute the verbatim text as a shell command within the execution harness. The resulting exit code, the stdout and stderr streams are also recorded. If the command never terminates after 10 minutes, we record a timeout, and mark the remaining instructions as unreachable.

To measure executability, we count the number of code block executions reporting a non-error exit code (indicated by 0). We do not directly use stderr to determine execution failure, as some commands print information to stderr, such as `python -version`. We discuss the limitations of this approach in Section 2.6.

To identify why execution fails, we first cluster the commands by exit code after execution, as done by Horton and Parnin [39]. The distribution of execution failures provides a high-level overview of failure causes (e.g. an exit code status of `E127` occurs when a command cannot be found). To understand why a code block cannot be naively executed, we then perform an *open card sort* [90], using similar method used by Begel and Zimmermann [5], to organize code blocks into descriptive categories. The categories provide insight into various possible interpretations under which a code block could be executed.

Step 1 — Installing StrongSwan

First, we'll install StrongSwan, an open-source IPSec daemon which we'll configure as our VPN server. We'll also install the StrongSwan EAP plugin, which allows password authentication for clients, as opposed to certificate-based authentication. We'll need to create some special firewall rules as part of this configuration, so we'll also install a utility which allows us to make our new firewall rules persistent.

Execute the following command to install these components:

```
$ sudo apt-get install strongswan strongswan-plugin-eap-mschapv2 moreutils \
                                                              iptables-persistent

error: exit code = 100
output:
E: Unable to locate package strongswan
E: Unable to locate package strongswan-plugin-eap-mschapv2
E: Unable to locate package moreutils
E: Unable to locate package iptables-persistent
 exit code: 100 command output:
 Reading package lists...
Building dependency tree...
Reading state information...
```

Note: While installing `iptables-persistent`, the installer will ask whether or not to save current IPv4 and IPv6 rules. As we want any previous firewall configurations to stay the same, we'll select yes on both prompts.

Now that everything's installed, let's move on to creating our certificates:

Step 2 — Creating a Certificate Authority

An IKEv2 server requires a certificate to identify itself to clients. To help us create the certificate required, StrongSwan comes with a utility to generate a certificate authority and server certificates. To begin, let's create a directory to store all the stuff we'll be working on.

```
$ mkdir vpn-certs
$ cd vpn-certs
```

Figure 2.3: Human-annotation execution harness can generate a report based on the original tutorial, which enables inspection of tutorial execution. In this example, the ‘`apt-get install`’ command did not execute successfully.

2.3.5 Human-Annotation-Based Execution

To answer RQ2, we implement a technique, with the help of human annotations that can behave according to the code block types annotated by humans.

Context

Rather than devising an automatic execution technique, we wanted to use the opportunity to derive annotations for instructions based on human classification. This approach has several benefits: First, the annotations provide a *bounded* interpretation of instructions. For example, if we provide an annotation with an expected response from a command, such as type "yes" or enter, we can specifically measure how the presence of interactive prompts effects executability. More importantly, we can begin to model different levels of tutorial reader understanding. For example, a complex operation might involve starting a command as background process, which requires more knowledge about shell operations. Finally, automation is ultimately possible in future efforts, such as techniques which try to automatically infer annotations for commands.

Design

The design of our human-annotation execution harness is inspired by behavioral-driven testing tools, such as Cucumber, which uses an external stepfile for selecting and asserting expected test behaviors. Here, we extend this concept by providing capabilities for first selecting a command from a tutorial, and then providing an annotated step that describes how to execute the command. Steps select code blocks by matching text occurring above the block. For example, in the stepfile shown in Figure 2.4, the text "Let's create a small server using PHP" can be used to select the associated code block. The `file` annotation tells our execution harness that content should be saved as a file called server.php.

Human-annotation execution harness also has the ability to generate an HTML report based on the original tutorial, which enables inspection of tutorial execution. In this report, passing code blocks are highlighted in green, failing code blocks are highlighted in red, and ancillary information such as error output and exit codes errors are added to the end of code blocks. An example report is shown in Figure 2.3.

```
tutorial_1.html:
  steps:
    - file: "Let's create a small server using PHP. => server.php"
    - serve: "Start it!"
    - run:
        select: "install unzip"
        input: "Do you want to continue? => yes"
        user: root
```

Figure 2.4: An example `steps.yml` file

Annotations

We derived annotations from the results of our open card sort analysis (see Section 2.4.3). For example, we created the expect annotation based on the *Output* category derived in Section 2.4.3. When creating the annotations, we balanced engineering effort with support from tutorials. For example, we did not implement replacing search and replace operations on files, since few tutorials used this technique and engineering effort would be high.

We derived the following annotations:

- **run.** Run a code block as-is, that is *naively*.
- **file.** Select content and store as a file located at the given path.
- **user.** Perform command as user. For example, installation often requires being run as the root user.
- **input.** Provide input to interactive prompts.
- **expect.** Code block is expected output of command another command.
- **serve.** Run code in a background process.
- **persistent.** Allocate a terminal shell to run a series of commands. For example, some tutorials are written to be run in different terminals.

Execution Failures and Peer-Debriefing with Informants

We conduct a qualitative analysis on the reports produced by human-annotation execution harness. We performed an inductive thematic analysis [10] to organize and cluster the execution failures observed in the execution reports into general themes. To further characterize the reports, we performed an additional *purposive sampling*, or non-probabilistic sampling, on tutorials in the entire dataset and composed *memos* [6]. These memos, or author annotations on tutorials, capture interesting exchanges or properties of the software tutorials, promote depth and credibility, and frame problems through tutorial takers information needs. That is, the memos provide a *thick description* to contextualize the findings [74].

Finally, to address the validity of the thematic analysis, we perform a *peer debriefing* [53] with 6 informants, who are experts in tutorials, documentation and book authors in DevOps, including core contributors to Ansible, and documentation leads at DigitalOcean. Our informants performed an expert review of our findings and generated reports that summarized their assessment on the validity and impact of our results.

2.4 Naive Execution Results

In this section we answer our research question: **RQ1: Can tutorials be naively executed? If not, why?**

2.4.1 Executability Rates

Naive execution of code blocks resulted in an execution rate of 13%—only 1,935 code blocks of 14,876 ran successfully (i.e., non-zero exit code). We also observed a high-rate of timed-

Table 2.1: Executability of tutorials in our different experiments. Naive: running all the code blocks of the tutorials, Naive++: running naive, but updating `apt` and `apt-get` commands to use `-y` option.

	Unreachable	F	P	Blocks	Timed out
N	9769 (66%)	3172 (21%)	1935 (13%)	14876	498 (75%)
N++	5314 (36%)	5646 (38%)	3916 (26%)	14876	261 (39%)

out tutorials (75%). Manual inspection revealed many timed-out while awaiting for user interaction. Often, these commands were early in the tutorial and typically associated with installation commands, such as "`apt-get install package_X`" command was waiting for the user to respond to "`Do you want to continue [y/N]?`" prompt. As a result of timed-out tutorials, many code blocks were simply unreachable.

We devised a small automated *naive patch* to tutorials to improve naive execution. We automatically updated the `apt` and `apt-get` commands within the tutorials to use the `-y | -yes` option, which provides an affirmative response to prompts [7].

We re-ran our experiments with our automated patch, which was effective in reducing the number of tutorials that timed out and allowed more code blocks to be executed. However, an overall low rate of executability persisted: 26% (3,916 out of 14,876 code blocks). Surprisingly, our results are consistent with naive executability rates found in other studies of code snippets. For example, a recent study by Pimentel et al. [73] found that only 24% of Jupyter notebooks could be executed without exceptions. Similarly, only 25–27% of Python code snippets found in Stack Overflow posts [41, 98] and GitHub gists [39] are executable.

2.4.2 Errors and Exit Codes

We categorized exit codes to identify preliminary explanations for why the code block could not be naively executed (see Table 2.2). Naive execution of many code blocks resulted in `E127`, which occurs when a command cannot be found. This indicates previous steps to setup tools did not succeed, or possibly content that was not a command was being executed. Exit codes, such as `E2`, `E5`, and `E128` indicated that commands failed when expected resources, services, or files were unavailable or inaccessible. Finally, many commands were still inaccessible, indicated that possibly other types of commands were still timing out due to interactive prompts.

2.4.3 Code Block Types

Our open card sort revealed four high-level categories based on the inspection of 787 code blocks.

[7] `https://linux.die.net/man/8/apt-get`

Table 2.2: Executed code blocks and corresponding execution status (exit codes) in naive++ approach. Non-zero exit codes indicate errors.

Status	**Blocks**	**Description**
–	5314	Unreachable code blocks
E0	3916	Successful execution
E127	2393	*command not found* error
E1	2032	Catchall for general errors
E100	269	*Unable to locate package X* as a result of running `apt-get install`
ETIME	261	Any command terminated after timeout (10 minutes)
E5	201	*Service X not found, no such file or directory* as a result of `systemctl` commands
E2	134	*Cannot open: No such file or directory*
E255	88	*Couldn't read packet: Connection reset by peer* and *Could not resolve hostname X* as a result of using a template value with `sftp` and `ssh`
E3	79	*Service X not found, no such file or directory* as a result of `systemctl` commands
E6	46	*usermod: user 'X' does not exist* or *curl: (6) Could not resolve host: example.com* as a result of using template values with `usermod` and `curl` commands
E128	30	*repository 'your-github-url' does not exist* as a result of cloning a repository that does not exist
E4	27	*Unit X.service could not be found.* as a result of running `systemctl status X`
E7	24	*Connection refused* as a result of running unavailable URL with `curl`
E8	15	*404: Not Found*, server issued an error response when using `wget`
OTHER	47	Misc. errors occurring infrequently
TOTAL	14876	

Commands (573)

Code blocks primarily involved commands that should be executed in a terminal. However, commands sometimes had specialized execution contexts:

- *text editor*: Open a text editor.
- *template*: Incomplete command requiring tutorial taker to fill in placeholder values.
- *interactive program shell*: The command opens an interactive shell to input more commands, such as the MySQL shell.

File Content (111)

Code blocks often referred to code snippets and configuration files that needed to be placed inside the file system. However, frequently the tutorial provided instructions for manipulating existing file content—and the types of manipulations varied greatly. We observed the following manipulation operations:

- *add*: Append content to the end of a file.
- *partial*: Update part of a file with content.
- *search and replace*: Substitute existing content with new values.
- *uncomment*: Enable content in existing configuration files.
- *conditional*: Multiple options exist on what to update in file.

Instruction Output (93)

Code blocks often contained the *output* of a command, that is, the standard output resulting from executing a command. How the output was displayed varied in several subtle ways:

- *partial*: Trimmed output.
- *template*: Output with placeholder values.
- *interactive prompt*: Output that also includes responses to interactive prompts.
- *interactive program shell*: Results of interactive shell commands, such as results of a SQL query.

Presentation (10)

We also found ten code blocks that presented ancillary information, such as showing a URL (figure 2.5). These code blocks were not directly relevant for execution of the tutorial.

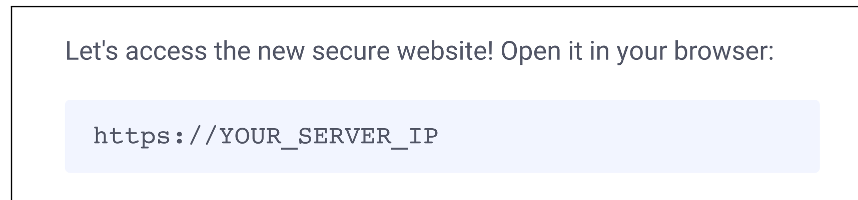


Figure 2.5: An example of a presentation code block.

2.4.4 Summary

Naive execution yielded a low execution rate. Multiple factors contributed to why naive execution failed. Cascading failures, incomplete instructions, missing interactive input and a lack of human interpretation were just some of the factors. Our inspection of code blocks further revealed that in addition to there being multiple types of code blocks, those code blocks also required considerable nuance in interpretation when executing them.

2.5 Human-annotation-based execution results

In this section, we answer the research question: **RQ2: Can tutorials be executed with limited human-annotation?**

2.5.1 Annotations

We created 771 annotations for the 40 tutorials. The majority of annotations corresponded to labeling code blocks as simple commands (455). However, there were more complex commands which needed one or more additional annotations, including 38 input annotations, 47 user annotations, and 3 commands that needed to be run in the background (serve). Finally, 82 commands needed to be associated with a specific terminal session (persistent). We also created 112 file annotations for creating file content at a specified path

and 39 annotations for matching expected output with command output.

We labeled 131 code blocks as `skip` that we deliberately decided to not execute, either because another code block would subsume it, or it was for presentation. For example, some tutorials presented file content in a stagged manner. That is, they built up the file content, provided an explanation, and then continued to explain more fragments of file content. In this case, we implicitly applied `skip` annotations for the intermediate content and then annotated the final and complete code block with the appropriate `file` annotation. Finally, we explicitly skipped and marked 47 code blocks as *failed*, when running a command that would significantly disrupt the execution harness. We discuss the impact on executability rates in our limitations (Section 2.6).

2.5.2 Executability Rates

Human-annotation-based execution of code blocks resulted in an execution rate of 52.3% (343 out of 656 code blocks, when excluding previously mentioned 131 skipped code blocks). Furthermore, no tutorial timed-out and thus all code blocks were reachable. But even with these improvements, no single tutorial completed successfully in its entirety. **Thus, even with limited human annotation, we were unable to bring any tutorial to its "final, working setup."**

2.5.3 What Execution Failures Remain?

We present the remaining execution failures as themes based on our qualitative analysis of human-annotation-based execution reports, and give examples of tutorials instructions that exhibited this problem. Finally, we include excerpts of the informants expert review of our findings.

Simplifying assumptions about environment. Tutorials make simplifying assumptions about the user's environment. For example, some tutorials will assume the package manager's listing is up-to-date and do not explicitly mention running the `'apt-get update'` command. Unfortunately, omitting this step can result in intermittent errors (see Figure 2.6). Another way tutorials make simplifying assumptions is by requiring prerequisites. Prerequisites are sometimes listed at the beginning of the tutorial and described in natural language or reference other tutorials which should be completed first. In many cases we found there are more than one link for each prerequisite and it is not clear which one should be used.

Perform the following steps to install Steam on your Ubuntu desktop:

1. Start by enabling the Multiverse repository which contains software that does not meet the Ubuntu license policy:

```
$ sudo add-apt-repository multiverse
```

```
'multiverse' distribution component enabled for all sources
```

2. Next, install the `steam` package by typing:

```
$ sudo apt install steam
```

error: exit code = 100 **output:** E: Unable to locate package steam

Figure 2.6: We observed a problem in the Linuxize tutorial for installing the package "steam", which omits an explicit apt-get update command.

Our informants recognized this as a general problem with tutorials: “There’s an implicit assumption about the environment” (I5) and “many tutorials assume you have things like a working database” (I4). If tutorials “were all written with *less* assumptions and were more comprehensive that would be great, and people could just skim over the parts they already knew” (I4).

Informants believed tutorials became problematic if they use “too much referral within a tutorial or setup instructions to another tutorial” (I4). Furthermore, “most of tutorials build upon others, and sometimes that can run multiple levels deep” (I3). For a tutorial taker, it can be unclear how long the additional setup will take, or whether those prerequisite tutorials are up-to-date. Even informants can be frustrated with this as tutorial takers: “Once I was *attempting* (I gave up) to install an application and the first tutorial allowed me a choice of 6 ways to install something and none worked.” (I4)

Inconsistent file content blocks Tutorials often contain file content blocks without explicit indication of the expected action a tutorial taker should make. Sometimes the whole content of the file is shown, and sometimes only a subset of a file that need to be updated. Other times the tutorial instructs the tutorial taker to uncomment part of the existing file or append to the end of file.

There is no standard convention across tutorials for representing and presenting these differences. For example, some tutorials indicate omitted content with "..." characters and expect you to add the new content displayed. Other tutorials display exiting content and expect you to replace a small part of the file. For example, in figure 2.7, the code block shows the existing content in the 'settings.py' for Django.

Inadvertently replacing the whole 'settings.py' file with this content would not only introduce syntax errors, but would also omit essential content. Finally, there is a small part of the file that needs to be updated: `"your-server-ip"` should be replaced with the real IP address of the server, which could be missed by the tutorial taker.

