Docable: Evaluating the Executability of Software Tutorials

Samim Mirhosseini
NC State University
Raleigh, Texas, North Carolina
smirhos@ncsu.edu

Chris Parnin
NC State University
Raleigh, North Carolina, USA
cjparnin@ncsu.edu

ABSTRACT

The typical software tutorial includes step-by-step instructions for installing developer tools, editing files and code, and running commands. When these software tutorials are not executable, either due to missing instructions, ambiguous steps, or simply broken commands, their value is diminished. Non-executable tutorials impact developers in several ways, including frustrating learning experiences, and limiting usability of developer tools.

To understand to what extent software tutorials are executable—and why they may fail—we conduct an empirical study on over 600 tutorials, including nearly 15,000 code blocks. We find a naive execution strategy achieves an overall executability rate of only 26%. Even a human-annotation-based execution strategy—while doubling executability—still yields no tutorial that can successfully execute all steps. We identify several common executability barriers, ranging from potentially innocuous causes, such as interactive prompts requiring human responses, to insidious errors, such as missing steps and inaccessible resources. We validate our findings with major stakeholders in technical documentation and discuss possible strategies for improving software tutorials, such as providing accessible alternatives for tutorial takers, and investing in automated tutorial testing to ensure continuous quality of software tutorials.

KEYWORDS

software tutorials, documentation, testing, continuous integration

1 INTRODUCTION

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 1. Tutorials are featured prominently in developer-related search results [33], and serve a variety of purposes, such as integrating with instructional materials for classrooms [10], supporting documentation efforts [27], and training underrepresented and professional developers through community-driven [16] or paid workshops [2].

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

The ideal tutorial, as described in DigitalOcean's guidelines for tutorial creators [34], 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 [30], preventing instructions from becoming stale as tools or environments quickly evolve [18], and overcoming the *expert blind spot effect* [25], when tutorial creators do not anticipate steps where novice tutorial takers

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

[2] https://learnk8s.io/

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

may have difficulty [23]. As a result, undiscovered issues [17] in software tutorials can lead to frustrating and ineffective learning experiences [15, 22].

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 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:

2.1 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?

2.2 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 [26]. We used a web-scraping 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 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 [15], 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 [14] by considering each source as a strata. We used a relaxed confidence interval (80% ± 10%) to calculate the sample size of each strata, allowing us to target diversity over representativeness [24]. This yielded a total of 40 tutorials containing 787 content blocks—20 tutorials from Vultr, 14 tutorials from DigitalOcean, and 6 tutorials from Linuxize.

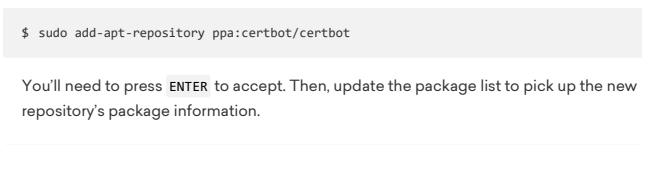

Figure 2: Tutorial fragment with code block. We extract the command 'sudo add-apt-repository ppa:certbot/certbot' for running in our execution harness.

2.3 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.

