Codehound: Helping Instructors Track Pedagogical Code Dependencies in Course Materials

Sam Lau
lau@ucsd.edu
UC San Diego
La Jolla, California, USA

Philip J. Guo
pg@ucsd.edu
UC San Diego
La Jolla, California, USA

Abstract

Instructors of programming courses must manage a variety of pedagogical dependencies in their teaching materials. For instance, updating the code used in a single lesson can require cascading changes to other lessons in the course. Currently, they must manually maintain these dependencies across many files, which is tedious and error-prone. To help instructors track pedagogical code dependencies, we created a system called Codehound that uses static analysis to automatically detect where functions are introduced and reused through an entire course. To show how Codehound can be used, we present three usage scenarios inspired by our own experiences teaching large data science courses. These scenarios demonstrate how Codehound can help instructors create new content, collaborate with staff to refactor existing content, and estimate the cost of future course changes.

CCS Concepts: • **Social and professional topics** → Computing education.

Keywords: code dependencies, course material management

ACM Reference Format:
Sam Lau and Philip J. Guo. 2022. Codehound: Helping Instructors Track Pedagogical Code Dependencies in Course Materials. In *Proceedings of the 2022 ACM SIGPLAN International SPLASH-E Symposium (SPLASH-E '22), December 05, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3563767.3568126

1 Introduction

To design effective courses, instructors must constantly update and improve their teaching materials. This process is tedious and error-prone for those who teach technical courses that use code, such as intro. to computer science or data science. In a typical week, these instructors must manage code across different course materials: e.g., a lecture might introduce example code that students need to reuse for discussion worksheets, labs, and homework assignments. Each change that instructors make to their course materials can cause unforeseen cascading changes to the rest of their course. For instance, if they decide to remove even a single lecture slide, they must make sure that the rest of their course materials do not rely on ideas or example code from the removed slide.

In essence, coding instructors face the challenge of what some researchers have called "intricate pedagogical dependencies" between their course materials [11]. Unlike traditional software dependencies where one file runs code from another, pedagogical dependencies happen whenever one file reuses example code that was previously introduced earlier in the course. In coding-heavy courses such as data science, each piece of material builds upon and depends on prior material, and keeping track of these code dependencies requires time and effort that could otherwise be used for teaching.

Beyond individual courses, this challenge also appears in long-form instructional content. For example, CS textbook authors (e.g. [15, 16]) must maintain pedagogical dependencies of example code that appear in different book chapters.

How can we help instructors manage intricate pedagogical dependencies in the code they use for teaching? To address this question, we created *Codehound*, a prototype that allows instructors to track pedagogical dependencies between code in course materials. Codehound uses static analysis to automatically figure out where functions are introduced and reused in a course. It presents this to the instructor directly within their development environment to help them ensure that their course materials are consistent with each other. To demonstrate the capabilities of Codehound, we present three example usage scenarios drawn from our experiences teaching and updating large data science courses. These scenarios show that Codehound can help instructors to create new content, collaborate with TAs to refactor existing content, and estimate the cost of updates.

2 Related Work

To our knowledge, Codehound is the first system that helps instructors track code dependencies across course materials.

One set of related work describes systems to help instructors manage assignments in coding courses, such as those

SPLASH-E '22, December 05, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9900-5/22/12.
https://doi.org/10.1145/3563767.3568126