Figure 2.7: An example of a inconsistent file content block, which includes omitted content indicated by "..." and a placeholder value indicated by `"your-server-ip"`

Consensus on this topic was mixed. Informants recognized the inconsistency in dealing with file content, but some believed they were justified showing truncated content: "Sometimes the configuration can be long" (I5). Another informant believed that "ambiguity in code blocks are designed to guide a human reader to the right part of the file, and were never designed to be interpreted by a machine" (I6), and "a code block with non-important bits cut out with a "[...]" is better for a human reader". (I6)

Informant (I1) had a different perspective and claimed that summarizing content with

`" . . . "`, "is really not helping people." Furthermore, the informant recognized their *expert blind spot*, and sometimes inappropriately summarized content "because I know what I'm looking for, but a person who has never used the (tutorial's tool)" may not. Tutorial takers may have difficulty "understanding what the summary is showcasing" and may not be able to locate the relevant information: "you tell them run this and look for something and there are a lot of things like ip addresses...". The informant recommends using "something like jq which automatically filters out the output and give what is needed" in order to reduce the reader's confusion. At the same time a shorter output that is programmatically generated helps with automated execution of tutorials.

Missing, contradictory, and volatile instructions. Some tutorials have missing steps that are necessary to successfully run the tutorial. There was no shortage of examples. Several tutorials did not include steps need for creating users, for example, if the tutorial does not mention the creation of a required "backup" user, the next commands that need to run as the "backup" user will fail. Tutorials often omitted whether or not a tutorial taker should use sudo or not for a command, resulting in multiple command failures. Tutorials also did not include steps on how to reach a desirable state: For example, one tutorial provides a command to check whether `ufw`, a firewall tool, is enabled without providing any instructions on *how to enable it*. Tutorials also sometimes contain contradictory instructions. For example, in a tutorial shown in figure 2.8, it asks to install version 0.9.3 of a tool, but later in the tutorial, it shows version 0.9.2 of that tool is installed.

```
$ wget https://github.com/moncho/dry/releases/download/v0.9-beta.3/dry-linux-amd64
```

…

You can test that `dry` is now accessible and working correctly by running the program with its `-v` option.

```
$ dry -v
```

This will return the version number and build details:

```
Version Details Output
dry version 0.9-beta.2, build d4d7a789
```

Figure 2.8: An example of a contradictory instruction which asks to install `0.9.3` version of a tool and then checks if `0.9.2` has been installed.

We also observed several instances where tutorials contained *volatile* instructions, that is instructions whose behavior would produce different results. Volatile instructions often involved commands that output dynamic information, such as PID (process ID), IP addresses, and usernames. As a result, the actual output produced by the command would not match the expected output in the code block, when run by any future tutorial taker (figure 2.9).

To check if the package is installed run the following ls command:

```
$ ls -l /etc/cloud/cloud.cfg
```

If the package is installed the output will look like the following:

```
-rw-r--r-- 1 root root 3169 Apr 27 09:30 /etc/cloud/cloud.cfg
```

error: exit code 0 **actual output:** -rw-r--r-- 1 root root 3612 Oct 4 15:35 /etc/cloud/cloud.cfg

Figure 2.9: An example of a volatile instruction, which includes the date and time a file was modified when tutorial author created these instructions.

All informants emphasized the importance of consistency in instructions. Informants noted the importance of executability to test consistency in tutorials: "*If you have a unified set of annotations it makes it hugely better for everyone. of course now if everyone adds their own style of annotations it sort of defeats the purpose because now you have five ways of running code and there is no uniformity. (For example,) Stackoverflow has a special block specifically for JavaScript (with an option called run snippet) which makes it easier for people to run custom code.*"

After being introduced to our human-annotation-based execution, some informants started systematically reviewing their own tutorials "*From our side, since we started playing with [human-annotation-based executions] we made a few corrections on how we write tutorials. Overall, what this has promoted for us is consistency.*" (I5) For example, the informant now ensures that "*Tutorials tend to have similar instructions or structure. In the example above, all the powershell snippets start with* If you're using Powershell, the command is: *which is easy to grep, and lint. As we develop more material we have the opportunity to standardise our language which makes it easier for students to understand.*" (I5)

Documentation rot. Tutorial instructions, especially if they use reference external resources, may become unavailable or change over time. We encountered several instances where a tutorial failed due to expired resources, such as a url (see Figure 2.10). For example, one tutorial references a PGP key for verification of packages, but that key has since expired, which results in failing installation of packages and in most of the remaining tutorial also not working. Another way tutorials can become out-of-date is if the newer versions of tools or packages are published and no longer work with the instructions provided in the tutorial.

Informants confirmed their difficulty with maintaining tutorials: "*tutorial maintenance is inherently difficult for us. We don't currently have the means to automate tutorial changes, so any time an update happens we have to make changes manually.*" (I2) Another informant describes how they have been internally searching for a way to test tutorials: "*we've actually thought about a lot of these problems ourselves. I would love to be able to automatically execute our tutorials and run some tests as a way to make sure they don't break due to updates to our platform, our target OSs, or other software issues. Alas... it's a fairly hard problem.*" (I3)

One informant shared their frustration as a tutorial taker when tutorials become out-of-date: "*another problem I've definitely seen where [human-annotation-based executions] and others would help is making sure the docs are up to date. I've tried to learn Rust about 3 or 4 times, and each time I gave up because the tutorial was not close to being in sync with current syntax and standard library features.*" (I4)

Figure 2.10: The download link referenced in this tutorial is no longer available.

Inaccessible resources. Some tutorials require having access to additional resources or services, such as the online services from DigitalOcean, GitHub, Vultr, domain names, and additional disks. If these resources are not accessible, tutorials will often fail, and it is not possible to execute the tutorials. For example, in figure 2.11, tutorial requires a DigitalOcean storage volume. Other common examples include domain names, which might involve billing a credit card to register a domain name, or integration with online services such as GitHub or Slack which require registration and credentials.

```
$ sudo rsync -av /var/www/html /mnt/volume-nyc1-01

error: exit code = 23
output:
rsync: change_dir "/var/www" failed: No such file or directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
at main.c(1183) [sender=3.1.1]

 sending incremental file list

sent 20 bytes  received 12 bytes  64.00 bytes/sec
total size is 0  speedup is 0.00
```

Figure 2.11: An example of an inaccessible resource used in a tutorial.

Informants recognized inaccessible resources as a barrier for tutorial takers and described how they attempt to minimize them: "*At DigitalOcean, we aim for our tutorials to be as tech-agnostic as possible, meaning that any reader should be able to complete the tutorial regardless of whether they're using a server provisioned from DigitalOcean or any other provider. This helps make our tutorials accessible, but also leads to lots of caveats like "Depending on your cloud hosting provider, you may need to do [X] or run [X command]". Occasionally, we'll even need to provide alternative instructions to account for different environments. I could see this adding to the difficulty of developing a tutorial execution tool that works flawlessly.*" (I2)

2.5.4 Summary

Executing tutorials with limited human-annotation substantially improves executability rates of tutorials over a baseline of naive execution. However, all evaluated tutorials still contained execution failures, often caused by a multitude of problematic tutorial writing practices, such as missing and inconsistent information and documentation rot. Informants offered their insight on the validity and impact of these findings.

2.6 Limitations

Our mixed-methods approach of investigating tutorials introduces certain trade-offs and limitations.

As explained in the study by Kim et al. [46], tutorials exist in different genres such as interactive tutorials (ex. Khan Academy[8]), web reference (ex. W3Schools[9]), MOOCs (ex. edX[10]), educational games (ex. Code Combat[11]), and creative platforms (ex. Alice[12]). In this study, we focused on web reference genre on topics related to configuration of tools and computing environments. Other types of genres and domains may have different kinds of executability barriers.

There were several trade-offs we made in our implementation of the execution harness and executability study. For example, because we remotely executed through ssh, a few commands, such as `ufw`, could disrupt the network connection. We also limited our scope of execution. For example, we did not execute inline code blocks, which were relatively rare—only 10 instances found in our sample dataset. Therefore, even higher execution rates could be observed through additional engineering effort on the execution harness without necessarily changing tutorials.

Finally, we acknowledge that qualitative research, however rigorously conducted, involves not only the qualitative data under investigation but also a level of subjectivity and interpretation on the part of the researcher as they frame and synthesize the results of their inquiry. In this study, we focused on validity over reliability [54]. To support interpretive validity, we followed the guidelines set by [53] and performed a peer debriefing with our results. Our sampling method was parameterized with a higher tolerance for error. Therefore our study limits our ability to draw precise conclusions for a sample-to-population or *statistical generalization* when characterizing the frequency or portion of failures.

2.7 Related Work

The work by Kim and Ko [46] is the closest related work in terms of research method and goal. Kim and Ko [46] also performed an qualitative inspection of 30 tutorials across a variety of

[8]`https://www.khanacademy.org/`
[9]`https://www.w3schools.com/`
[10]`https://www.edx.org/`
[11]`https://codecombat.com/`
[12]`https://www.alice.org/`

domains. However, their method differed in how they characterized tutorial content. In the study, the authors made an assessment of whether tutorials fit best practices in pedagogy, such as connecting to learners' prior knowledge, and encouraging meta-cognitive learning. They found deficiencies in all these categories across a variety of tutorials and domains. In both Kim and Ko [46] and our work, an overarching goal is to assess the quality of tutorials; however, whereas Kim and Ko [46] view quality through a pedagogical lens, we view quality through an *executability* lens, with a longer-term goal of automated and continuous testing.

Studies have characterized the pain points of both tutorial takers and tutorial creators. Tutorial takers stumble when tutorials contain missing dependencies or steps [60], do not explain unexpected errors or side effects, and have unclear adaption paths for tailoring content to different goals [49]. Head et al. [36] interviewed 12 tutorial creators and discovered pain points related to duplicate instructions and composing and reusing code fragments from a working example. Furthermore, the authors created an interactive tutorial author tool, Torii, which allowed authors to split, annotate, and link tutorial content. Our study provides empirical evidence validating several of these concerns, and provides additional techniques for allowing continuous testing of tutorials.

More broadly speaking, our study relates to other studies that have examined the executability of code artifacts. For example, a recent study by Pimental et al. found that only 24% of Jupyter notebooks could be executed, and only 4% had reproducible results [73]. Similarly, only 25–27% of Python code snippets found in Stack Overflow posts [41, 98] GitHub gists [39] are executable. In these empirical studies, a common factor for low rates of executability include missing configuration information, such as dependencies, and software decay due to reliance on older versions (e.g., the code only works with older versions of APIs). Our observation of documentation rot is closely related to other forms of software decay, such as unavailable urls in source code [35]. In summary, executability is a useful property for understanding the quality and replicability of software artifacts, including those found in software tutorials.

2.8 Discussion

Our findings demonstrate numerous ways in which tutorials contain instructions that are not directly executable. When naively executed, we found only 26% of code blocks inside tutorials ran successfully, consistent with other experiments on executability of other

software artifacts [73, 41, 98, 39]. Human annotation was necessary in order to identify responses to interactive input, supply missing implicit information such as those that need to be executed with privileges, and perform actions such as saving content as a file on a particular path. Although executing human annotated tutorials substantially improved executability, numerous issues which could not be simply resolved by annotation persisted, revealing underlying issues with quality and accessibility of instructions.

In the remainder of this section, we present design implications for technical writers, toolsmiths, and software engineering researchers that can help reduce some executability difficulties in software tutorials.

2.8.1 Implication I—Provide Accessible Alternatives for Tutorial Takers

We observed several accessibility barriers that limit who can fully take advantage of the learning experiences offered by tutorials, such as inaccessible resources (Section 2.5.3) and interactive prompts (Section 2.4.1), which can increase the difficulty of successfully executing a tutorial (Section 2.5.2).

Not every tutorial taker will have readily available access to costly infrastructure resources. Several tutorials required resources such as registered domain names and extra features, such as additional disk volumes, object-stores, and clusters—all expenses that can quickly exclude disadvantaged and non-traditional learners who do not have the resources necessary to obtain them. Professionals can be impacted as well. A tutorial taker may be working on a prototype or investigating possible platforms for a product, and may be weary of making large investments in order to follow a tutorial that may not even work.

Tutorials can inadvertently exclude novice learners by omitting details that are important for shepherding cautious learners through their first learning experience. Research in computer education has found that learners with lower self-efficacy have more difficulty handling unexpected and exploratory behaviors [4], such as interactive prompts [49]. Interactive prompts can be difficult for learners who do not know how to response exactly, especially if they are first learning a new tool or command. For example, commands such as '`mysql_secure_installation`' or '`lxd init`' can ask up-to dozens of prompts, including complicated configuration options such as bridge networking. We found many instances of tutorials which simply asked the reader to *answer the prompts* or *follow the rest of the prompts* without giving details for what should be the response to each prompt. When unguided, such prompts can be frustrating and overwhelming for novices.

Step 1: Setup the FQDN (fully qualified domain name)

As required by Zammad, you need to properly setup the FQDN on your server instance before you can remotely access the Zammad site.

Use the `vi` text editor to open the `/etc/hosts` file.

```
sudo vi /etc/hosts
```

Insert the following line before any existing lines.

```
203.0.113.1 helpdesk.example.com helpdesk
```

Figure 2.12: Using `/etc/hosts` file as an alternative for requiring a domain name in the instructions.

Learners desire more accessible resources that leverage their background [87] and providing alternative formats for flexibility in learning [11]. Surprisingly, simple steps can provide improve accessibility for tutorial takers. For example, many tutorials required prerequisites, such as having a registered domain name, before the reader can follow the steps of the tutorial. However, we found a few tutorials (figure 2.12) that offered a simple alternative if this was currently unobtainable for the tutorial taker. The tutorials instruct the reader to update their `/etc/hosts` file, which allows routing the requests of the domain name to a local IP address, therefore eliminating the requirement for a registered domain name. Similarly, commands with interactive prompts may not always be asking for configuration options relevant to the tutorial. When possible, interactive prompts can be reduced by using default values, or providing command flags, such as `-y` or `-q`. As a result, tutorials can reduce uncertainty for tutorial takers, and improve the ability for learners of all backgrounds to learn.

2.8.2 Implication II—Invest in Automated Tutorial Testing

We observed several issues that impact the quality of tutorials, such as documentation rot, missing and volatile instructions (Section 2.5.3) which resulted in lower execution rates (Section 2.4.1 and 2.5.2). "*Tutorial maintenance is inherently difficult...our tutorials are written by and for humans, so there's always a chance that there will be errors (even with all the work we do to minimize them*" (I2).

Automated tutorial testing can help if tutorial creators are willing to make minimal investments. For example, tutorials at LEARNK8S[13] uses annotations on their code blocks for rendering output (figure 2.13). These can be extended to support annotations that assist with automated execution:

```
1  Start the service.
2  ```bash|user=root|prompt='$'
3  systemctl start rot13
4  ```
5
6  Test the service by typing in `Hello, world!`.
7  ```bash|input='Hello, world!'|expect=1
8  client$ nc -w 1 -u 127.0.0.1 10000
9  Uryyb, jbeyq!
10 ```
```

Figure 2.13: Sample of annotations used by tutorials at LEARNK8S for rendering code blocks

Another place to invest in improving executability of tutorials is summarizing, updating, and presenting file content (Section 2.4.3). Tutorials can show intermediate steps of updating a file, but should include a final content block, or offer programmatic methods of displaying and updating content, such as "jq which automatically filters out the output and give what is needed" (I1).

Finally, tutorial testing can be complemented with other sources of testing and feedback. For example, Drosos et al. [20] deployed pain-scale surveys in an online learning environment to automatically determine which programming features were frustrating to learners, and Mysore and Guo [62] explored using tutorial profiling to identify problematic instructions. These measures can be combined with "*using data such as page bounce*

[13] https://github.com/learnk8s/learnk8s.io/

rates, shares, whether the user bookmarks a tutorial, whether the tutorial is 'ripped-off' or translated into other languages, etc" (I6).

2.8.3 Implication III—Provide Self-Contained Tutorials.

We observed tutorials that made simplifying assumptions about environment and tutorials that required running other tutorials that sometimes could "*run multiple levels deep*" (I3). Furthermore, when tutorials do not explicitly control their dependencies, documentation rot could lead to breaking changes within the tutorial (Section 2.5.3). Many tutorials that we analyzed included multiple links to other tutorials which explain how to setup a prerequisite but it is not clear which link should be used by the user, or if those tutorials will also have their own prerequisites.

One informant mentioned in recent tutorials they've "*tried to pull all dependencies up and "flatten out the dependency tree" so that it's clear up-front each step you need to take first*" (I3). We observed several Vultr tutorials, where all dependencies could be installed in a single tutorial[14]. Another informant described their efforts to better packaging tutorials: "*If the tutorial is using a VM, we package all dependent files and artifacts such as Docker images within the VM. When we depend on an external script, we fork the script and host it alongside the documentation. This is similar to your finding to **avoid use of external services as much as possible***" (I5).

Other interesting approach would involve automatic inference and construction base images for running tutorials. For example, DOCKERIZEME [40] is a technique that given a Python code snippet, can automatically infer system and transitive dependencies required to successfully execute a code snippet. Such a technique could be adopted to determine the dependencies necessary for running a tutorial. Similarly, a tool such as OPUNIT [56] could provide a simple mechanism for determining whether or not prerequisites were appropriately satisfied before starting a tutorial.