[3] https://www.vultr.com/docs/category/ubuntu/
[4] https://www.digitalocean.com/community/tutorials
[5] https://linuxize.com/tags/ubuntu/
[6] https://www.digitalocean.com/community/tutorials/how-to-set-up-continuous-integration-pipelines-with-drone-on-ubuntu-16-04

```
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 3: An example `steps.yml` file

Commands are instrumented to log output, failures, and exit status of the operation. A new execution harness is created for every tutorial.

2.4 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 [36] or gists [11] when measuring executability rates.

We designed a custom CSS selector to extract code blocks from each source of tutorials (see Figure 2). 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 5.

To identify why execution fails, we first cluster the commands by exit code after execution, as done by Horton and Parnin [11]. 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* [32] to organize code blocks into descriptive categories. The categories provide insight into various possible interpretations under which a code block could be executed.

2.5 Human-Annotation-Based Execution

To answer RQ2, we implement a technique, we call Docable. The name is derived from a portmanteau of the words documentation and runnable.

2.5.1 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

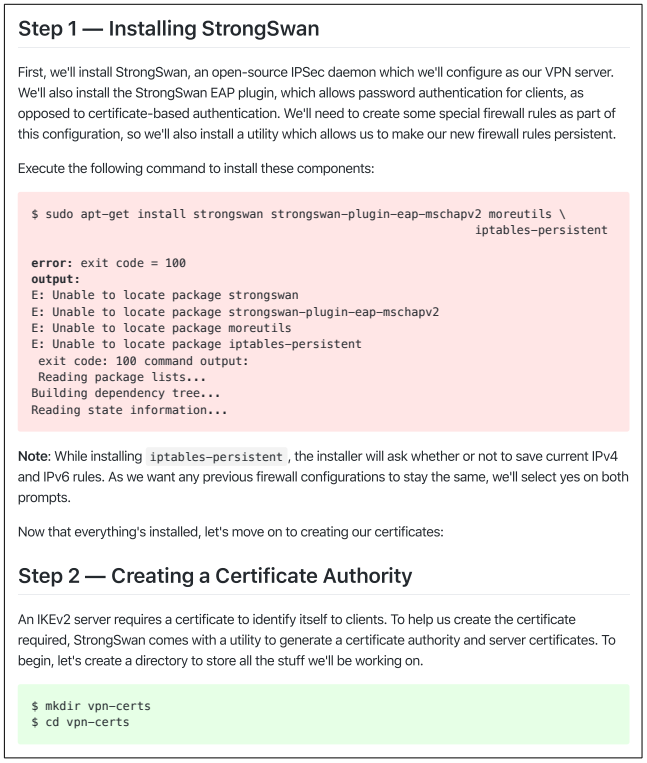

Figure 4: Docable can generate a report that enables inspection of tutorial execution. Here, the 'apt-get install' command did not successfully execute.

model different levels of 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.

2.5.2 Design. The design of Docable 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 3, the text "Let's create a small server using PHP" can be used to select the associated code block. The `file` annotation tells Docable that content should be saved as a file called server.php.

Docable also has the ability to generate a html report that provides the execution status of a command, and includes ancillary information, such as error output and exit codes. An example report is shown in Figure 4.

2.5.3 Annotations. We derived annotations from the results of our open card sort analysis (see Section 3.3). For example, we created the expect annotation based on the *Output* category derived in

Table 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. T/O stands for timed out.

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

Section 3.3.3. 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.

2.5.4 Execution Failures and Peer-Debriefing with Informants. We conduct a qualitative analysis on the reports produced by DOCABLE. We performed an inductive thematic analysis [3] 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* [2]. 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 [29].

Finally, to address the validity of the thematic analysis, we perform a *peer debriefing* [19] with 6 informants, who are responsible for writing and managing technical documentation at major corporations, including 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.

3 NAIVE EXECUTION RESULTS

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

3.1 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-out tutorials (75%).

Table 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	

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, many code blocks in the tutorial 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

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

study by Pimentel et al. [28] 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 [13, 36] and GitHub gists [11] are executable.

3.2 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). 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.

3.3 Code Block Types

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

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.3.4 Presentation (10). We also found ten code blocks that presented ancillary information, such as showing a URL. These code blocks were not directly relevant for execution of the tutorial.

Let's access the new secure website! Open it in your browser:

```
https://YOUR_SERVER_IP
```

3.4 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.

4 HUMAN-ANNOTATION-BASED EXECUTION RESULTS

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

4.1 Annotations

We created 771 annotations for the 40 tutorials. The majority of annotations corresponded to labeling code blocks as commands (455). However, many commands needed to be provided with 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 5).

The docable tool, stepfiles, and tutorials are available at: https://github.com/docable/docable.

4.2 Executability rates

Docable execution of annotated code blocks resulted in an execution rate of 52.3% (343 out of 656 code blocks, when excluding 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."**

4.3 What execution failures remain?

We present the remaining execution failures as themes based on our qualitative analysis of DOCABLE 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 5). 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.

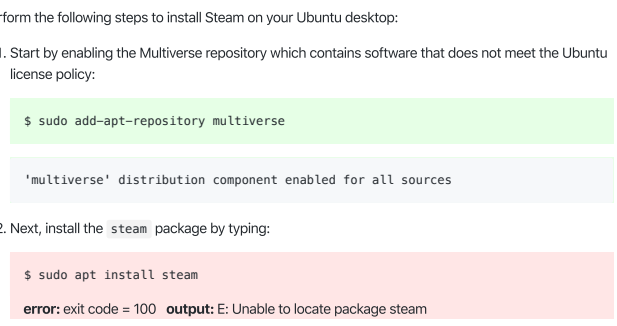

Figure 5: We observed a problem in the Linuxize tutorial for installing the package "steam", which omits an explicit apt-get update command.

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 instruct 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 the tutorial below, 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.

settings.py

```
"""
Django settings for testsite project.