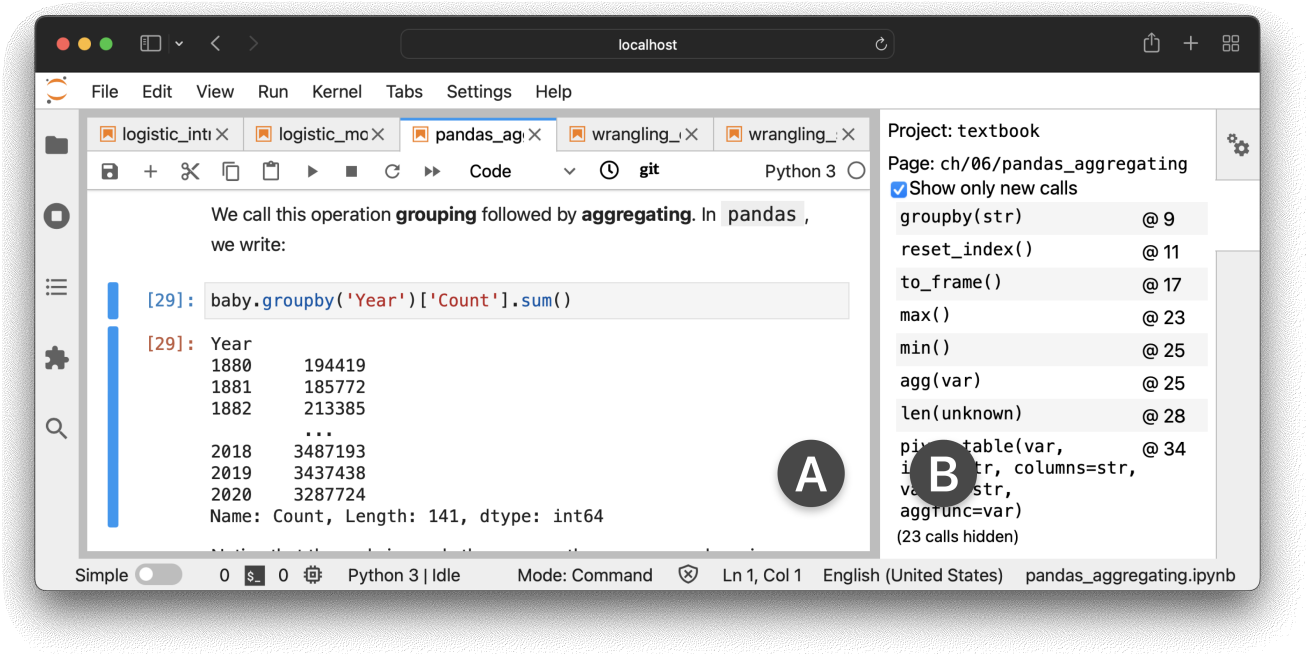


Figure 1. Codehound uses static analysis to help instructors track pedagogical dependencies in course materials. A) Instructors can develop their materials using familiar development environments, such as Jupyter notebooks for data science instructors. B) As they edit example code in a lesson, Codehound automatically updates a display of functions that this lesson introduces for the first time in the course. Clicking on a function displays a list of all the files in the course that also use this function.

that help them grade student work by executing and testing code [13, 14, 17], clustering submissions by identifying student tactics [7], and grading handwritten submissions [18]. Another set of systems directly help students complete assignments, for instance by providing automatic feedback [4, 22], clarifying program outputs [2], and using fuzz testing to generate test cases [19]. Others describe how instructors use software tools to manage student submission files [5, 21]. In contrast to this prior work which focuses on the logistics of running a course, Codehound aims to help instructors create coding-related instructional materials.

An analogue to maintaining instructional code comes from software engineering research [20]. For example, software maintenance [3] and evolution [8] are classic topics that have been studied as early as the 1960s. In our context, instructors write code that they plan to teach to students. However, this code also needs to be maintained and updated as underlying computing environments change over the years. For instance, example code presented in lectures need to be updated when programming languages and software packages update [11]. Like software engineers who update their code in response to new specifications, instructors also update code when they want to change what they teach, for example as they collaborate and receive feedback from other instructors [12]. Codehound adapts ideas like refactoring [6] from software engineering and applies them to instructional design.

3 System Design and Implementation

We drew from our own experiences as instructors and from prior research on the challenges instructors face in managing code-intensive courses [11] to set two design goals:

D1. **Embedded within existing workflows.** Instructors already need to work with a set of familiar software apps in creating courses, so a system should not require them to switch to yet another new application.

D2. **Leverage sequential course structure.** Instructors order their course content carefully to facilitate learning, so a system should help them efficiently rearrange and refactor their lesson sequences.

Our Codehound prototype is implemented as a JupyterLab extension using Python and Typescript, which lets instructors use Codehound as a sidebar in the same browser window as the course materials they are developing (D1). We chose Jupyter [9] since this is a common way for data science instructors (like ourselves) to create course materials such as lectures, assigned readings, and homework assignments: computational notebooks like Jupyter allow explanatory text and runnable example code to be interleaved [10]. Note that our ideas are not specific to Jupyter, though; a similar system for different target audiences could be implemented in an IDE like Visual Studio Code or a LaTeX editor like Overleaf.

The Codehound sidebar shows function calls made by the code in the Jupyter notebook that the user is now editing.