2.8.4 Implication IV—Be Intentional on Whether Instructions are Human-Readable or Machine-Readable

We observed several executability barriers, such as placeholders in commands (Section 2.4.3), conditional logic in file content (Section 2.4.3, interactive prompts (Section 2.4.1), and "*tem-*

[14] `https://www.vultr.com/docs/how-to-install-bookstack-on-ubuntu-16-04`

plate values are inconsistent between tutorials and rarely machine readable" (I6), which needed extensive human interpretation (Section 2.5.1) to overcome. These executability barriers often involves, human-readable steps, "*commands and file contents with placeholder values that should be changed by the reader, all for the purpose of providing additional information and context*" (I2), such as *fill in a password* or *replace this IP or MAC address*. One tutorial even asks the reader to go to a website, locate a download link, and use it in a `curl` command. Tutorial creators need to consider whether these human readable steps are intended to be truly educational or an unwanted learning barrier that reduces the accessibility of the tutorial.

Mirhosseini and Parnin [56] reported their experiences teaching over 400 students skills related to automatically configuring computing environments in a graduate university course. When following step-by-step tutorials, such as the instructions for configuring a mattermost server [15], students often struggled to detect errors related to incorrectly executed commands, or malformed configuration files and lacked the appropriate debugging skills to diagnosis them. For example, it was common to update the wrong section of a configuration file, especially when the option was repeated across multiple areas. Professionals also struggle to reuse material from online resources, due to configuration and documentation issues [23], being uncertain on whether the material is up-to-date [11, 95], and determining what portion might require manual adaption [16, 95].

Machine-readable instructions can have several benefits for learners. They can ensure learners are more likely to reach a desired state regardless of skill level. Furthermore, readers can more quickly and confidently execute instructions, providing them to safe foundation to experiment and adapt instructions to their learning goals. There are several steps that tutorial creators can take to ensure a better experience for tutorial takers while maintaining machine-readable steps. Tutorials can "*make it possible for users to fill out the variable values and have it flow into the tutorial code*" (I3). For example, if the tutorial needs to replace the text `sammy@openvpn_server_ip` inside commands, there can be an interactive component in the tutorial that allows the user specify the desired value for this parameter, which is then automatically propagated throughout the tutorial instructions. Finally, "*it would be interesting if there were a standardised format for storing tutorial data, which would allow one piece of structured data to both be rendered into a human-friendly page, but also interpreted by a machine and automatically executed*" (I6).

[15] `https://docs.mattermost.com/install/install-ubuntu-1604.html`

2.9 Conclusion

In this study, we conducted a mixed-methods study through an empirical study on tutorials collected from various online sources, and through a qualitative analysis of tutorials. We first used a naive execution strategy to determine a baseline level of executability. We then inspected code blocks in tutorials and devised a set of annotations that could be applied to improve their executability. By using human-annotations to execute these tutorials, we were able to obtain a higher rate of executability, but still observed a significant amount of failures in overall tutorial execution. A qualitative inspection of tutorial failures and peer debriefing with tutorial creators revealed various issues which prevented successful execution and impact the accessibility and quality of tutorials. We discuss possible strategies for improving software tutorials, such as providing accessible alternatives for tutorial takers, and introducing automated tutorial testing to ensure continuous quality of software tutorials.

CHAPTER

3

INVESTIGATING INSTRUCTOR PAIN POINTS AND DESIRES

This chapter is based on material originally published in SIGCSE 2023 [58].

3.1 Study Rationale

After identifying several barriers for the tutorial creators which resulted in inconsistencies and failures of the tutorials, I then conducted this **investigation of CS instructor obstacles, workarounds, and desires** [58] which involved interviewing 32 instructors. The goal of this study is to clarify the common tasks that takes instructors' time, the possible pain points in completing these tasks, and how they (wish they could) overcome these pain points so far. In this study we share implications of our findings and provide suggestions of practices or tools that may be helpful to instructors, shortcomings of solutions that instructors currently tried, as well as barriers in adopting some solutions.

This study satisfies part of the thesis by better understanding frictions related to **prepa-**

ration and delivery of educational material, as well as discussing the **needs which may be addressed with the help of ClassOps**.

3.2 Methodology

To understand the challenges of computer science instructors, how they tackle those challenges, and what remains as a pain point we conducted a semi-structured interview with 32 participants. In this section we explain details of our qualitative study which was designed to answer the following research questions:

- **RQ1: What did instructors wish they could change?** Can instructors suggest what they would like to change that is not possible right now? What can we conclude with a high level view of these issues and the current resources available to instructors?
- **RQ2: What are current attempts of addressing pain points?** What are the workarounds instructors use to reduce or solve their pain points right now? What software solutions exist and what research has been conducted?

3.2.1 Interviews

We conducted semi-structured interviews with 32 computer science instructors to address our research questions. In this section we share details about participants and the interview process.

Demographics

In an attempt to reduce biased results toward any specific group of participants, we selected interviewees with varying teaching experiences as university instructors. We selected these instructors from 26 different institutions ranging from large R1 doctoral research universities to smaller institutions such as liberal arts and community colleges. To be more specific, interviewees included 3 adjuncts, 7 teaching faculty, and 22 tenured/tenure-track professors from 5 countries (25 from USA, two from India, and one each from Canada, Germany, Netherlands, Mexico, and Austria).

Recruitment

We recruited the participants for the interview by first selecting institutions and then recruiting from their affiliated instructors. We selected 26 institutions and connected with a sample of their instructors who were actively teaching in the same or previous academic term. We then used snowball sampling by asking our interview participants to help us connect with their colleagues or anyone that they think can have relevant experience.

Protocol

We conducted 30-minute semi-structured interviews virtually while transcribing and taking notes. During the interview, we followed a script which included questions about:

- Lecture structure and deliverables of students
- Availability of resources such as TAs and tools
- Time spent on materials, grading, and lecture preparation
- Techniques to improve learning experiences
- Most painful aspects of running courses
- Thoughts about potential solutions to pain points

We dynamically revised our script template based on the responses that we received from instructors to be able to extract more meaningful information from the conversations and understand motivation and importance of what they currently do or what they wish they were able to change. We intentionally asked about the pain points last, because at this stage, we have collected supporting information for why something can became a challenge or pain point, and discuss possible solutions.

3.2.2 Interview Analysis

Using our interview notes, we first focused on the last topic which was "*challenges and wishes*" of the instructors. We analyzed the instructor responses through thematic analysis [9] by reviewing the transcripts, coding the parts of interest, and then finding themes in the coded transcripts. Next we reviewed other topics covered in the interview transcripts to add supporting themes.

3.3 Findings

In the analysis of our semi-structured interviews we learned about the pain points of instructors, the workarounds they use, and what they wish to do which is not possible at the current time. To provide context to our research findings, we first present an overview of how the participants spend their time in regard to their courses:

Content creation and maintenance: Instructors typically spend 6-8 hours, per class meeting, to create the content for a course. However, most of the interviewees mentioned this is only for the first time they teach, and in the future terms they focus on content maintenance which takes 2-3 hours per class meeting.

Lecture preparation: Instructors reported spending 10 minutes to 2 hours on lecture preparation depending on if they have taught the course before, their general teaching experience, and the tools they use.

Helping students: Instructors reported spending considerable time helping students individually or in small groups. Moreover, they invest even more time into creating efficient infrastructures for students to get additional help (e.g., TAs, study groups, online discussion boards, etc.).

Grading and feedback: Grading and giving feedback largely varies based on availability and skills of teaching assistants (TAs) as well as how much automation has been adopted in the course. Instructors reported spending up to 50% of their time on grading.

In the remainder of this section, we share the findings of this study to answer our research questions regarding pain points and workarounds.

3.3.1 RQ1: What did instructors wish they could change?

To answer our first research question, we asked instructors about challenges, pain points, and what they wish they could change. We categorized the responses into the themes below:

Where are students struggling?

Instructors face substantial barriers in understanding what students are struggling with and where they are spending their time. In particular, instructors often only see the final output (e.g., homework submissions) but do not see the process that students go through to arrive at their solution. P27 framed this clearly: "*The challenge I have is knowing what students struggle with outside of class. I can't see where they get stuck and many don't ask*

questions." Similarly, P17 discussed the need to know where a given student is: "*I want to see the progress for one student. They might be making the same mistake over and over in different assignments.*". P21 expressed that if they could answer these questions, then they could "*apply some pedagogy there*". P30 told us "*students are doing lots and lots of things but I can't process all they are doing*" and that it is a "*lot of pipelines all strung together*". Only possible option at the moment is "*to scour all different places to understand students progress—code, Q&A, quizzes, git logs, etc.*".

Instructors mentioned they want to "*do things more in class that guarantees that [students] are not just sitting there silent. But kind of force them to wrestle with the ... issue on the on the table*" (P5). However, the current tooling available to instructors doesn't allow them to verify this live during class sessions. P5 continued, "*I basically got the out of class situation like that covered. It's the in-class version...*".

Answering students' questions

Students do tend to ask many questions, possibly in an attempt to overcome their struggles. However, these questions can overwhelm instructors and TAs during class, at office hours, through email, and through other software like Piazza. P18 shared their experience: "*Although the presence of tutors helped addressing some of the queries raised by the students, it was difficult to address all of them in an organized fashion, as much of it was lost in the barrage of questions that were raised.*". They continued, especially in a large course, it is "*difficult to pay attention to individual doubts of students*". Furthermore, the questions are "*repetitive in nature*" (P17) and "*variations of the same*" (P25). To complicate this further, questions are often asked in batches, or as P19 put it, "*If they don't understand an assignment or get stuck, they will wait until last minute or after the due date to say something.*", which puts a strain on instructors and TAs to respond promptly.

It is particularly problematic to troubleshoot technical issues related to the student's computer or development environment. P23 said that the hardest questions from students are of the type, "*Why isn't Docker working on my laptop anymore?*" or "*Why does Python no longer exist?*" These questions are also "*the most painful thing*" for P23's teaching assistants to address.

Limited TA support

Towards supporting instructors in answering questions and identifying where students are struggling, departments will hire teaching assistant. However, these TAs are a finite and highly sought after resource in departments. Many institutions, especially community colleges and smaller institutions, "*struggle with TA support*" (P17). In fact, several instructors revealed that they have little to no TA support beyond the large introductory courses. As P3 said, "*TAs... I wish we had those*".

Instructors mentioned that even when they do have TAs, they may require considerable time to manage and may not be reliable. As P17 put it, "*TAs have their own biases ... some TAs are not mature programmers*". They continued that in some cases TA are limited by experience, for example "*TAs sometimes only run the unit tests and never read the code, [so] two submissions that were nearly identical, but one got [high] marks and the other got [low] marks*". Although P26 said they had sufficient TAs in their courses, they warned that the TAs also ask them a significant number of questions.

Other universities have restrictive policies on the work that TAs can perform. For example, "*in terms of grading, we have kind of a policy that, TAs are allowed to grade multiple choice questions, but for other questions like open-ended questions or coding problems, they're not really expected to grade those unless [instructor is] there to supervise them*" (P10), so as a result it is just easier for the instructor to grade directly. In a different institution, the instructor explained "*one of the university policies is [that] we don't allow students to grade other students work*" (P3). Similarly, another instructor shared that in their department, TAs are only allowed to lead group study sessions and manage in-class activities, but cannot grade.

Grading & feedback

Grading is a time-consuming tasks that nearly all of the instructors mentioned. There are many dimensions to grading, including returning grades to students in a timely manner, providing high-quality qualitative feedback, covering all artifacts that students produce, transparency in why points were taken off, and being consistent and fair across students and assignments. Instructors gave conflicting interpretations of grading as either a chore or an opportunity. For example, P9 said, "*grading is probably the biggest burden of the courses*" and P20 said, "*grading is an impossible task*". In contrast, P13 prefers to grade things themselves even if they has TAs "*because [of] the feedback I can get from ... their*

homework and assignments".

A particularly challenging aspect of grading is developing, sharing, and adhering to rubrics. P27 designs their grading rubrics to minimize time spent grading. P20 said they makes a new rubric that focuses on different metrics for almost every assignment but knows this "*is not a great solution*". P23 emphasized "*transparency in grading and feedback is a priority*" and said that a good rubric is the key. They continued about the need for high-quality rubrics, "*in an ideal world, they can self grade or know exactly why they got their grade*" and one way they have done so is through "*boolean*" grading that eliminates subjectivity. However, even this is still time consuming for P23 and their TAs.

Although automated grading does exist, the "*human touch*" is valuable to instructors, especially as they deal with increasing enrollment. P22 said, "*our program is pretty well scaled, and it is a huge challenge to give quality feedback.*". P23 remarked that they wish for more automation of mundane tasks, but expressed that they are strongly opposed to automating feedback to students: "*I think this is the wrong direction for education. Striping away community and humanity from learning.*". Other instructors (P26, P30, and P32) shared a similar sentiment that their contribution is that of one-on-one feedback and to help the students build a community in the classroom. The COVID-19 pandemic reinforced the need for these personal interactions for P32.

Course material preparation

Another common challenge for instructors is the high cost of developing or adopting course materials to use as examples, assignments, lecture notes, quizzes and supplemental reading in their courses. P15 mentioned they cannot change the whole assignments often because they "*just don't have the resources*". Additionally, "*finding those is one of the biggest challenges, because ... there are so many different resources here and there. This does that, this does that ... [instructors] just don't have the time, especially at a smaller institution*" (P3). Having more examples or variety of assignments could benefit students both as additional resources as well as a way to prevent plagiarism.

Shifting courses to online or to hybrid modalities, such as a response to the COVID-19 pandemic, presented additional barriers and work for instructors. For example, P16 mentioned the problem of keeping students engaged, "*everyone's there at first and then it's just like monotonically decreasing. Until I get to end of semester, then it's like 20% or 15% of the students are there, and so that's my biggest problem right now of like how do I get people*

how do I incentivize lecture without just forcing them to go by holding their grade hostage." They continued, "*I've checked and there's very little few few people watching these videos*". As P8 explains, moving content online is an upfront investment, but may not take much time to maintain: "*When I was making the videos it was taking significant amount of time to make the videos and once those are done, then I don't spend very much time preparing now*". These challenges were also published in a recent case study of transitioning a course online because of COVID-19 closures [12].

Administrative tasks

There is managerial and administrative tasks involved with running any course, or as P16 put it, "*grunt work*". Examples include managing the social dynamics in class, preparing multiple course delivery formats, accreditation tasks, enforcing academic honesty policies, assigning teams, dealing with LMS quirks, transitioning to new software systems, and managing individual student accommodations. While these tasks may not necessarily be the most practical use of an instructor's time, instructors usually have to do these tasks because of other limited resources.

Time spent on administrative tasks often prevented instructors from making improvements in the course. For example, P9 told us "*I would perfect the projects, if ... I could pause the clock.*", P4 told us they want to "*make things as close to industry as possible to help with smooth transitions from student to software engineer*", and P8 wanted to adopt mastery learning [25] model and "*have them do it again and do it again ... fix it and resubmit [until] I'm satisfied with it ... it is a really important learning process ... but takes a lot of time and effort on my part*". P22 expressed the desire to maintain notes on improvements for future versions of the same course but added that doing so is a chore.

Furthermore, identifying when students cheat was a concern for many instructors, especially in regards to moving courses online for COVID-19 and scaling to support higher enrollments. P19 explained that they have a 30-page syllabus to make it clear what is allowed and not allowed, along with quizzes covering academic dishonesty. P17 and P29 mentioned that it is not feasible for them to put as much time into checking for plagiarism as they should, but it is important to them.

3.3.2 RQ2: What are current attempts of addressing pain points?

To answer our second research question, we analyzed and categorized the current efforts used by instructors to address the pain points they mentioned in the previous section, which includes software tools and pedagogical techniques:

Interactive textbooks/exercises: Two of the instructors that we interviewed had created their own interactive textbooks, one based on Jupyter Notebooks and another using a proprietary interactive platforms that they developed. As instructors explained, interactive textbook "*fixes a problem we had before*" (P14) which was keeping content up-to-date. Interactive (executable) textbooks allows instructors to ensure the material remains up to date through a continuous integration mechanism. P14 said programming related textbooks "*come with all the usual well maintenance problems of code*", and with their interactive textbook "*when somebody, somewhere out there in the world changes some Python package, such that my code ... no longer works, I get notified the day after*" (P14).

Additionally, instructors mentioned they were able to use their interactive textbook to encourage a more active learning experience. With a traditional textbook students can read the code "*but well, not much fun to read code, you won't be able to execute it. But [in interactive format] it's more fun to actually toy with it*" (P14) right in their web browser, while they read the course material or watch a lecture. However, a recent study found that students had significantly fewer interactions with their eBook and fewer study days during the pandemic than they had before [99].