Generated by 'django-admin startproject' using Django 2.0.
...
"""
...
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

# Edit the line below with your server IP address
ALLOWED_HOSTS = ['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 one tutorial, 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
```

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.

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

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 Docable, some informants started systematically reviewing their own tutorials "From our side, since we started playing with docable we made a few corrections on how we write tutorials. Overall, what docable 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 6). 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 docable 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)

After getting the link, download the file:

```
$ sudo wget https://sonarsource.bintray.com/Distribution/sonarqube/sonarqube-7.0.zip
```

Figure 6: 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 the screenshot below, 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.

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)

```
$ 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
```

4.4 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.

5 LIMITATIONS

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

As explained in the study by Kim et al. [15], 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 [20]. To support interpretive validity, we followed the guidelines set by [19] 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.

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

6 RELATED WORK

The work by Kim and Ko [15] is the closest related work in terms of research method and goal. Kim and Ko [15] also performed an qualitative inspection of 30 tutorials across a variety of 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 [15] and our work, an overarching goal is to assess the quality of tutorials; however, whereas Kim and Ko [15] 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 [22], do not explain unexpected errors or side effects, and have unclear adaption paths for tailoring content to different goals [17]. Head et al. [9] 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 [28]. Similarly, only 25–27% of Python code snippets found in Stack Overflow posts [13, 36] GitHub gists [11] 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 [8]. In summary, executability is a useful property for understanding the quality and replicability of software artifacts, including those found in software tutorials.

7 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 [11, 13, 28, 36]. 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 annotated tutorials with Docable 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.

7.1 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 4.3) and interactive prompts (Section 3.1), which can increase the difficulty of successfully executing a tutorial (Section 4.2).

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 [1], such as interactive prompts [17]. 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.

Learners desire more accessible resources that leverage their background [31] and providing alternative formats for flexibility in learning [4]. 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 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.

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
```

7.2 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 4.3) which resulted in lower execution rates (Section 3.1 and 4.2). "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. These can be extended to support annotations that assist with automated execution:

````
Start the service.
```bash|user=root|prompt='$'
systemctl start rot13
```

Test the service by typing in `Hello, world!`.
```bash|input='Hello, world!'|expect=1
client$ nc -w 1 -u 127.0.0.1 10000
Uryyb, jbeyq!
```
````

Another place to invest in improving executability of tutorials is summarizing, updating, and presenting file content (Section 3.3.2). 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. [6] deployed pain-scale surveys in an online learning environment to automatically determine which programming features were frustrating to learners, and Mysore and Guo [23] explored using tutorial profiling to identify problematic instructions. These measures can be combined with "using data such as page bounce rates, shares, whether the user bookmarks a tutorial, whether the tutorial is 'ripped-off' or translated into other languages, etc" (I6).

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

7.3 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 4.3). 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 their effort to *flatten dependencies* "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 [12] 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 [21] could provide a simple mechanism for determining whether or not prerequisites were appropriately satisfied before starting a tutorial.

7.4 Implication IV—Be intentional on whether instructions are human-readable or machine-readable

We observed several executability barriers, such as placeholders in commands (Section 3.3.1), conditional logic in file content (Section 3.3.2, interactive prompts (Section 3.1), and "template values are inconsistent between tutorials and rarely machine readable" (I6), which needed extensive human interpretation (Section 4.1) 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 *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 [21] 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 [7], being uncertain on whether the material is up-to-date [4, 35], and determining what portion might require manual adaption [5, 35].

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).

8 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 executing these annotated tutorials with Docable, 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.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. 2006. Tinkering and Gender in End-User Programmers' Debugging. In *Proceedings of the SIGCHI Conference on Human*

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

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

Factors in Computing Systems (CHI '06). Association for Computing Machinery, New York, NY, USA, 231–240. https://doi.org/10.1145/1124772.1124808

[2] Melanie Birks, Ysanne Chapman, and Karen Francis. 2008. Memoing in qualitative research: Probing data and processes. *Journal of Research in Nursing* 13, 1 (jan 2008), 68–75. https://doi.org/10.1177/1744987107081254

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

[4] Jennifer Brill and Yeonjeong Park. 2011. 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 (January 2011), 5–26. https://www.learntechlib.org/p/33278

[5] Rylan Cottrell, Robert J. Walker, and Jörg Denzinger. 2008. 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)*. Association for Computing Machinery, New York, NY, USA, 214–225. https://doi.org/10.1145/1453101.1453130

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

[7] Denae Ford and Chris Parnin. 2015. Exploring Causes of Frustration for Software Developers. In *Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '15)*. IEEE Press, 115–116.