The key idea is to make the user aware of which calls are *new*: i.e., those that have not appeared before in the course (D2). Showing new function calls alerts the user to the fact that they may want to write explanatory text to introduce the concepts embodied by those functions; otherwise students may be confused to encounter a function without knowing what it does. (See Section 4 for more example usage scenarios.)

By default, only new function calls are displayed, and the sidebar has a toggle for the user to display both new and old function calls. Codehound also displays the cell number next to each function call. When a user clicks on the cell number, their cursor will jump to the corresponding cell in the notebook. Whenever the user saves their notebook, Codehound automatically updates the function list to match the user's changes. When the user switches to a different notebook (which usually corresponds to a different lesson in the course), Codehound automatically switches to showing the function call list for the newly-opened notebook.

The Codehound sidebar displays only the function calls in the currently-opened notebook by default. The user can also select a function to find out what other notebooks in the course also call that function. To do so, they can click on the function name in the Codehound sidebar. This causes Codehound to display a list of notebook names that have called the function, in the order that the notebooks appear in the course. This is essentially like navigating a function call graph in an IDE, except that Codehound is aware of the order of instructional materials presented in a course (D2).

Codehound's implementation is summarized in Figure 2. To understand whether a function call is new or not, Codehound requires the instructor to specify the order that students will see each notebook in the course (D2). In our current prototype this ordering is written in a separate YAML file, although future iterations of Codehound could automatically parse this ordering from a course's website or syllabus.

Codehound uses this YAML file to extract the order that function calls appear in course materials. To do this, Codehound parses the Python code in each Jupyter notebook. It uses the `libcst` library [1] to construct an abstract syntax tree for each code snippet. Each time Codehound encounters a function call, it records the function name, its arguments, and the notebook where the call appeared. Codehound also parses function calls within method chain expressions. For example, for this snippet of `pandas` data science code:

```
df.groupby('a').mean()
```

Codehound extracts both `groupby()` and `mean()` calls. This chaining pattern is very common within data science code.

To figure out whether a function call is new or not, Codehound marks each call with the number of times that function has been called up to that point in the course. If the function has never been called before, it is considered to be newly-introduced, since that is the first time students would encounter it in the course. Codehound advises the instructor to write an explanation of what that call does at this point.

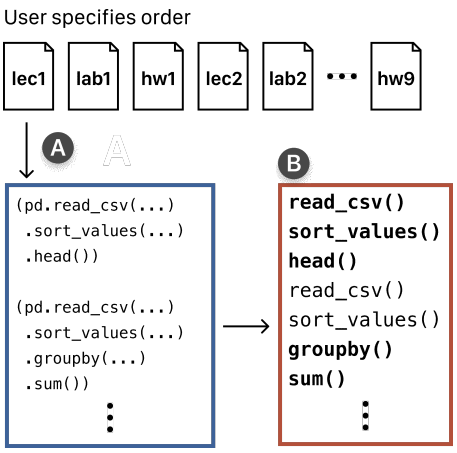


Figure 2. To use Codehound, users specify the order that each piece of content (e.g., a lecture, lab, or homework) is presented in the course. A) Codehound reads in instructional code in the same order it appears in the course. B) Codehound parses the code and extracts each function call in order of appearance (newly-encountered functions are shown in bold).

Scope and Limitations: Codehound's approach is most useful in domains where students reuse a relatively-small API throughout the course, like data science courses that use a subset of the `pandas` library. Currently Codehound does not parse other types of expressions like indexing, slicing, or boolean mask expressions that appear in introductory data science courses. A future iteration could use more sophisticated static analysis to extract these types of expressions.

4 Example Usage Scenarios

We present three example usage scenarios for Codehound to demonstrate its potential capabilities. These scenarios revolve around a hypothetical instructor named Mel who is teaching an introductory data science course with a team of teaching assistants. Each of these scenarios is based off real-world problems that we have personally encountered in our experiences teaching and developing course materials.

Note that we and other data science instructors create courses using a set of Jupyter notebooks to hold lecture materials, readings, labs, and homework assignments; but the same ideas could apply to any document format with code that can be extracted by another program.