Online IDEs and code visualizers: Some instructors adopted online IDEs and code visualizers which may increase engagement in class and in assignments. Such tools have been previously studied and found to create a faster feedback cycle for students [45] and eliminate computing environment configuration issues. P12 said "*we want to give the students a uniform platform, we don't want them to need to switch to [a separate IDE] ... WebLab is really an IDE in the browser. So we code in a window in a browser and we push compile and outcomes everything", they continued "We also have unit tests in there ... The students can program and they can run their solution against the unit tests that we provide.*"

Another instructor from a smaller university mentioned they adopted using Java Visualizer[1] which an online tool based on Guo's Python Tutor [32]. This online tool allows students run a code snippet in their browser, visualize it and share their snippet using a permanent link generated by the tool. In this course, Java Visualizer was used for teaching the coding

[1] `https://pythontutor.com/java.html`

examples, as well as student submissions. When we asked the instructor what pain point they tried to address by adopting this tool, they mentioned, Java Visualizer has helped their students who are new to programming "*so that they can see what is happening when they're coding in memory and they can immediately see what's occurring*" (P3). Additionally, they mentioned some of their students did not have access to a capable computer, so having everything online in the browser alleviated environment issues. Other institutions aimed to eliminate these technical issues by standardizing the computers used or by moving to cloud-based software (e.g., P3, P10, P17, P24, P25, and P27).

Automated grading: Instructors have adopted several different tools and approaches for automated grading. Two examples of automated grading implementations used Web-CAT[2] [21], and GitHub Actions[3] to run an internally developed grading tool. This automation setup requires significant time investment by the instructors and makes changing the content more difficult [51]. However, in both cases the instructors were able to reduce the grading time and the TAs could simply "*look at the code to catch things that automated test does not catch*" (P5) instead of manually checking everything. This has allowed the instructors and TAs use the time they saved to help students in the learning process. Another example of that is Harvard's Check50 tool [86], which is an API-based tool that gives feedback to students before assignments are submitted as well as automates portions of the grading. This approach may also help the instructor promote mastery learning [25]. As P5 explained with this approach, their students "*can submit as many times as they want. Looking at the feedback, looking at the results of those tests and fix anything that those tests address and resubmit.*"

Flipped classroom: Several instructors mentioned they switched to a flipped classroom method of teaching or some hybrid model which has enabled more time for in-class interaction with students. This style involves assigning readings and pre-recorded lecture videos for students to watch and use the class meeting time to answer questions and work on interactive live coding. This change of teaching style was primarily adopted as a result of COVID-19 pandemic in 2020 [12], however instructors decided to continue to use this method even though universities are back to normal operations at the time of our study. As P15 explained, they had two main motivations to keep using a flipped classroom. First is that it has allowed them to have extra time to dedicate for interaction with the students in the class. And second is the "*the feedback ... from students the last couple of years is that they*

[2] `https://web-cat.org/projects/Web-CAT`
[3] `https://github.com/features/actions`

actually prefer having this lightweight flipped classroom model where they can just watch the theory part, they can scroll back, they can pause they can try stuff on their own. And in the live lecture they get the interaction with the lecturer".

Peer instruction: One of instructors mentioned they adopted extra "office hour" time with small groups of students to run peer instruction [75] sessions which was seen as a positive change to increase understanding of concepts and engagement to work on examples together. However, as P8 explained this approach also had some weaknesses, "*it took a lot of time because I had to have so many meetings to have meetings at small, so it was not very efficient use of my time.*" Several instructors shared that their departments now require student-led study groups for core courses, which they believed to be a positive addition without taking time from instructors (P26, P28, and P32). Research supports the use of UTAs as an effective means to scale classes through peer instruction and student groups [59]. Alternatively, one school experimented with using a department-wide question and answering site (essentially a private Stack Overflow), with promising results [42].

3.4 Limitations

Our qualitative approach in this study may introduce certain limitations to the results. First, the findings may carry a level of interpretation of the researchers as they analyzed the responses in the interviews using thematic analysis. Second, our methodology uses *semi-structured* interviews and by definition, the questions asked during the interview or their order may slightly change based on the conversation. These variations may have some effects on the responses we received from instructors. Third, we covered the transitional changes in courses material and teaching styles as a result of COVID-19 pandemic as well as some associated pain points (3.3.1), however, it is difficult to predict whether any of those changes would revert to pre-pandemic norms over time and when. Finally, although the number of our participants was larger than most of the related studies, our findings may not be representative across demographics.

3.5 Related Work

In both Lau et al. [51] and our work, the overarching goal is to find challenges that instructors experience. However, Lau et al. [51] focuses on challenges related to *maintaining content*

in a *large scale data science course*, while we focus on challenges related to *assessments, integration of technology in class and coding* that can potentially effect pedagogy in different computer science classroom and institutions sizes. Lau et al. [51] method also differs in using case studies and previous experience while we conduct semi-structured interviews with computer science instructors.

Other related works with similar goals are from Yadav et al. [96, 97]. These studies have a goal of understanding challenges and experience of computer science teachers in K-12, with a focus on the increasing need for training new instructors with the growing computer science education demand. Similar to this study, Yadav et al. [96, 97] work used interviews with the instructors and qualitative analysis as the method for conducting the study. Some of the findings from this study, and the work by Yadav et al. [96, 97], such as grading challenges, are common and hold relevant. More broadly speaking, our study can also relate to other studies that offer a solution to pedagogy challenges in computer science classrooms.

3.6 Discussion

Our findings categorized the pain points of CS instructors, and described some of the workarounds that instructors have tried in their workflow so far. We present implications for CS education researchers, instructors, and toolsmiths for reducing instructor pain points, and improving learning experiences for students.

Helping instructors surface student struggles: Instructors had low visibility into the student struggles (3.3.1), seeing where they got stuck, or identifying issues and answer questions in a timely fashion (3.3.1). Additionally, when instructors provide feedback or grade submissions (3.3.1), gathering information about student activities involved many "*mechanical actions to get all the info and put it in one spot*". For example, if an instructor wanted to verify if students performed effective code reviews of pull requests and made sufficient contributions to a project, they would have to navigate and visit each pull request to manually inspect review comments, and then navigate through the commit logs, possibly accounting for nuances such as commits on different branches or possible peer-coding activities. Often surfacing simple information would have helped instructors, for example, P16 used mastery grading, but their gradebook system did not display number of attempts, making it difficult to see if a student was "*trying a bazillion times*".

In general, instructors need help discovering and consolidating data that's "*in a bunch*

of places to better find the students that are struggling" (P16). Some research efforts offer a promising start. Mysore and Guo [61] profiled student activity on tutorials and overlaying heatmaps of activity hotspots, encountered error messages, and embedded screencast videos of user actions. Adopting and extending this approach in other contexts, can help surface unknown struggles and provided actionable feedback for improving instructional material and assignments. Learning dashboards [92] have been used in a variety of contexts, including monitoring progress on assignments [18], visualizing group work contributions [28], and guiding students in self-reflection [83]. However, capturing these activities across a suite of tools, customizing and personalizing analysis to course content, and deploying these systems remain a challenge. Finally, simple interventions, such as weekly surveys can identify struggling students or teams [76].

All together, visibility into student struggles can help instructors identify misconceptions, sources of frustration, and problems in instructional material. Once struggles are better known, instructors will have the opportunity explore "*less defined assignments and then later dive in to see what they're doing*" (P21) and use the insights as "*guard rails for rabbit holes*".

Helping instructors leverage automation for class operations: Instructors frequently spent time on tasks that were "*repetitive in nature*" (P17) and "*variations of the same*" (P25), including answering questions (3.3.1), preparing assignments and quizzes (3.3.1), and administrative tasks (3.3.1). For some instructors, these problems could be overcome by scaling with more TAs (25 TAs on average for P12's class), but for many other instructors, they lacked TAs or TAs were limited in what actions they were allowed to do (3.3.1).

Automation of class operations offers a potential way to scale teaching efforts, allowing instructors to spend more time with students that need help or developing new course material. While decades in the making [13, 85], today AI systems have improved and with the introduction of large-language models (LLMs), such as OpenAI Codex[4]. Such models have been used recently for automatically generating "novel" and "applicable" programming exercises [84], generating solutions for introduction assignments [22], generating final exams [101], and answering questions [47]. Along with traditional AI tutor systems (e.g., AutoTutor [67]), virtual TAs (e.g., Jill Watson[30]), and approaches such as automated program repair [100], these systems offer considerable support for reducing the effort involved in aiding struggling students, responding to questions, preparing assignments, managing TAs,

[4] `https://openai.com/blog/openai-codex/`

and grading.

However, automation brings its own challenges. Autograders may be viewed by students as unforgiving and opaque since they do not adequately explain why points were taken off. For example, P10 described when an interactive textbook gives a grade, it deducts points even if there is only a whitespace difference with the solution. They continued this is "*in my opinion, unreasonable and my students struggle so much with it and they spend hours trying to get the white space correct in their program when in reality that's not what I want them spending spending time on*". P17 shared a similar observation, that autograders are "*too harsh*" and still require a human to review the grades. They described an example where a student received a 0 for a submission to the autograder, but the cause was a minor issue that should have only resulted in a few points deduction.

Although instructors can automate their classes using LLM-based technologies, it also means students can automate their assignments. In fact, there is a growing concern[5] that Copilot, an intelligent code suggestion tool that uses Codex, can complete homework assignments in seconds with little conceptual knowledge. A professor recently wrote a satirical article on how to cope with students using Copilot in your class[6]. Others have taken to Twitter to discuss how to design "Copilot-proof" assignments[7].

Instructors stressed the importance of promoting a diverse and inclusive classroom, which automation may negatively impact. For example, P17 worried that when a student sees the feedback from their autograder, the student will lose confidence because of how strict the system is. P29 has trouble finding assignments and lecture notes online that are designed to be more equitable and inclusive. They explained that they do not want homeworks consisting of abstract problems to be solved in a terminal, but rather the homeworks should be culturally relevant. Furthermore, they stated that having a database or system to generate these assignments would push the classroom to be more diverse.

Helping instructors reduce friction in course delivery: Instructors faced obstacles maintaining code exercises and course material (3.3.1), troubleshooting technical issues in student's computing environments (3.3.1), keeping students engaged (3.3.1). Common workarounds, included outsourcing course content to interactive textbooks, and integrating cloud-based programming environments. Benefits included, monitoring student progress, supporting active learning experiences for students, reducing questions related to technical

[5] https://twitter.com/search?q=copilot%20homework&src=typed_query

[6] https://itnext.io/coping-with-copilot-b2b59671e516

[7] https://twitter.com/deliprao/status/1557913160140656640

environment issues.

However, these workarounds often did not completely resolve these obstacles and other desires remained. Instructors using existing interactive textbooks, such as zyBooks, could not modify material, and found content is often difficult to align with existing course structure and were "*pedagogically not nearly as solid*" (P5). P10 quickly realized students could not perform many basic operations, such as working with file systems, for their course because "*zyBooks is ... just kind of a simulation*". Similarly, P14, who made their own interactive textbook based on Jupyter notebooks, had to invest significant time. They explained, "*many things I had to invent and build myself. I had to build quite an infrastructure around this to make this whole thing workable for me*". They continued "*Jupyter Notebooks typically are not being used by programmers ... they are typically used by data scientists*" so it has a different purpose and it's "*awful in terms of code*" support out of the box. Finally, instructors often had to navigate usage limits as well as longevity associated with cloud-based solutions. For instance, P28's department adopted a cloud platform, however that platform "*disappeared a year or two after migrating to it*", and as a result, they became very hesitant to invest in adopting external tools.

Advances in educational technology seem like a game of whack-a-mole, addressing one challenge often brings another one. For example, while interactive textbooks have several benefits and reduce certain pain points, instructors now have to deal with significantly reduced engagement associated interactive textbooks [99]. Moving forward, educational technology needs a design that simultaneously addresses or blends several needs, such as improving student engagement, interactivity and liveness, peer interaction, and advanced computing environments. For example, a participant described a live document that embedded code editors, slides, and quizzes, backed by a real environment, and was shareable by instructors and peers "*that would be a perfect blend*" (P9).

3.7 Conclusion

In this work, we conducted interviews with 32 computer science instructors about obstacles, workarounds, and desires when teaching their courses. We identified several pain points, including limited access to resources, repetitive tasks, such as manually gathering information about student activities for grading, or answering variations of the same question, troubleshooting technical issues in students' computing environments, and endless

administrative tasks. Common workarounds included automated grading, peer instruction, outsourcing course content to interactive textbooks, and integrating cloud-based programming environments. Instructors identified several desires, including (1) more visibility into their students struggles, problem-solving process, and programming environments, (2) better support for automating repetitive tasks, and (3) improved interactive environments and streamlined course delivery. In general, we believe that investing in the educators will improve the learning experiences and quality of life for everyone.

CHAPTER

4

INVESTIGATING ISSUES RELATED TO COMPUTING ENVIRONMENTS

This chapter is based on material originally published in DEVOPS 2017 [55].

4.1 Study Rationale

In chapter 3, we found several pain points that instructors experienced. This study offers a solution to one of those pain points, *answering questions* and, more specifically, questions related to computing environment issues. In this study, we present Opunit, a tool for checking the properties of a computing environment. We then report students' experiences using Opunit to check their environments to resolve their issues.

This study satisfies part of the thesis related to the **delivery** of CS-Ed and our supporting **automated ClassOps tool**, Opunit, which **improves** instructors' ability to assist students regarding computing environment issues.

4.2 Background

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 [78], 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 [29]. 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 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" [1]. 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" [1]. 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 [69]. 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,

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

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.

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 (Figure 4.1). 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.

```
➜ Pipelines git:(master) ✗ opunit verify local

Checks

          Essential workshop tools

          version check
               ✔  node --version: 11.9.0 > ^10.x.x => true
          version check
               ✔  git --version: 2.20.1 > ^2.17.x => true
          version check
               ✔  curl --version: 7.54.0 > ^7.54.x => true

          Demo hook

          reachable check
               ✔  [hook-demo/.git/hooks/post-commit] status: true
          contains check
               ✔  [...post-commit]  contains [google.com] status: true message: NA

          App setup

          contains check
                  If this fails, you may need to ensure you checked out submodule. `cd App` then run `git
submodule update --init --recursive`.
               ✔  [...main.js]  contains [express] status: true message: NA
          reachable check
               ✔  [App/] status: true
               ✔  [App/node_modules] status: true

          Pre-commit setup

          contains check
                  App is a submodule, its hooks are located in `.git/modules/App/hooks`.
               ✔  [...pre-commit]  contains [npm test] status: true message: NA

          Deploy directory setup

          reachable check
               ✔  [App/deploy] status: true
               ✔  [App/deploy/production-www] status: true
               ✔  [App/deploy/production.git] status: true
          version check
                  Install with `npm install pm2 -g`
               ✔  pm2 --version: 3.2.4 > ^3.2.4 => true

          Post-receive setup

          contains check
               ✔  [...post-receive]  contains [production-www/ git checkout -f] status: true message: NA
          reachable check
               ✔  [App/deploy/production-www/main.js] status: true

Summary

          100.0% of all checks passed.
          15 passed · 0 failed
```

Figure 4.1: OPUNIT's verify command to test pipelines workshop

- 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.

4.3 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.

4.3.1 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 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

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

to other instances of the same application still running in the 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 Figure 4.2 in red color.

```
{
    "a": "b",
    "c": "d"
}
```

```
{
    "a": "b",
    "c": "e",
}
```

```
{
    "a": "e"
    "c": "d"
}
```

Figure 4.2: Examples of students' broken JSON files shown 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 [93]. 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.

4.3.2 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.

[5] personal correspondence from industry

4.3.3 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.

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.

4.3.4 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 1GB 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/`

4.4 Opunit

Inspired by the properties that we identified, we developed an environment testing automation tool, OPUNIT. Figure 4.1 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.

4.4.1 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.

```
1 - group:
2   description: "Check node.js support"
3   checks:
4   - version:
5       cmd: node --version
6       range: ^10.x.x
```

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

[7] https://semver.org/

4.4.2 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 section 4.3 and each require different parameters which are explained in more details in OPUNIT documentation[8].

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.

4.4.3 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.

[8] https://github.com/ottomatica/opunit

4.4.4 Report

After OPUNIT verification checks are executed, the results are printed in the terminal window. Figure 4.1 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.