[8] Hideaki Hata, Christoph Treude, Raula Gaikovina Kula, and Takashi Ishio. 2019. 9.6 Million Links in Source Code Comments: Purpose, Evolution, and Decay. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1211–1221. https://doi.org/10.1109/ICSE.2019.00123

[9] Andrew Head, Jason Jiang, James Smith, Marti A. Hearst, and Björn Hartmann. 2020. 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)*. Association for Computing Machinery, New York, NY, USA, Article Paper 669, 12 pages. https://doi.org/3313831.3376798

[10] Sarah Heckman, Kathryn T. Stolee, and Christopher Parnin. 2018. 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)*. ACM, New York, NY, USA, 1–4. https://doi.org/10.1145/3183377.3183393

[11] E. Horton and C. Parnin. 2018. Gistable: Evaluating the Executability of Python Code Snippets on GitHub. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 217–227. https://doi.org/10.1109/ICSME.2018.00031

[12] Eric Horton and Chris Parnin. 2019. DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 328–338. https://doi.org/10.1109/ICSE.2019.00047

[13] Md Monir Hossain, Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Abram Hindle. 2019. Executability of Python Snippets in Stack Overflow. *arXiv preprint arXiv:1907.04908* (2019).

[14] Glenn D Israel. 1992. *Sampling the evidence of extension program impact*. Citeseer.

[15] Ada S. Kim and Amy J. Ko. 2017. A Pedagogical Analysis of Online Coding Tutorials. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 321–326. https://doi.org/10.1145/3017680.3017728

[16] Sean Kross and Philip J. Guo. 2019. 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)*.

[17] Benjamin Lafreniere, Tovi Grossman, and George Fitzmaurice. 2013. Community Enhanced Tutorials: Improving Tutorials with Multiple Demonstrations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 1779–1788. https://doi.org/10.1145/2470654.2466235

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

[19] Yvonna .S. Lincoln and Egon G. Guba. 1985. *Naturalistic Inquiry*. Sage Publications, Newbury Park, CA.

[20] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and Inter-Rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article Article 72 (Nov. 2019), 23 pages. https://doi.org/10.1145/3359174

[21] Samim Mirhosseini and Chris Parnin. 2020. Opunit: Sanity Checks for Computing Environments. In *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer (Eds.). Springer International Publishing, Cham, 167–180.

[22] Alok Mysore and Philip J. Guo. 2017. 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)*. ACM, New York, NY, USA, 703–714. https://doi.org/10.1145/3126594.3126628

[23] Alok Mysore and Philip J. Guo. 2018. 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)*. ACM, New York, NY, USA, 201–212. https://doi.org/10.1145/3242587.3242633

[24] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in Software Engineering Research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 466–476. https://doi.org/10.1145/2491411.2491415

[25] Mitchell J. Nathan, Kenneth R. Koedinger, and Martha W. Alibali. 2001. Expert Blind Spot : When Content Knowledge Eclipses Pedagogical Content Knowledge.

[26] 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. 2017. The Top 10 Adages in Continuous Deployment. *IEEE Software* 34, 3 (May 2017), 86–95. https://doi.org/10.1109/MS.2017.86

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

[28] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-scale Study about Quality and Reproducibility of Jupyter Notebooks. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*.

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

[30] Daniele Procida. 2017. What nobody tells you about documentation. https://www.divio.com/blog/documentation/

[31] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. [n.d.]. Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language? ([n. d.]).

[32] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[33] Christoph Treude and Maurício Aniche. 2018. Where does Google find API documentation?. In *Proceedings of the 2nd International Workshop on API Usage and Evolution*. ACM, 19–22.

[34] Hazel Virdó and Brian Hogan. 2020. Technical Writing Guidelines. https://www.digitalocean.com/community/tutorials/digitalocean-s-technical-writing-guidelines

[35] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. 2018. How do developers utilize source code from stack overflow? *Empirical Software Engineering* (2018), 1–37.

[36] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. 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)*. ACM, New York, NY, USA, 391–402. https://doi.org/10.1145/2901739.2901767