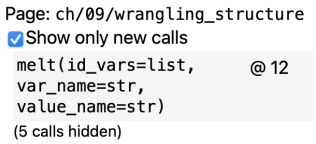
4.1 Creating New Course Content

Codehound can help instructors when they are creating new course content. Let's say Mel is working on a new lesson to introduce data cleaning concepts. For this lesson, she

creates a new lecture that includes both explanatory text and example code in a Jupyter notebook. At this point in the course, students have already had practice with the Python pandas data science library, so Mel wants to apply their knowledge in the context of data cleaning and reuse pandas methods that her students are already familiar with. To do this, she first inserts the notebook into her course's YAML syllabus so that Codehound knows the order of her lessons.

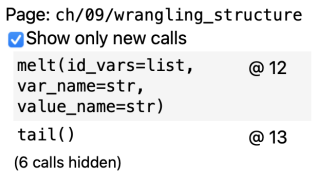
As Mel works on her notebook for this new lesson, she periodically checks the Codehound sidebar. When she writes a function call that students have already seen in a previous lesson, that function will not appear in the sidebar's default display. This lets Mel verify she is using function calls that students are already familiar with, to minimize confusion.

As Mel works on her lesson, she eventually writes example code that calls the `melt()` method from the pandas API. Because this function call is new for her students, it suddenly appears in the Codehound sidebar:



When Mel sees this, she knows she cannot assume that students will understand this `melt()` function call at first glance – Codehound reminds her that she needs to explain what it does, so she writes an explanation in her lesson.

Later, Mel writes example code that uses the `tail()` method. Since this method is relatively simple and common in real-world usage, Mel thinks that it should have already been covered by this point in the course. To her surprise, Codehound also marks this method as new:



Mel sees this and realizes that she should have put the `tail()` method in her *previous* lecture on pandas, so she edits the previous lecture's notebook. After making this change, Mel returns to designing her lesson on data cleaning. Since `tail()` has now previously appeared in the course materials, Codehound removes it from its list of newly-introduced functions. Now Mel can continue working on her lesson.

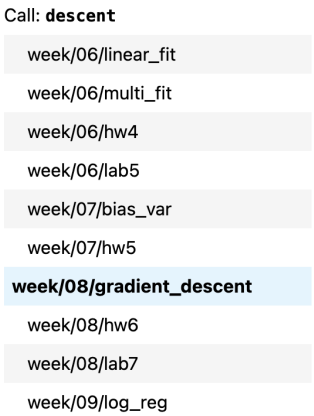
Codehound is useful here since without it Mel would need to *manually and continually verify that she is reusing functions that have previously been introduced in order not to confuse her students.* However, data cleaning uses many different pandas functions, and checking each function one-by-one is time-consuming and error-prone. It would be especially difficult for Mel to see whether a function like `melt()` is being introduced for the first time. She would need to manually look through all of her previous course materials to know whether `melt()` is familiar or brand-new for her students.

4.2 Refactoring Existing Course Content

Codehound can also help an instructor who is working with TAs to refactor (e.g., move around) existing course content.

Let's say Mel currently has a lesson on gradient descent in week 5 of her course, but she feels that it makes more pedagogical sense to move that lesson later in the term to week 8. To move this lesson, Mel updates her YAML course syllabus so that Codehound knows the new ordering of her course material notebooks. The gradient descent lecture defines a function called `descent()`. Since this lesson used to appear in week 5, the `descent()` function is called multiple times throughout lectures and assignments after week 5. After moving her lesson around, now Mel needs to make sure that `descent()` is not ever called until week 8 (after the lesson on gradient descent) or else students will be confused.

Right after moving gradient descent to week 8, Mel opens her lecture notebook on gradient descent and looks at the Codehound sidebar. The `descent()` function does not appear in the default view of the sidebar since it had been called in earlier weeks (somewhere between weeks 5 and 8). To see where it appeared, Mel unchecks the "Show only new calls" box to see all function calls, not just new ones. When Mel clicks on the `descent()` function, Codehound shows a list of notebooks where that function is currently being called:



She sees there are several notebooks that use `descent()` *before* week 8. Now Mel can make changes to her lectures in weeks 6 and 7 so they do not use `descent()`. There are also a few assignments that appear in weeks 6 and 7 that use `descent()`. Since Mel now knows all of the notebooks that use `descent()`, she can tell her TAs which assignments need to also be updated. When her TAs go through assignments, they can use the Codehound sidebar to find all the cells that use `descent()` and make the necessary edits there.