4.5 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.

```
➜ demo opunit profile CSC-DevOps/profile:519.yml
Using profile CSC-DevOps/profile:519.yml

Checks

        Essential development tools

        version check
            ✔  node --version: 11.7.0 > ^10.x.x => true
        version check
            ✔  git --version: 2.20.1 > ^2.x.x => true
        contains check
               Checking email is set for git commits
            ✔  [....gitconfig]  contains [@] status: true message: NA

        Shell utilities

        version check
            ✔  curl --version: 7.54.0 > ^7.x.x => true
        version check
            ✔  wget --version: 1.20.1 > ^1.9.x => true

        Editor support

        version check
               Visual Studio Code is a great editor for editing configuration scripts.
            ✔  code --version: 1.30.2 > ^1.30.1 => true
        contains check
               Syntax highlighting should be enabled in vim!
            ✖  [....vimrc]  contains [syntax on] status: false message: NA
        contains check
               Setting pastetoggle is useful when copying code in vim. Example `set pastetoggle=<F8>`
            ✔  [....vimrc]  contains [set pastetoggle] status: true message: NA

        Virtualization support and tools

        capability check
            ✔  [cores] expected at least: 2 actual: 8
            ✔  [memory] expected at least: 4GB actual: 16GB
            ✔  [virt] expected: true actual: true
        version check
            ✔  baker --version: 0.6.15 > ^0.6.15 => true
        version check
            ✔  vagrant --version: 2.2.2 > ^2.1.1 => true
        version check
            ✔  VBoxManage --version: 5.2.22 > ^5.2.18 => true

Summary

        92.9% of all checks passed.
        13 passed · 1 failed
```

Figure 4.3: Result of running an OPUNIT profile

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

The resulting output is shown in Figure 4.3. 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.

4.5.2 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`.

4.5.3 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 [82] 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.

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

```yaml
- group:
    description: "Pre-commit setup"
    checks:
      - contains:
          comment: App is a submodule, its hooks are located in
    ↪   `.git/modules/App/hooks`.
          string: npm test
          file: .git/modules/App/hooks/pre-commit

- group:
    description: "Deploy directory setup"
    checks:
     - reachable:
         - deploy
         - deploy/production-www
         - deploy/production.git
     - version:
         comment: Install with `npm install pm2 -g`
         cmd: pm2 --version
         range: ^3.2.4
```

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

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 4.1. 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.

Table 4.1: Follow-up Survey Responses

	% Agree	**Likert Responses**[1] 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.

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.*

4.6 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

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

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 [2]. 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.

4.7 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.

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

CHAPTER

5

INVESTIGATING NEEDS FOR CREATING AND DELIVEING COURSE CONTENT

5.1 Study Rationale

In Chapter 3, we detailed several pain points that instructors experienced. This chapter offers insights into how to address most of *course material preparation and delivery* pain points. In this study, we recruited nine instructors to use a computational notebook system, DOCABLE, in their courses, and report instructor and student experience through semi-structured interviews and surveys. We then formalize essential properties of tools in this space to guide the design of future ClassOps solutions.

This study satisfies part of the thesis supporting **preparation and delivery of CS-Ed material** with the help of **ClassOps**, by **simultaneously addressing several needs of instructors**.

5.2 Background and Related Work

Computer science instructors face numerous challenges in preparing and delivering course content [58, 51]. Many computer science instructors do not have sufficient teaching assistants to help with their workload. They often need to teach the courses in different formats (in-person, online, and hybrid) simultaneously and to groups of students with varying backgrounds and experiences. With other challenges in place, working towards improving the student learning experience, for example, through adopting more active learning components in the class, is even more challenging for the instructor. Additionally, the growing demand for computer science courses and lack of adequate computer science faculty puts additional strain on existing instructors by requiring them to teach more courses every semester or increase their class sizes. It is challenging for the instructors to find effective ways to teach the concepts, engage students in the process and build confidence by providing them with instant feedback while interacting with the content.

To better illustrate these challenges, consider a simple in-class activity in a CS2 Java course where students are learning to implement the `add()` method for a `LinkedList`. One instructional approach involves explaining the `add()` method using slides and demonstrating its implementation in front of the class. However, this approach does not necessarily create the desired learning experience because students are passive observers. Alternatively, the instructor may explain the `add()` method using slides and allow students class time to implement it themselves. This seemingly simple approach involves many additional tasks. For instance, students must create a Java project in their integrated development environment (IDE), copy a partial `LinkedList` class to this project, ensure that they have a driver function that can correctly reference this class, and still have a way to verify and get feedback on their implementation.

Moreover, the instructor must have a way to monitor the student progress, assign them participation points, and incentivize completion of this activity. However, given the limited time available in a class session, it will only be feasible to devote up to 10 minutes to such an activity, which is inadequate to complete all the tasks noted above. Currently, the instructors we have engaged with have been compelled by their time limitations to skip interactive in-class exercises for such topics, relying solely on passive student observation or using tools such as Google Forms to create in-class coding activities. However, platforms such as Google Forms are not designed with coding or educational pedagogy in mind, effectively restricting students to writing code in a limited environment without syntax highlighting

or execution feedback. Likewise, instructors can only assign participation points without feedback on correctness and based on students' attempted submissions.

Despite the challenges related to in-class activities, instructors must consider other aspects when preparing and delivering the teaching material. For instance, many studies suggest interactive course material can make students' learning experience more engaging and effective by promoting active learning [68, 33] and better critical thinking and conceptual comprehension [19] through blended learning method in activities [81]. Over the years, notebook systems have been proposed to satisfy such desires related to active learning. For example, Graasp [27] was designed close to two decades ago with a focus on bringing active learning to students' education experience. Jupyter Notebooks, although designed for and by data scientists, has been tested in various courses from biology and chemistry [89] to computer science to deliver Lectures/Discussions, Readings/Flipped classroom, Homework, and even Exams [34]. Similarly, Google Colaboratory, although defined as "*a data analysis and machine learning tool*" google-colab, has been used to overcome some limitations found in Jupyter by creating an online coding environment [8, 34].

Previous studies have reported inherent pitfalls of Jupyter Notebook systems in educational use cases that may confuse students. For instance, some technical issues that instructors may encounter are client-server mode, 'magic commands', hidden states, out-of-order executions [44], and in general, the significant infrastructure investment needed for using Jupyter for coding [58]. Similarly, a study by Lau et al. analyzed 60 notebook systems [50], which summarized the capabilities of each system with a technology-centric view for data science use cases. In this study, *we target CS instructors and use a task-centric approach to understand what they need to do with a notebook system.*

In this study, we experiment with a type of computational notebook system called DOCABLE, which has a pedagogical-focused design compared to most off-the-shelf notebook systems like Jupyter, aiming to address some of the instructor challenges (Chapter 3). What sets DOCABLE apart from a system like Jupyter is a more versatile computing environment support and a more predictable execution mechanism design which we will detail more in-depth in section 5.3. While computational notebooks have been widely used for data science, literate programming, and presenting instructions blended with code snippets, our focus in this study is on addressing the unique requirements of CS instructors and analyzing their feedback.

To evaluate the effectiveness of DOCABLE's design, we conducted a qualitative study in twelve classes involving nine instructors in four different institutions. Instructors had

complete freedom to utilize DOCABLE in their classes during the semester as they saw useful. At the end of the semester, we conducted semi-structured interviews with the instructors. Additionally, we administered a feedback and usability survey to the students, providing us with valuable insights and perspectives. The contributions of this study present:

- Analysis of feedback received from participants, highlighting the key themes that emerged from their engagements with DOCABLE and how effective it was.
- A set of essential properties for ClassOps tools in this domain, to guide design and development of more effective tools in the future.

5.3 The Notebook Systems: DOCABLE

In this work, we chose to study DOCABLE's effectiveness because it aims to address the instructors' needs that Jupyter and Google Colaboratory did not wholly satisfy. DOCABLE is still considered a notebook system in format, meaning it allows the blending of different types of components. The main differences are being a hosted software, supporting more versatile computing environments, and avoiding hidden states achieved by a more minimal approach in its execution mechanism. To provide more context, we briefly describe some of the unique design differences in DOCABLE compared to Jupyter Notebooks in the following subsections.

5.3.1 Content

DOCABLE supports a range of components, encompassing static and interactive elements. Static cells include formatted text, embedded videos, slides or third-party HTML content. Interactive cells, on the other hand, offer features such as code editors with execution support (Figure 5.1), terminals (Figure 5.3), multiple choice quizzes, and the ability to create interactive exercises like the ones depicted in Figure 5.2. On the other hand, the design of Jupyter Notebooks encourages poor coding practices because it makes it challenging to organize code logically [72] making the environment limiting for more experienced students [8].

Modules (splitting code into separate files)

Modules are used to split your code into separate files. Modules are written using the export keyword. Modules can have properties and methods. Properties are variables that belong to the module. Methods are functions that belong to the module. Modules can also have a default export, which is a special export that is executed when the module is imported.

Example:

Figure 5.1: A snapshot of a notebook on DOCABLE demonstrating: (A) a file editor that's editable by the reader and saved on the file system, (B) a live executable file which is also editable by the reader and allows execution with the help of the run button. In this screenshot, (C) `usingPersonClass.js` has been executed and the output is displayed.

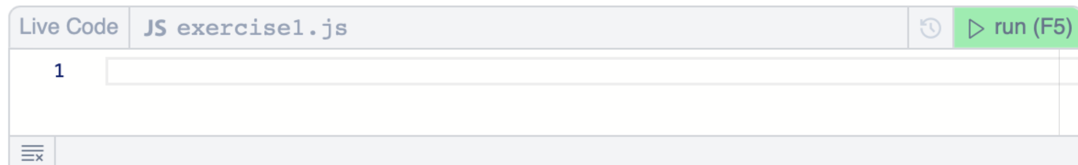
Exercise

As a quick excercise, solve these problems from your homework 2, but now using **axios** instead of **curl**:

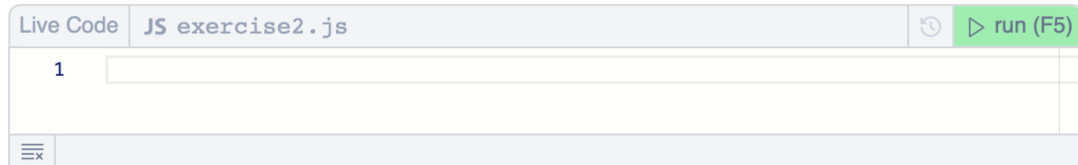
JSONPlaceholder is a free online REST API that you can use whenever you need some fake data. It is a great tool for testing and prototyping. Here we're going to use it to practice making HTTP requests using cURL.

Use JSONPlaceholder's documentation (https://jsonplaceholder.typicode.com/guide/) and cURL to answer the following questions:

1. Get the title of the post with id 5.



2. Get the name of the first comment of the post with id 5.



3. Delete the newly created post.

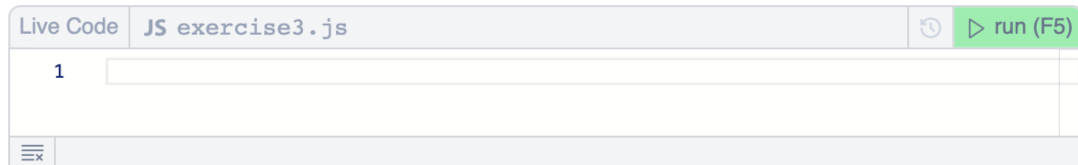


Figure 5.2: A snapshot of a notebook on DOCABLE demonstrating coding exercises that provide reader with an executable file editor where they can implement and run their answers.

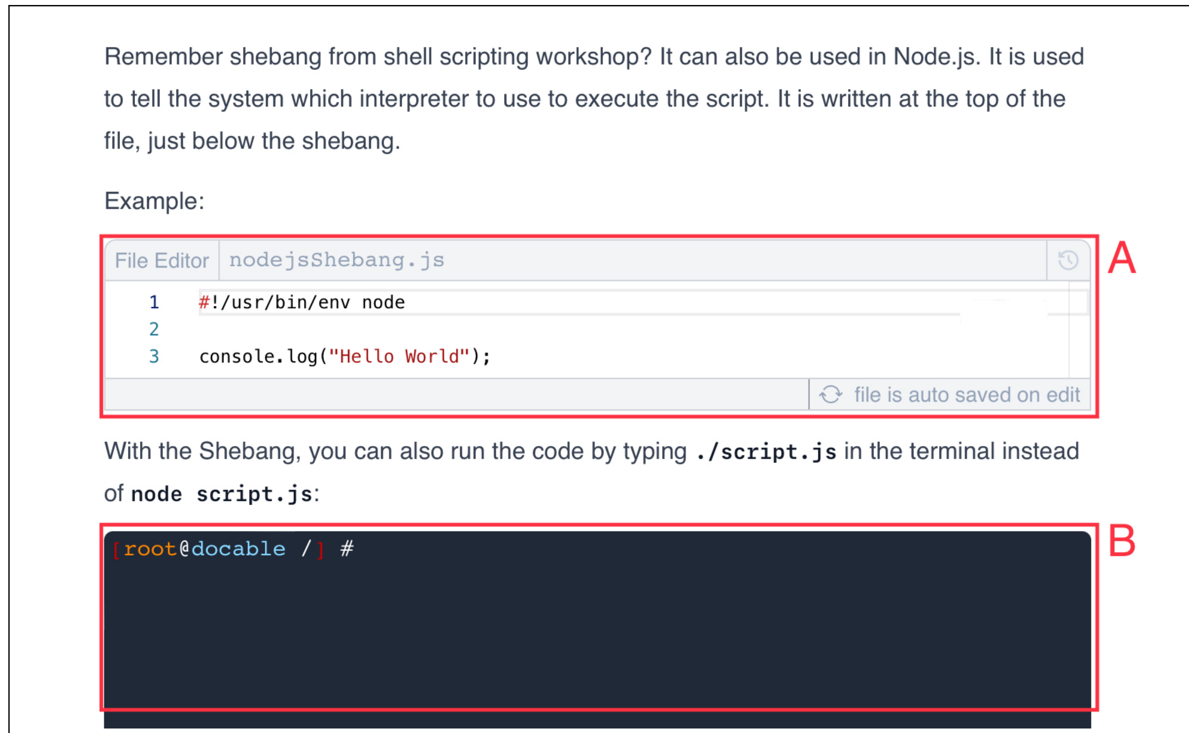


Figure 5.3: A snapshot of a notebook on DOCABLE demonstrating (A) a file editor that's editable by the reader and saved on the file system, (B) a live terminal connected to the reader's isolated cloud environment.

5.3.2 Authoring Experience

DOCABLE utilizes a visual editor interface to create the authoring user experience. Standard text, images, videos, and other static elements such as Google Slides can be added similar to a traditional document editor. Additionally, the interactive components can be added using the add menu (Figure 5.4) and then configured by the author (Figure 5.5). The author can further customize the computing environment of the notebooks that will be given to each reader, as described in the following section.

DOCABLE authoring experience also includes a new experimental feature that utilized AI systems (large language models) to help write a section of the page or generate an interactive exercise based on the covered topics on the same page (Figure 5.4). The generated content remains interactive, and the author can modify it to fit their teaching goals best.

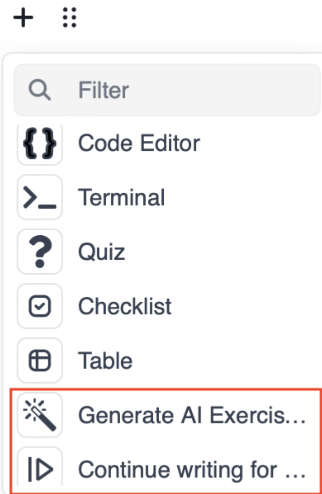


Figure 5.4: DOCABLE + menu for adding various components to the notebook, and AI content and exercise generation features marked in red.

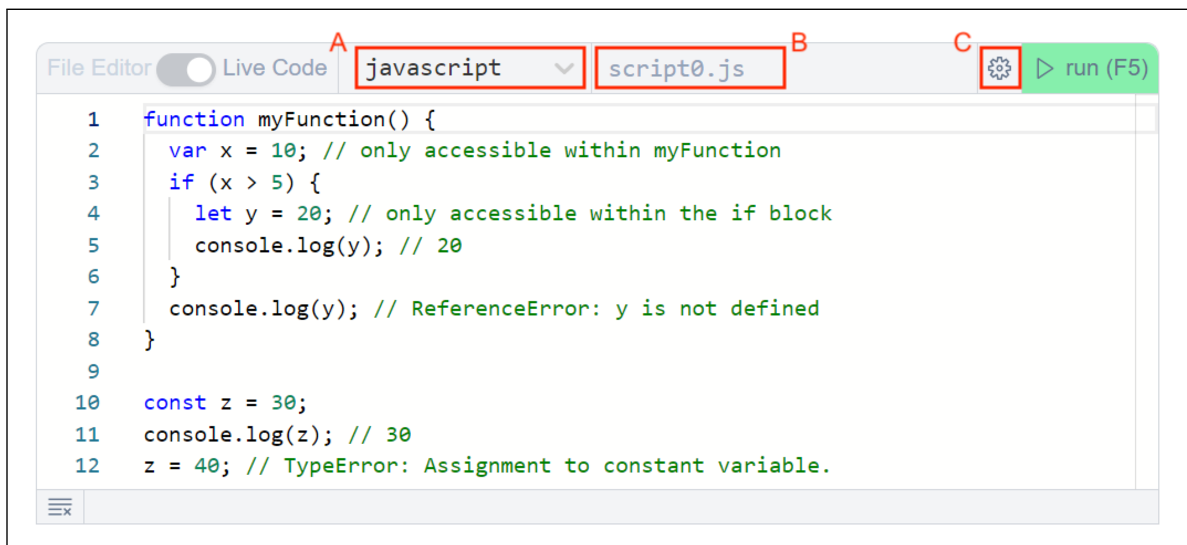


Figure 5.5: A code editor cell, from author's view, showing options for setting the language (A), script path on the file-system (B), and a gear icon to optionally customize the default execution behavior (C).

When the author is ready to share the content they created, DOCABLE provides a link to the author, which they can send to students. The link will load the notebook in a different

mode, only allowing editing of interactive components (such as code editors and terminals) but not the explanations. Each student who follows this link will see the notebook running with their own isolated computing environment.

5.3.3 Computing Environment

DOCABLE provides versatile hosted computing environments and customization capabilities. Specifically, each notebook is backed by a default Docker image with built-in essential tools and language support for common uses. However, if the author desires, they can customize this environment. DOCABLE gives authors two options to do so (figure 5.6): 1. to configure their own Docker image in the notebook settings, and 2. use DOCABLE snapshot feature to save the current state of their environment, to be used by the readers in the future. This results in a language/platform agnostic notebook system, which can practically have any language supported, tools installed, or files available. In comparison, one of the limitations of Jupyter Notebooks and Google Colaboratory is that they are constrained to Python programming language and are not as friendly to support other tools or computing environments.

From the readers' perspective, on DOCABLE each reader is given their isolated Docker container as a hosted environment based on the image or snapshot that the author configured. When using the notebook, if the readers make a change by mistake that they wish to revert, DOCABLE provides them an option to revert their environment to the default state instantly. On the other hand, on Jupyter Notebooks, the environments typically need to be self-hosted, requiring more investment in setting them up.

5.3.4 Execution Mechanism

DOCABLE has a minimalist approach to implementing execution mechanisms compared to other notebook systems like Jupyter. Instead of having a running an in-memory kernel to share one execution context between all the cell executions, DOCABLE provides separate execution context for each cell, and keeps the persistence in the environment instead of in memory. This approach is similar to how developers work on their local machines outside of a notebook systems, so it may feel like a more natural flow to the new users.

In comparison, Jupyter Notebooks' execution mechanism can have hidden states and 'magic commands' which confuse the students [44] and as a writer of "Data Science Com-

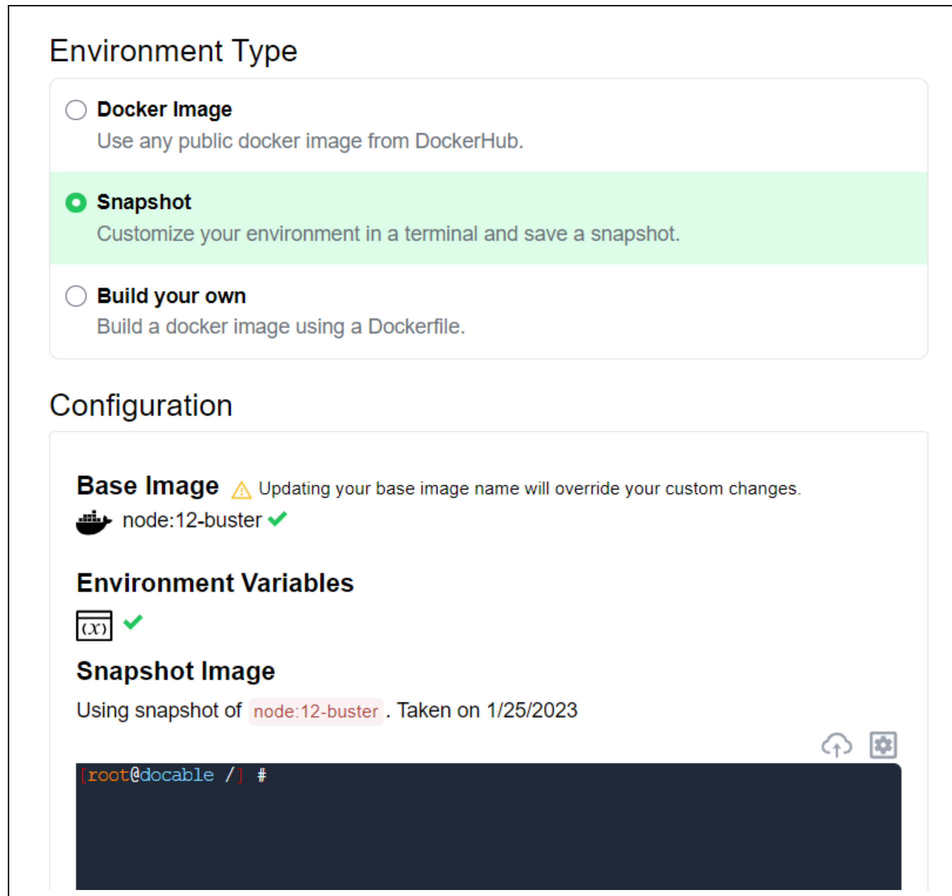


Figure 5.6: DOCABLE notebook's computing environment settings

munity Newsletter" tweeted to know if the notebook works, you should really "restart and run all or it didn't happen" [1].

5.4 Methodology

To evaluate the effectiveness of the current features in DOCABLE and identify the challenges that DOCABLE can address and those that remain unaddressed or require further improvements, we conducted a study involving nine instructors. We provided these instructors with access to DOCABLE for a semester. We allowed them to freely adopt DOCABLE in their teaching practices as they saw fit within their available free time constraints. A minimum re-

[1] https://twitter.com/digitalFlaneuse/status/996481061092806658

quirement was set for each class to incorporate at least one activity during the semester that took advantage of DOCABLE. After that, we conducted semi-structured instructor interviews and distributed surveys among their students.

This section explains the details of our qualitative research methods, which were employed to gather the following contributions:

- **Evaluation of an alternative computational notebook design in CS education.** This study aims to identify the strengths and limitations of DOCABLE as an alternative notebook system in the context of CS education.
- **Identification of essential properties of ClassOps tools, as perceived by CS instructors and students.** Tools may have different implementations of features to help instructors in course creation and delivery, but discovering the necessary properties can guide the design and development of future tools in this domain.

5.4.1 Participants

We conducted semi-structured interviews with nine computer science instructors and collected survey responses from 317 students. This section details the participants, their demographics, the type of courses involved, the interview, and the analysis process.

Demographics

We recruited nine computer science instructors (P1-P9) from four institutions in the USA, including large R1 doctoral research universities and smaller liberal arts and community colleges. Table 5.1 provides more specific details for each of the participating instructors and the course levels in which we distributed student surveys.

Protocol

We conducted two semi-structured interviews with each instructor to gain insights from the instructors' perspectives. The first interview took place at the beginning of the semester, and served as an onboarding process, allowing us to understand their expectations and needs concerning course material preparation. The second interview was conducted at the end of the semester to capture their experience, identify the most beneficial aspects of DOCABLE, and uncover any remaining desired features. The interview modality was virtual and about

Table 5.1: Participants (ID), their titles, institution size, courses they teach, years of teaching experience (Exp), and their DOCABLE adoption in this study.

ID	Title	Institution Size	Course	Exp.	DOCABLE adoption
P1	Assistant Professor	R1	SE	3	course workshops
P2	Full Professor	R1	CS2, Java	≥ 10	in-class exercises
P3	Lecturer	Liberal Art College	SE + CS1, Python	3	in-class exercises, workshops
P4	Lecturer	Community College	CS1, Python + Java	≥ 10	course notes, exercises
P5	Assistant Professor	R1	Special Topics	3	in-class exercises
P6	Associate Professor	R1	CS1, Python, C++	≥ 10	course lecture, exercises
P7	Lecturer	CC	Linux	≥ 10	in-class exercises, playgrounds
P8	Associate Professor	R1	Special Topics	8	course lectures, workshops
P9	Lecturer	R1	Special Topics	1	course lectures, workshops

30 minutes long for each session, during which we transcribed the conversations and took notes. During the interviews, we adhered to a rough script covering the topics below and iteratively improved our questions to be able to extract more meaningful feedback from the conversations:

- What improvements did you wish to make in your course delivery?
- To what extent did DOCABLE meet those improvement needs?
- What types of activities or course materials did you create using DOCABLE?
- What was your previous approach before adopting DOCABLE for this aspect of your course?
- Do you plan to continue using DOCABLE in the future?
- Have you explored or heard of any other relevant technologies?
- What are your thoughts on using AI for providing feedback to students and creating course material?

In order to understand students' perspectives, we designed a voluntary usability survey that included 12 multiple-choice questions in the Likert chart presented in Table 5.2. We also designed a feedback form in which we asked students to explain their experience with DOCABLE compared to the other assignments or courses that had not utilized DOCABLE.

We distributed the usability survey and feedback form among students at the end of the semester. Combining the two, this general goal was to gather feedback on their experience with DOCABLE and allow students to express their thoughts, suggestions, or complaints.

5.4.2 Analysis

We analyzed our instructor interview notes and the student responses to the feedback form through thematic analysis [9]. We reviewed the transcripts and survey responses, coded the relevant parts, and found themes in the coded feedback. Based on these themes we then formalized a set of desired properties. Additionally, we used the results from our Likert scale part of the student survey, which we use to confirm the findings about the effectiveness of DOCABLE.

5.4.3 Limitations

This study uses qualitative methods, which may introduce certain limitations to the findings. The findings from analyzed responses in interviews and surveys may include the researchers' interpretations. Additionally, the questions asked in semi-structured interviews or their order may slightly change for each participant to accommodate the flow of conversation with each participant. Finally, we consider the instructors' perception of students' learning experience to be a metric to identify any negative experiences for the students. By definition, perception refers to the instructor's observation without gathering quantitative metrics of students' work before and after DOCABLE that can be used for a statistically significant change.

5.5 Findings

In the analysis of our semi-structured interviews and survey responses, we discovered what worked and what did not work for the use case of instructors when they experimented with DOCABLE. To provide more context for the properties that we deem necessary for EdTechs in this space, we will first present an overview of the reactions to DOCABLE, then go into the details of how and why each property is essential.

5.5.1 How Did Instructors Use DOCABLE?

We allowed the instructors to use DOCABLE as they see fit for their course and as much as their time allows. The extent of DOCABLE usage varied among participants, with some incorporating it into their select in-class exercises and others utilizing it for all course workshops, lectures, or both. Table 5.1 shows an overview of adoption by each instructor.

5.5.2 Instructors' Perspective

From the perspective of instructors, DOCABLE affected four areas: 1. content creation, 2. content pedagogy, 3. course delivery, and 3. instructors' perception of its effects on students' learning experiences.

Authoring Experience

DOCABLE uses a visual editor, which most instructors found "*nice to be able to add whatever thing the text needs and save it. So it (was) easy to make content*" (P1). Moreover, in terms of liveness of the documents "*it was awesome that **I could create, run and test my tutorial on the same page**, and did not have to make separate project locally, extract parts of it to a markdown file and then wonder if it would still work when my students try to run it as a whole*" (P9). Another positive feedback is that "***it's instant (to update)***"; for example P7 "*got the first student (who found) something was wrong, ... I could go in and fix it, and then re-publish that*" instantly before any other student encountered it. "*It was a godsend for*" P6 who "*came out of it as being a huge fan, even was trying ... to get some other people using it*". Although some instructors pointed out the transition of bringing content to DOCABLE may add some "*overhead*", since they "*have (existing content) ..., it's not super difficult to do*" (P2).

Instructors that tried DOCABLE's AI content generator were "***excited to have more practice area for students** to work through code because generating the problems is a ton of work*" (P4). AI generation was also helpful to less experienced instructors, P9 shared "*it was a huge help for me to set up this course for the first time quickly*". However, as previous research [84] found, the AI-generated content may lack quality or accuracy; P9 "*often had to revise it to improve quality but nonetheless, it helped with brainstorming and getting started with a rough draft*".

Pedagogy Design

"*Making the same activities in a static environment would be a lot harder*", P1 explained, and when we asked them to compare their static content to DOCABLE content, they pointed "*(static) can't be as complex. most of the static ...is like homework questions or things that are useful for studying, but not actually practical for doing things*" and a hands-on learning experience. P2 expressed "*it would be really cool to have (students) write test cases to find bugs in code*" in an in-class activity and "*I don't know that I could do it (before)*". With DOCABLE it is possible "*to* ***do some things, that would be harder to do in Google forms***". Furthermore, in comparison, other tools such as Practice-It[2] and TYPOS [26], always hide a lot of infrastructure for running the code, while P2 expressed "*I feel like* DOCABLE *(can) reveal more*" if author desires. So those other tools are "*really intended a bit more for CS1 courses because they*" abstract away details and focus on method level, and in a "*CS2 course, focusing at the method level, isn't sufficient*".

Delivery of Course

P6 was interested in making their C/C++ course more interactive, but for "*C/C++ it's really hard to actually have a programming environment that's online*". DOCABLE "*met that need to show more examples (and) walk them through the material, I'd say in a simpler fashion, or, you know, more organized fashion*".

DOCABLE also helped instructors save time when delivering the course. For example, before DOCABLE they "*could never run workshops or have students work on exercises in the classroom... because nothing would work... (students) need to download and install this thing that takes 30 minutes, or 100 people had 100 different issues... and as a result, we never got to learn the thing we wanted to learn*" (P8). P7, who teaches a Linux system administration course, explained although "*virtualization was not a component of the course*" students had to set it up to have a Linux environment which added "*too much of a learning curve*". In comparison, "*it was quicker for (instructor) to go in* DOCABLE *and create the environment*" and tell the student "***here's a link, do it here***". This was "*a lot easier than trying to walk them through setting up*" virtualization.

[2] `https://practiceit.cs.washington.edu/`

Instructors' Perception of Learning Experience

DOCABLE "*is really good for delivering students a self-contained experience*" to introduce students to new material, "*give them stuff to read through, give them stuff to try, and give them stuff to write. It's a good experience for them*" and students "*were a lot more engaged now than what I've seen before*", P3 explained. Instructors recognized that students typically learn by doing and "*it is hard (for them) to pay attention when you are just talking for a long time*" (P3). Conceptually explaining topics on a whiteboard can help, but it may not always be enough. "*The whiteboard and PowerPoint don't have a compiler on it*" but "*the students ultimately really like being able to compile and execute their code*" (P2).

In some instances, instructors also noticed that students spend more time studying the content. As a result, instructors get "*less coding questions, because (*DOCABLE *format) is helping students*" (P4). Students "*who were just looking at notes, now watch (my embedded) YouTube videos if they have a question, versus just shooting off an email. Because that's generally what would happen (before)*" (P4) and instructors felt "DOCABLE *really created a long-term reference the students (which they) could always go back to*" (P6) if they need practice or have a question.

5.5.3 Students' Perspective

In addition to the instructor feedback, we also analyze the students' perspective to verify the instructor feedback and potentially gain new insights from the learners perspectives. The overall sentiment in the feedback form and the students' usability survey was positive. Three common student comments were related to DOCABLE making students more efficient in learning, gaining more hands-on experience, and enjoying a better learning experience.

Students' feedback showed DOCABLE was "***easy** to navigate*", and the structure was "*simple and easy to understand*". DOCABLE "*streamlined the learning process, so instead of having to switch between different tools and environments to experiment with code, (students)* ***can work directly within the course material****. This can save time and make the learning process more efficient*". One of the most common benefits mentioned was the "***immediate feedback** on whether (they are) completing the lesson correctly*", it helped them learn in a "*fast and in an efficient manner*" and "*boosted (their) confidence with hands-on practice*". In some cases, students even found learning on DOCABLE "*was fun*" and indicated they "*looked forward to doing the* DOCABLE *assignments*" because it was "*2x* ***more engaging***"

Table 5.2: Usability Survey Responses

	% Agree	Likert Responses[1] SD	D	N	A	SA	Distribution[2]
I felt more engaged in part of the course material that used DOCABLE.	87%	4	0	21	69	95	
I imagine most people would learn to use DOCABLE very quickly.	86%	2	10	32	131	141	
I like to use DOCABLE notebooks in future tutorials & exercises.	83%	6	7	41	115	147	
DOCABLE was easy to use.	82%	5	11	42	125	133	
I felt very confident using DOCABLE.	76%	6	20	51	122	117	
I found the various features in DOCABLE were well integrated.	75%	8	12	60	131	105	
I felt more confident about my learning in DOCABLE exercises.	71%	11	17	64	108	115	
I needed to learn lots of things before using DOCABLE features.	12%	125	106	46	24	15	
I need assistance to start using DOCABLE in future exercises.	12%	115	125	38	23	15	
I thought there was too much inconsistency in the DOCABLE.	11%	121	119	42	22	12	
I found DOCABLE notebooks very cumbersome/awkward to use.	9%	143	112	33	16	12	
DOCABLE was unnecessarily complex.	8%	115	146	31	13	11	

[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.

compared non-DOCABLE activities.