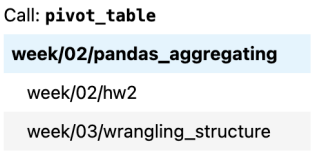
After Mel and her TAs finish refactoring their course materials, Mel can verify that `descent()` indeed appears for

the first time in the gradient descent lecture in week 8 by seeing it in the Codehound sidebar as a new function call.

4.3 Estimating the Cost of a Potential Course Update

Like many instructors, Mel constantly looks for ways to streamline her course content. From experience, she knows that learning the `pd.pivot_table()` function is challenging for students. She has a sense that this function could be cut out of the course without major changes to other materials, but she wants to verify whether her intuition is correct.

To estimate the cost of taking out `pd.pivot_table()` from her course entirely, Mel can open any notebook in the course that uses the function. Then she can click on `pd.pivot_table()` in the Codehound sidebar list to see the list of notebooks that currently use this function. She sees that the function is used in only three notebooks:



Now Mel knows that `pd.pivot_table()` can be trimmed out of course materials with updates to only a few notebooks. If she had instead seen that this function was used in many notebooks throughout the course, then she might have decided to keep the function in the course.

5 Discussion

Here we propose ways to expand Codehound's approach to other tasks that instructors do. We also introduce a broader vision for courseware engineering tools for instructors.

5.1 Applying More Sophisticated Code Analysis

Codehound demonstrates that even simple static analysis can help instructors with course materials. What about more sophisticated analyses? A future version of Codehound could not only track individual function calls but also other types of programming constructs like `if` statements, list comprehensions, and other coding idioms. A more sophisticated analysis could also track higher-level ideas that require several function calls. For instance, data scientists use looping and random sampling together to perform simulations, so a future Codehound could tell instructors whether a simulation scheme is being shown for the first time in a course.

Since Codehound runs in the instructor's development environment (e.g., Jupyter or an IDE), a future version of Codehound could also use runtime analysis to surface information to the instructor. For example, data science instructors often introduce a dataset that is reused in multiple lessons. If an instructor edits or reorders their lessons, they also need to make sure that the dataset is properly introduced when it appears for the first time. Codehound could use runtime analysis to inspect variable values, which would let it tell instructors where a dataset is currently being used in course materials and where it was introduced for the first time.

One salient observation from our own teaching experiences is that instructors need to manage code across many types of files and applications. Code not only appears in scripts and computational notebooks but also in lecture slides and PDF worksheets. Regardless of where code appears, it needs to remain consistent with the course sequence. A future version of Codehound could let instructors track code in *any* text box, regardless of the application it resides in.

5.2 Towards a Vision of Courseware Engineering

One inspiration for Codehound was our observation that managing and updating a technical course has similarities with maintaining a large open-source software project. Both instructors and open-source software maintainers need to keep track of multiple files that contain code and depend on each other in subtle ways. Software development tools help engineers perform routine tasks like refactoring and linting. However, instructors lack analogous tools that take into account the pedagogical sequencing of course materials.

We envision Codehound as an example of a broader set of tools for *courseware engineering*. Unlike software engineering projects, technical courses contain code that appear in a set sequence of lessons. Courseware engineering tools that are aware of a course's structure can help automate repetitive tasks so instructors can have more time to focus on pedagogy.

For example, software engineering tools use code coverage to automatically tell engineers what parts of a codebase have not yet been covered by test cases. An analogous idea for courses is to track *concept coverage*—instructors often want their homework assignments to reinforce the ideas and code introduced in that week's corresponding lecture. A tool that is aware of a course's sequence and structure can automatically tell instructors whether they wrote code in lecture that was not covered in the homework, or vice versa.

Courseware engineering tools can not only help instructors manage their materials but also help students as they work through homework assignments. Current IDE tools can show function documentation inline. A similar tool for students could expand on this idea by also telling students where the function was previously covered in their course materials and provide specific examples from lecture that they can use as reference. This is important because instructors use course-specific idioms—e.g., where one instructor might use a for-loop, another might instead use functional programming. Courseware engineering tools can take this context into account to help students recall familiar course-specific examples from prior lectures or assignments.

6 Conclusion

In this paper we presented Codehound, a prototype system that uses static analysis and an embedded notebook UI to help instructors track pedagogical code dependencies in course materials. The key insight of Codehound is that *order matters*—instructors want to make sure that each piece of course material reinforces previous ideas and code in a logical way. Codehound's design provides an example of how systems can leverage course structure to help coding instructors plan and refactor instructional content. Using this prototype as a starting point, we advocate for further research into *courseware engineering tools* that take care of behind-the-scenes logistics in order to let instructors devote more time to what matters most: teaching students.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. NSF IIS-1845900.