Compared to the workaround that instructors previously used (if any), DOCABLE was superior from the point of view of students; for example, students indicated "*other online platforms (that they experienced) are not as simple, some having difficult to navigate chapters, pages, and quizzes*". Students pointed they "*prefer* DOCABLE *because (they) get instant feedback ... instead of typing things into a google form that doesn't let you know if you did it correctly*". We believe one of the related factors is that DOCABLE promotes instructors to create a more customized experience for their students; students wrote to us that "*being*

told to read a chapter out of a book and write a summary on what I read is so boring and it doesn't feel like the teacher really cares whereas the videos on DOCABLE *are recorded by the teacher to give examples and provide code examples to use or manipulate to further understand the material*". This customized learning experience on DOCABLE can also be more concise and to the point because "*standard online textbooks do not have that level of control from the professor, leaving a lot of extra information that is unnecessary*".

Table 5.2 presents the details of student responses to the Likert scale usability survey, which confirms the findings of the general feedback described above.

5.5.4 Remaining Needs

Although the overall sentiment from both instructors and students was positive, they also pointed out four main needs that are still not addressed:

Cannot See Students' Progress

Several instructors mentioned they "*would really like a way of seeing how many students have done a problem, and where they stand*" (P3), or "*even just to see the logs of who has actually read*" their content (P4). This kind of visibility can help the instructor "*find students who really need me to come around and help them on stuff*" (P3). When a student asks for help on an assignment and only "*spent 11 min (studying whole) chapter*", knowing this information "*gives (instructor) some fire power*" (P4). This information can also assist instructors to "*keep track of lecture participation*" because manually "*keeping track of that information, in Google forms is a giant pain. And if there is an easier way to do it, that would be fantastic*" (P2).

Live Collaborative is Not Supported

Instructors pointed out that they desire more live collaboration capabilities on the notebooks they create. For example, "*features to help students collaborate on their project is useful for trying things*" together, do pair programming in a class activity, or help each other. Furthermore, literature has found such collaborative features, for example, from CODESTRATES[80] (a literate computing environment), can be helpful to students [8].

Notebook Format

DOCABLE, like other notebook systems, has a linear notebook structure that combines components such as code, description, videos, and more in an interactive way. One of the instructors expressed although this format works in many parts of the course, it may not seem like a good fit to all instructors and for all use cases. For example, in a larger project, "*what could it possibly look like for (students) to build and run a web-based application within it? You can do it, but it is not the natural fit*" (P6).

Authoring Interface Preference

Although most instructors liked the visual authoring interface of DOCABLE, some pointed "*possibility to edit Markdown directly would have been nice*" (P6). P3 described "*I just instinctively use back ticks to try and do pre-formatted text, underscores for italics etc. I do this in* DOCABLE *and it didn't work*" so "*(formatting) some of the activities was a large ... pain*".

5.6 Discussion: Essential Properties

Our findings analyzed and categorized the experience of instructors and students with DOCABLE into different themes. In this section we present implications of findings to form five essential ClassOps tool properties which we hope will guide tool smiths in the design of this space.

Interactivity

The liveness and interactivity of the platform is important to both the instructor workflow and the student learning experience:

One of the common things instructors mentioned about creating content was switching between multiple mediums, copying and pasting, or cherry-picking parts of a full implementation to the present in a separate document for the students. This workflow is inefficient and error-prone. With a live authoring interface, if the instructor adds a coding example to a document, they can execute this code immediately on the same page to see its execution in the same computing environment that students will use. This property was identified by instructors when P6 "*was able to create, 16 or so* DOCABLE *pages in relatively*

short time-frame because (there were) basically writing those as going along" and content creation "*that quickly was extremely beneficial*" to this course.

Similarly, liveness is essential for a "*learning while doing experience*" (P-student). Student without a live document, can only try a code example by copying and setting it up in their integrated development environments (IDE) to experiment with. In contrast, on a live document, students can execute and instantly "*see the results of their code changes in real-time, making the learning experience more interactive and engaging*" (P-student).

Finally, we argue that the concept of liveness can be described through various levels, similar to the metaphorical sense of automated vehicles [66]. For example, the next level of liveness would include live collaboration features, which we discussed in section 5.5.4, allowing students to do live pair programming on the same document.

Embeddability

Instructors' teaching methods and modalities often require a blend [81] of different components to construct a learning material. The requirements vary for each instructor, but some examples of components that instructors in our study utilized in their course are slides, lecture notes, videos, code examples, quizzes, and more. For a ClassOps that aims to assist instructors in course delivery, an essential property is the ability to "*integrate everything into one piece*" (P3) in a seamless way. Better integration will eliminate the need for "*switch between different tools and environments, (and) students can work directly within the course material*". For example, in a C/C++ debugging lecture, the instructor may need to present students with a demo video of an analysis tool, Valgrind[3], before having an in-class activity. Having the description, demo video, code example, and a live terminal window on the same document makes teaching more efficient for instructors and learning more focused on the course's primary objective.

Have a Real Environment

To execute code and support more instructors with their needs, a more holistic view and recognizing the need for a computing environment is crucial. Other studies found the importance of the computing environment and how a simulated environment limited the instructors [58]. Specifically, among the benefits that our participants mentioned, "*real*" and "*hosted*" computing environment in DOCABLE was a common benefit.

[3]`https://valgrind.org`

For a tool that aims to support course material and delivery, providing a real, customizable, and hosted computing environment is essential. More specifically, "real" means the environment is not simulated with limited functionalities. "customizable" means the platform allows instructors to install additional software or data sets and save them for the students to create a consistent experience. And "hosted" means that instructors or students will not need to set up any additional software or tools on their computers to access or use this environment.

Sharability

Instructors often need to link to the material during lectures, in announcement emails, or make a post on the learning management system (LMS), so the notebook system must make it easy to publish the material. Shareable links, combined with a hosted computing environment (section 5.6), makes a seamless experience for the student, allowing them to follow a link in their web browser to arrive at a page with all the interactive learning material they need in a ready-to-use environment without installing additional software. This user experience aligns with the "walk-up-and-use" design guideline from Guo's scalable academia software guidelines [31].

Visibility and Verifiability

As discussed in section 5.5.4, one of the instructors' essential needs are understanding the students' progress and time spent studying the material they created. We can argue that instructors need *visibility* to identify whether students are studying the material and *verifiability* to understand how well they perform in the activities. This property can give instructors feedback on how well the class is engaged with the content and give them insights on what they can focus on to assist students that need it the most. An idea for providing more visibility that received support from our participants is to inspire the design of a monitoring dashboard similar to dashboards used to monitor infrastructure in DevOps. Such a dashboard can include metrics such as user logs, time spent by each student, and status of their exercise executions and submissions.

Similarly, verifiability is essential to the students' learning experience. In the feedback we received from students, they expressed that one of the main reasons they preferred DOCABLE activities over other methods was that it made it "*easier to follow and implement the exercises. And the feedback given while implementation and writing tests is also neat and*

efficient" P-student and can give students more confidence. Tools can implement different different types of verifiability, for example, using tests, input/output checks, static analysis, or use artificial intelligence (AI) and large language model (LLM) capabilities which have been a recent hot topic [7, 79, 14].

5.7 Conclusion

Course material preparation and delivery pose significant challenges for computer science instructors, exacerbating numerous other teaching challenges. In this study, we investigated the effectiveness of computational notebooks in addressing those challenges, with a particular focus on DOCABLE, a design with features specifically tailored for computer science education. Our goal was to assess how various features can support instructors in creating and delivering their courses while enhancing the student learning experience. To achieve this, we conducted semi-structured interviews with instructors and distributed surveys among students to gather their perspectives. The insights gained from these interviews and surveys allowed us to identify several benefits and shortcomings of this notebook system. Building upon these findings, we formalized a set of essential properties that can guide the design of future ClassOps tools in this domain for maximizing effectiveness in addressing the needs.

CHAPTER 6

CONCLUSION

The thesis of this dissertation is the following statement:

> ClassOps can improve educational material preparation and delivery by simultaneously addressing several needs, such as computing environment, automation, and authoring interface.

In this dissertation, I defended the thesis statement in four studies (chapters 2, 3, 4, 5). In the first study (chapter 2), I focused on online tutorials as one of the most widely available types of educational material with a very large readership. I conducted an empirical study to understand online tutorials' current state better and measure their quality through the lens of executability. This study used two strategies for measuring executability, *naive* inspired by other studies in this domain [39, 41], and *human-annotation-based*. Results showed that both strategies concluded with meager executability rates of 26% and 52%, respectively. I analyzed these execution logs and identified several common barriers ranging from less harmful causes, such as interactive prompts requiring human responses, to more serious errors, such as missing, outdated, or inconsistent instructions. Finally, I validated these

findings in discussion with major stakeholders in the technical documentation to guide tutorial authors and toolsmiths in improving the quality of tutorials.

In the second study (chapter 3), to better understand needs and design future studies, I found it essential to dive deeper and study the creators of educational material, i.e., instructors. I interviewed 32 CS instructors from 26 institutions and asked about their pain points, challenges, and the associated workarounds they have attempted to use. After a thematic analysis of the instructors' experiences, I identified several pain points, including challenges regarding getting visibility into students' struggles, answering their questions, and course material preparation. Additionally, the instructor feedback showed that workarounds often did not completely resolve the pain points and even brought other new challenges. Finally, I summarized the primary desires of instructors as more visibility into the student struggles, better support for automating repetitive tasks, improving interactive environments, and more streamlined course delivery.

Following the second study, I conducted my two studies to focus on possible ways to address challenges discovered previously. Chapter 4 covers a more in-depth study of instructor pain points regarding helping students with their computing environment issues. Using the preliminary findings, I listed the common root causes for issues students experience in their computing environments. Then I evaluated OPUNIT, a tool for environment verification inspired by mentioned root causes. Students in a survey indicated OPUNIT was effective in helping them discover and fix the issues and improve confidence.

Finally, in chapter 5, I studied how we can help instructors prepare and deliver the course material. With the preliminary findings described in chapter 3, I conducted a study on a type of computational notebook, DOCABLE, as a potential solution that may address the needs of computer instructors in this area. This study's primary goal was to discover further details of instructor desires related to course material preparation and delivery. In this study, I conducted instructor interviews with 9 participants and collected survey responses from more than 300 students. Then through a thematic analysis process, I formalized the findings as a set of essential properties for ClassOps tools in this area, which can guide the toolsmiths for more effective implementations in the future.

REFERENCES

[1] B. Adams, S. Bellomo, C. Bird, T. Marshall-Keim, F. Khomh, and K. Moir. The practice and future of release engineering: A roundtable with three release engineers. *IEEE Software*, 32(2):42–49, Mar 2015. ISSN 0740-7459. doi: 10.1109/MS.2015.52.

[2] A. Basiri, N. Jones, A. Blohowiak, L. Hochstein, and C. Rosenthal. *Chaos Engineering*. O'Reilly Media, Inc., 2019.

[3] L. Bass, I. Weber, and L. Zhu. *DevOps: A software architect's perspective*. ADDISON-WESLEY, 2015.

[4] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell, and C. Cook. Tinkering and gender in end-user programmers' debugging. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, page 231–240, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933727. doi: 10.1145/1124772.1124808. URL `https://doi.org/10.1145/1124772.1124808`.

[5] A. Begel and T. Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 12–23, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568233. URL `https://doi-org.prox.lib.ncsu.edu/10.1145/2568225.2568233`.

[6] M. Birks, Y. Chapman, and K. Francis. Memoing in qualitative research: Probing data and processes. *Journal of Research in Nursing*, 13(1):68–75, jan 2008. doi: 10.1177/1744987107081254.

[7] S. Bordt and U. von Luxburg. Chatgpt participates in a computer science exam. *arXiv preprint arXiv:2303.09461*, 2023.

[8] M. Borowski, J. Zagermann, C. N. Klokmose, H. Reiterer, and R. Rädle. Exploring the benefits and barriers of using computational notebooks for collaborative programming assignments. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 468–474, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367936. doi: 10.1145/3328778.3366887. URL `https://doi.org/10.1145/3328778.3366887`.

[9] V. Braun and V. Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. doi: 10.1191/1478088706qp063oa. URL `https://www.tandfonline.com/doi/abs/10.1191/1478088706qp063oa`.

[10] V. Braun and V. Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

[11] J. Brill and Y. Park. Evaluating online tutorials for university faculty, staff, and students: The contribution of just-in-time online resources to learning and performance. *International Journal on E-Learning*, 10(1):5–26, January 2011. ISSN 1537-2456. URL https://www.learntechlib.org/p/33278.

[12] A. Brooks, C. Hardin, J. Scianna, M. Berland, and L. H. Legault. Approaches to transitioning computer science classes from offline to online. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, page 81–87. ACM, 2021. ISBN 9781450382144. doi: 10.1145/3430665.3456366. URL https://doi-org.prox.lib.ncsu.edu/10.1145/3430665.3456366.

[13] M. Chassignol, A. Khoroshavin, A. Klimova, and A. Bilyatdinova. Artificial intelligence trends in education: a narrative overview. *Procedia Computer Science*, 136:16–24, 2018. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2018.08.233. URL https://www.sciencedirect.com/science/article/pii/S1877050918315382. 7th International Young Scientists Conference on Computational Science, YSC2018, 02-06 July2018, Heraklion, Greece.

[14] E. Chen, R. Huang, H.-S. Chen, Y.-H. Tseng, and L.-Y. Li. Gptutor: a chatgpt-powered programming tool for code explanation. *arXiv preprint arXiv:2305.01863*, 2023.

[15] A. Cook, A. Zaman, E. Hicks, K. Malasri, and V. Phan. Try that again! how a second attempt on in-class coding problems benefits students in cs1. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, page 509–515. ACM, 2022. ISBN 9781450390705. doi: 10.1145/3478431.3499362. URL https://doi.org/10.1145/3478431.3499362.

[16] R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 214–225, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939951. doi: 10.1145/1453101.1453130. URL https://doi.org/10.1145/1453101.1453130.

[17] C. R. A. (CRA). The phenomenal growth of cs majors since 2006, May 2017. URL https://cra.org/data/generation-cs/phenomenal-growth-cs-majors-since-2006/.

[18] N. Diana, M. Eagle, J. Stamper, S. Grover, M. Bienkowski, and S. Basu. An instructor dashboard for real-time analytics in interactive programming assignments. In *Proceedings of the Seventh International Learning Analytics & Knowledge Conference*, LAK

'17, page 272–279. ACM, 2017. ISBN 9781450348706. doi: 10.1145/3027385.3027441. URL https://doi.org/10.1145/3027385.3027441.

[19] R. DiYanni and A. Borst. *Chapter 3: Active Learning*, page 42–60. Princeton University Press, 2020.

[20] I. Drosos, P. J. Guo, and C. Parnin. Happyface: Identifying and predicting frustrating obstacles for learning programming at scale. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 171–179, Oct 2017. doi: 10.1109/VLHCC.2017.8103465.

[21] S. H. Edwards. Improving student performance by evaluating how well students test their own programs. *J. Educ. Resour. Comput.*, 3(3):1–es, sep 2003. ISSN 1531-4278. doi: 10.1145/1029994.1029995. URL https://doi.org/10.1145/1029994.1029995.

[22] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, and J. Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Australasian Computing Education Conference*, ACE '22, page 10–19, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450396431. doi: 10.1145/3511861.3511863. URL https://doi.org/10.1145/3511861.3511863.

[23] D. Ford and C. Parnin. Exploring causes of frustration for software developers. In *Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '15, page 115–116. IEEE Press, 2015.

[24] A. B. Frymier and G. M. Shulman. "what's in it for me?": Increasing content relevance to enhance students' motivation. *Communication Education*, 44(1):40–50, 1995.

[25] J. Garner, P. Denny, and A. Luxton-Reilly. Mastery learning in computer science education. In *Proceedings of the Twenty-First Australasian Computing Education Conference*, ACE '19, page 37–46. ACM, 2019. ISBN 9781450366229. doi: 10.1145/3286960.3286965. URL https://doi.org/10.1145/3286960.3286965.

[26] A. M. Gaweda, C. F. Lynch, N. Seamon, G. Silva de Oliveira, and A. Deliwa. Typing exercises as interactive worked examples for deliberate practice in cs courses. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*, ACE'20, page 105–113, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376860. doi: 10.1145/3373165.3373177. URL https://doi.org/10.1145/3373165.3373177.

[27] D. Gillet, A. Vozniuk, M. J. Rodriguez Triana, and A. C. Holzer. Agile, versatile, and comprehensive social media platform for creating, sharing, exploiting, and archiving personal learning spaces, artifacts, and traces. In *The World Engineering Education Forum*, number CONF, 2016.

[28] N. Gitinabard, S. Heckman, T. Barnes, and C. Lynch. Designing a dashboard for student teamwork analysis. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, page 446–452, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450390705. doi: 10.1145/3478431.3499377. URL https://doi.org/10.1145/3478431.3499377.

[29] GitLab. Postmortem of database outage of january 31. https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/, 2017.

[30] A. K. Goel and D. A. Joyner. Using ai to teach ai: Lessons from an online ai class. *AI Magazine*, 38(2):48–59, Jul. 2017. doi: 10.1609/aimag.v38i2.2732. URL https://ojs.aaai.org/index.php/aimagazine/article/view/2732.

[31] P. Guo. Ten million users and ten years later: Python tutor's design guidelines for building scalable and sustainable research software in academia. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, page 1235–1251, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386357. doi: 10.1145/3472749.3474819. URL https://doi.org/10.1145/3472749.3474819.

[32] P. J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 579–584, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318686. doi: 10.1145/2445196.2445368. URL https://doi.org/10.1145/2445196.2445368.

[33] Q. Hao, B. Barnes, E. Wright, and E. Kim. Effects of active learning environments and instructional methods in computer science education. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, page 934–939, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450351034. doi: 10.1145/3159450.3159451. URL https://doi.org/10.1145/3159450.3159451.

[34] K. hara, D. Blank, and J. Marshall. Computational notebooks for ai education. 05 2015. doi: 10.13140/2.1.2434.5928.

[35] H. Hata, C. Treude, R. G. Kula, and T. Ishio. 9.6 million links in source code comments: Purpose, evolution, and decay. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1211–1221, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICSE.2019.00123. URL https://doi.org/10.1109/ICSE.2019.00123.

[36] A. Head, J. Jiang, J. Smith, M. A. Hearst, and B. Hartmann. Composing flexibly-organized step-by-step tutorials from linked source code, snippets, and outputs. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, New York, NY, USA, 2020. Association for Computing Machinery.

ISBN 9781450367080. doi: 3313831.3376798. URL https://doi.org/10.1145/3313831.3376798.

[37] S. Heckman, K. T. Stolee, and C. Parnin. 10+ years of teaching software engineering with itrust: The good, the bad, and the ugly. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '18, pages 1–4, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5660-2. doi: 10.1145/3183377.3183393. URL http://doi.acm.org/10.1145/3183377.3183393.

[38] S. Hooshangi, M. Ellis, and S. H. Edwards. Factors influencing student performance and persistence in cs2. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, page 286–292, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450390705. doi: 10.1145/3478431.3499272. URL https://doi.org/10.1145/3478431.3499272.

[39] E. Horton and C. Parnin. Gistable: Evaluating the executability of python code snippets on github. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 217–227. IEEE, Sep. 2018. doi: 10.1109/ICSME.2018.00031.

[40] E. Horton and C. Parnin. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 328–338. IEEE Press, 2019. doi: 10.1109/ICSE.2019.00047. URL https://doi.org/10.1109/ICSE.2019.00047.

[41] M. M. Hossain, N. Mahmoudi, C. Lin, H. Khazaei, and A. Hindle. Executability of python snippets in stack overflow. *arXiv preprint arXiv:1907.04908*, 2019.

[42] S. Hugtenburg and A. Zaidman. First impressions of using stack overflow for education in a computer science bachelor programme. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2*, SIGCSE 2022, page 1146, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450390712. doi: 10.1145/3478432.3499046. URL https://doi.org/10.1145/3478432.3499046.

[43] G. D. Israel. *Sampling the evidence of extension program impact.* Citeseer, 1992.

[44] J. W. Johnson. Benefits and pitfalls of jupyter notebooks in the classroom. In *Proceedings of the 21st Annual Conference on Information Technology Education*, SIGITE '20, page 32–37, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370455. doi: 10.1145/3368308.3415397. URL https://doi.org/10.1145/3368308.3415397.

[45] L. C. Kats, R. G. Vogelij, K. T. Kalleberg, and E. Visser. Software development environments on the web: A research agenda. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming*

and Software, Onward! 2012, page 99–116. ACM, 2012. ISBN 9781450315623. doi: 10.1145/2384592.2384603. URL `https://doi.org/10.1145/2384592.2384603`.

[46] A. S. Kim and A. J. Ko. A pedagogical analysis of online coding tutorials. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 321–326, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4698-6. doi: 10.1145/3017680.3017728. URL `http://doi.acm.org/10.1145/3017680.3017728`.

[47] K. Krishna, A. Roy, and M. Iyyer. Hurdles to progress in long-form question answering. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4940–4957, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.393. URL `https://aclanthology.org/2021.naacl-main.393`.

[48] S. Kross and P. J. Guo. End-user programmers repurposing end-user programming tools to foster diversity in adult end-user programming education. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, VL/HCC '19, Oct 2019.

[49] B. Lafreniere, T. Grossman, and G. Fitzmaurice. Community enhanced tutorials: Improving tutorials with multiple demonstrations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 1779–1788, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1899-0. doi: 10.1145/2470654.2466235. URL `http://doi.acm.org/10.1145/2470654.2466235`.

[50] S. Lau, I. Drosos, J. M. Markel, and P. J. Guo. The design space of computational notebooks: An analysis of 60 systems in academia and industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–11, Aug 2020. doi: 10.1109/VL/HCC50065.2020.9127201.

[51] S. Lau, J. Eldridge, S. Ellis, A. Fraenkel, M. Langlois, S. Rampure, J. Tiefenbruck, and P. J. Guo. The challenges of evolving technical courses at scale: Four case studies of updating large data science courses. In *Proceedings of the Ninth ACM Conference on Learning @ Scale*, L@S '22, page 201–211, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391580. doi: 10.1145/3491140.3528278. URL `https://doi.org/10.1145/3491140.3528278`.

[52] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: the state of the practice. *IEEE Software*, 20(6):35–39, Nov 2003. ISSN 1937-4194. doi: 10.1109/MS.2003.1241364.

[53] Y. S. Lincoln and E. G. Guba. *Naturalistic Inquiry*. Sage Publications, Newbury Park, CA, 1985.

[54] N. McDonald, S. Schoenebeck, and A. Forte. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice. *Proc. ACM Hum.-Comput. Interact.*, 3(CSCW), Nov. 2019. doi: 10.1145/3359174. URL https://doi.org/10.1145/3359174.

[55] S. Mirhosseini and C. Parnin. Opunit: Sanity checks for computing environments. In *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment: Second International Workshop, DEVOPS 2019, Château de Villebrumier, France, May 6–8, 2019, Revised Selected Papers*, page 167–180, Berlin, Heidelberg, 2019. Springer-Verlag. ISBN 978-3-030-39305-2. doi: 10.1007/978-3-030-39306-9_12. URL https://doi.org/10.1007/978-3-030-39306-9_12.

[56] S. Mirhosseini and C. Parnin. Opunit: Sanity checks for computing environments. In J.-M. Bruel, M. Mazzara, and B. Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 167–180, Cham, 2020. Springer International Publishing. ISBN 978-3-030-39306-9.

[57] S. Mirhosseini and C. Parnin. Docable: Evaluating the executability of software tutorials. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 375–385, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409706. URL https://doi.org/10.1145/3368089.3409706.

[58] S. Mirhosseini, A. Z. Henley, and C. Parnin. What is your biggest pain point? an investigation of cs instructor obstacles, workarounds, and desires. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 291–297, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394314. doi: 10.1145/3545945.3569816. URL https://doi.org/10.1145/3545945.3569816.

[59] D. Mirza, P. T. Conrad, C. Lloyd, Z. Matni, and A. Gatin. Undergraduate teaching assistants in computer science: A systematic literature review. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, page 31–40. ACM, 2019. ISBN 9781450361859. doi: 10.1145/3291279.3339422. URL https://doi.org/10.1145/3291279.3339422.

[60] A. Mysore and P. J. Guo. Torta: Generating mixed-media gui and command-line app tutorials using operating-system-wide activity tracing. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 703–714, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4981-9. doi: 10.1145/3126594.3126628. URL http://doi.acm.org/10.1145/3126594.3126628.

[61] A. Mysore and P. J. Guo. Porta: Profiling software tutorials using operating-system-wide activity tracing. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, page 201–212, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359481. doi: 10.1145/3242587.3242633. URL https://doi.org/10.1145/3242587.3242633.

[62] A. Mysore and P. J. Guo. Porta: Profiling software tutorials using operating-system-wide activity tracing. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 201–212, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5948-1. doi: 10.1145/3242587.3242633. URL http://doi.acm.org/10.1145/3242587.3242633.

[63] M. Nagappan, T. Zimmermann, and C. Bird. Diversity in software engineering research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 466–476, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322379. doi: 10.1145/2491411.2491415. URL https://doi.org/10.1145/2491411.2491415.

[64] N. Najjar, A. Stubler, H. Ramaprasad, H. Lipford, and D. Wilson. Evaluating students' perceptions of online learning with 2-d virtual spaces. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, page 112–118, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450390705. doi: 10.1145/3478431.3499396. URL https://doi.org/10.1145/3478431.3499396.

[65] M. J. Nathan, K. R. Koedinger, and M. W. Alibali. Expert blind spot : When content knowledge eclipses pedagogical content knowledge. In *Proceedings of the third international conference on cognitive science*, volume 644648, 2001.

[66] NHTSA, 2022. URL https://www.nhtsa.gov/sites/nhtsa.gov/files/2022-05/Level-of-Automation-052522-tag.pdf.

[67] B. D. Nye, A. C. Graesser, and X. Hu. Autotutor and family: A review of 17 years of natural language tutoring. *International Journal of Artificial Intelligence in Education*, 24(4):427–469, 2014.

[68] J. D. Ortega-Alvarez, C. Vieira, N. Guarín-Zapata, and J. Gómez. Flipping a computational modeling class: Strategies to engage students and foster active learning. In *2020 IEEE Frontiers in Education Conference (FIE)*, pages 1–4, 2020. doi: 10.1109/FIE44824.2020.9273890.

[69] C. Parnin. DevOps 519, 2019. https://github.com/CSC-DevOps/Course/#devops-csc-519.

[70] C. Parnin, C. Treude, and M. A. Storey. Blogging developer knowledge: Motivations, challenges, and future directions. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 211–214, May 2013. doi: 10.1109/ICPC.2013.6613850.

[71] C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, M. Stumm, S. Whitaker, and L. Williams. The top 10 adages in continuous deployment. *IEEE Software*, 34(3):86–95, May 2017. ISSN 1937-4194. doi: 10.1109/MS.2017.86.

[72] J. M. Perkel. Why jupyter is data scientists' computational notebook of choice. *Nature*, 563(7732):145–147, 2018.

[73] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. A large-scale study about quality and reproducibility of jupyter notebooks. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, 2019.

[74] J. Ponterotto. Brief note on the origins, evolution, and meaning of the qualitative research concept thick description. *The Qualitative Report*, 11(3), 2006.

[75] L. Porter, C. Bailey Lee, and B. Simon. Halving fail rates using peer instruction: A study of four computer science courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 177–182, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318686. doi: 10.1145/2445196.2445250. URL `https://doi.org/10.1145/2445196.2445250`.

[76] K. Presler-Marshall, S. Heckman, and K. T. Stolee. Identifying struggling teams in software engineering courses through weekly surveys. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, page 126–132, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450390705. doi: 10.1145/3478431.3499367. URL `https://doi.org/10.1145/3478431.3499367`.

[77] D. Procida. What nobody tells you about documentation, May 2017. URL `https://www.divio.com/blog/documentation/`.

[78] Puppet. 2018 state of devops report. https://puppet.com/resources/whitepaper/state-of-devops-report/, 2018.

[79] B. Qureshi. Exploring the use of chatgpt as a tool for learning and assessment in undergraduate computer science curriculum: Opportunities and challenges. *arXiv preprint arXiv:2304.11214*, 2023.

[80] R. Rädle, M. Nouwens, K. Antonsen, J. R. Eagan, and C. N. Klokmose. Codestrates: Literate computing with webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, page 715–725, New York,

NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349819. doi: 10.1145/3126594.3126642. URL https://doi.org/10.1145/3126594.3126642.

[81] M. Rodríguez-Triana, L. Prieto, A. Vozniuk, M. Shirvani Boroujeni, B. Schwendimann, A. Holzer, and D. Gillet. Monitoring, awareness and reflection in blended technology enhanced learning: a systematic review. *International Journal of Technology Enhanced Learning*, 9, 02 2016. doi: 10.1504/IJTEL.2017.10005147.

[82] C. P. Samim Mirhosseini. Opunit Survey, 2019. https://forms.gle/uhBYmtftdsfj5TxP8.

[83] J. L. Santos, S. Govaerts, K. Verbert, and E. Duval. Goal-oriented visualizations of activity tracking: A case study with engineering students. In *Proceedings of the 2nd International Conference on Learning Analytics and Knowledge*, LAK '12, page 143–152, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311113. doi: 10.1145/2330601.2330639. URL https://doi-org.prox.lib.ncsu.edu/10.1145/2330601.2330639.

[84] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*, ICER '22, page 27–43, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391948. doi: 10.1145/3501385.3543957. URL https://doi.org/10.1145/3501385.3543957.

[85] J. W. Schofield, D. Evans-Rhodes, and B. R. Huber. Artificial intelligence in the classroom: The impact of a computer-based tutor on teachers and students. *Social Science Computer Review*, 8(1):24–41, 1990. doi: 10.1177/089443939000800104. URL https://doi.org/10.1177/089443939000800104.

[86] C. Sharp, J. van Assema, B. Yu, K. Zidane, and D. J. Malan. An open-source, api-based framework for assessing the correctness of code in cs50. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, page 487–492, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368742. doi: 10.1145/3341525.3387417. URL https://doi.org/10.1145/3341525.3387417.

[87] N. Shrestha, C. Botta, T. Barik, and C. Parnin. Here we go again: Why is it difficult for developers to learn another programming language? In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 691–701, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380352. URL https://doi-org.prox.lib.ncsu.edu/10.1145/3377811.3380352.

[88] R. J. Sidelinger. College student involvement: An examination of student characteristics and perceived instructor communication behaviors in the classroom. *Communication Studies*, 61(1):87–103, 2010.

[89] A. A. Smith. Teaching computer science to biologists and chemists, using jupyter notebooks: Tutorial presentation. *J. Comput. Sci. Coll.*, 32(1):126–128, oct 2016. ISSN 1937-4771.

[90] D. Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[91] C. Treude and M. Aniche. Where does google find api documentation? In *Proceedings of the 2nd International Workshop on API Usage and Evolution*, pages 19–22. ACM, 2018.

[92] K. Verbert, S. Govaerts, E. Duval, J. L. Santos, F. Assche, G. Parra, and J. Klerkx. Learning dashboards: An overview and future research opportunities. *Personal Ubiquitous Comput.*, 18(6):1499–1514, aug 2014. ISSN 1617-4909. doi: 10.1007/s00779-013-0751-2. URL `https://doi.org/10.1007/s00779-013-0751-2`.

[93] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[94] H. Virdó and B. Hogan. Technical writing guidelines, Feb 2020. URL `https://www.digitalocean.com/community/tutorials/digitalocean-s-technical-writing-guidelines`.

[95] Y. Wu, S. Wang, C.-P. Bezemer, and K. Inoue. How do developers utilize source code from stack overflow? *Empirical Software Engineering*, pages 1–37, 2018.

[96] A. Yadav, S. Gretter, and S. Hambrusch. Challenges of a computer science classroom: Initial perspectives from teachers. In *Proceedings of the Workshop in Primary and Secondary Computing Education*, WiPSCE '15, page 136–137, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450337533. doi: 10.1145/2818314.2818322. URL `https://doi.org/10.1145/2818314.2818322`.

[97] A. Yadav, S. Gretter, S. Hambrusch, and P. Sands. Expanding computer science education in schools: understanding teacher experiences and challenges. *Computer Science Education*, 26(4):235–254, 2016.

[98] D. Yang, A. Hussain, and C. V. Lopes. From query to usable code: An analysis of stack overflow code snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 391–402, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4186-8. doi: 10.1145/2901739.2901767. URL `http://doi.acm.org/10.1145/2901739.2901767`.

[99] I. YeckehZaare, G. Grot, I. Dimovski, K. Pollock, and E. Fox. Another victim of covid-19: Computer science education. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, page 913–919, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450390705. doi: 10.1145/3478431.3499313. URL https://doi.org/10.1145/3478431.3499313.

[100] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 740–751, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. doi: 10.1145/3106237.3106262. URL https://doi.org/10.1145/3106237.3106262.

[101] S. Zhang, R. Shuttleworth, D. Austin, Y. Hicke, L. Tang, S. Karnik, D. Granberry, and I. Drori. A dataset and benchmark for automatically answering and generating machine learning final exams, 2022. URL https://arxiv.org/abs/2206.05442.