References

[1] 2022. LibCST Documentation. https://libcst.readthedocs.io/en/latest/. Accessed: 2022-08-18.

[2] Soumya Basu, Albert Wu, Brian Hou, and John DeNero. 2015. Problems before Solutions: Automated Problem Clarification at Scale. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*. 205–213.

[3] Barry W. Boehm. 1976. Software Engineering. *IEEE Trans. Computers* 25, 12 (1976), 1226–1241.

[4] Molly Q. Feldman, Yiting Wang, William E. Byrd, François Guimbretière, and Erik Andersen. 2019. Towards Answering "Am I on the Right Track?" Automatically Using Program Synthesis. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. 13–24.

[5] Joseph Feliciano, Margaret-Anne Storey, and Alexey Zagalsky. 2016. Student Experiences Using GitHub in Software Engineering Courses: A Case Study. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 422–431.

[6] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

[7] Yingjie Fu, Jonathan Osei-Owusu, Angello Astorga, Zirui Neil Zhao, Wei Zhang, and Tao Xie. 2021. PaCon: A Symbolic Analysis Approach for Tactic-Oriented Clustering of Programming Submissions. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E 2021)*. Association for Computing Machinery, New York, NY, USA, 32–42. https://doi.org/10.1145/3484272.3484963

[8] Michael W. Godfrey and Daniel M. German. 2008. The Past, Present, and Future of Software Evolution. In *2008 Frontiers of Software Maintenance*. IEEE, 129–138.

[9] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, and Sylvain Corlay. 2016. Jupyter Notebooks-a Publishing Format for Reproducible Computational Workflows.. In *ELPUB*. 87–90.

[10] Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. 2020. 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)*. IEEE, 1–11.

[11] Sam Lau, Justin Eldridge, Shannon Ellis, Aaron Fraenkel, Marina Langlois, Suraj Rampure, Janine Tiefenbruck, and Philip J. Guo. 2022. 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*. 201–211.

[12] Sam Lau, Deborah Nolan, Joseph Gonzalez, and Philip J. Guo. 2022. How Computer Science and Statistics Instructors Approach Data Science Pedagogy Differently: Three Case Studies. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. 29–35.

[13] David J. Malan. 2013. CS50 Sandbox: Secure Execution of Untrusted Code. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*. 141–146.

[14] David J. Malan. 2022. CS50 Docs: ALL THE DOCS! https://cs50.readthedocs.io/index/.

[15] Brad Miller and David Ranum. 2014. Runestone Interactive: Tools for Creating Interactive Course Materials. In *Proceedings of the First ACM Conference on Learning@ Scale Conference*. 213–214.

[16] Clifford A. Shaffer, Ville Karavirta, Ari Korhonen, and Thomas L. Naps. 2011. Opendsa: Beginning a Community Active-Ebook Project. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*. 112–117.

[17] Chad Sharp, Jelle van Assema, Brian Yu, Kareem Zidane, and David J. Malan. 2020. 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*. 487–492.

[18] Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. 2017. Gradescope: A Fast, Flexible, and Fair System for Scalable Assessment of Handwritten Work. In *Proceedings of the Fourth (2017) Acm Conference on Learning@ Scale*. 81–88.

[19] Sumukh Sridhara, Brian Hou, Jeffrey Lu, and John DeNero. 2016. Fuzz Testing Projects in Massive Courses. In *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*. 361–367.

[20] Hans Van Vliet, Hans Van Vliet, and J. C. Van Vliet. 2008. *Software Engineering: Principles and Practice*. Vol. 13. Citeseer.

[21] Alexey Zagalsky, Joseph Feliciano, Margaret-Anne Storey, Yiyun Zhao, and Weiliang Wang. 2015. The Emergence of Github as a Collaborative Platform for Education. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. 1906–1917.

[22] Lucas Zamprogno, Reid Holmes, and Elisa Baniassad. 2020. Nudging Student Learning Strategies Using Formative Feedback in Automatically Graded Assessments. In *Proceedings of the 2020 ACM SIGPLAN Symposium on SPLASH-E (SPLASH-E 2020)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3426431.3428654

Received 2022-08-26; accepted 2022-09-30