The Challenges of Evolving Technical Courses at Scale: Four Case Studies of Updating Large Data Science Courses

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

Justin Eldridge
jeldridge@eng.ucsd.edu
UC San Diego
La Jolla, California, USA

Shannon Ellis
sellis@ucsd.edu
UC San Diego
La Jolla, California, USA

Aaron Fraenkel
afraenkel@ucsd.edu
UC San Diego
La Jolla, California, USA

Marina Langlois
malanglois@ucsd.edu
UC San Diego
La Jolla, California, USA

Suraj Rampure
rampure@ucsd.edu
UC San Diego
La Jolla, California, USA

Janine Tiefenbruck
jlobue@eng.ucsd.edu
UC San Diego
La Jolla, California, USA

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

ABSTRACT

Instructors who teach large-scale technical courses, especially on data science and programming, must do a large amount of logistical work when updating their courses. All of this behind-the-scenes labor takes time away from the pedagogically-meaningful work of teaching students. Over the past five years, the authors of this paper have created and updated eight courses for an undergraduate data science program that serves over 2,000 students per year. We present four case studies from our teaching experiences that highlight major challenges in maintaining and updating technical courses: 1) There were intricate dependencies between course materials, so making updates to one part of the course would require updating many other parts. 2) We needed to maintain several variants of course materials such as assignments. 3) We wrote large amounts of ad-hoc custom software infrastructure to manage logistics. 4) We could not easily reuse software written by others. Our case studies point to design ideas for instructor-oriented tools that can reduce the logistical complexities of teaching at scale, thus letting instructors focus on the substance of teaching rather than on mundane logistics.

CCS CONCEPTS

• **Human-centered computing → HCI empirical studies**.

KEYWORDS

course logistics; software infrastructure; maintenance work

ACM Reference Format:
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 (L@S '22), June 1–3, 2022, New York City, NY, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3491140.3528278

L@S '22, June 1–3, 2022, New York City, NY, USA

ACM ISBN 978-1-4503-9158-0/22/06.
https://doi.org/10.1145/3491140.3528278

1 INTRODUCTION

Suppose Alli is an instructor teaching an introductory data science course. She first created this course four years ago, and enrollments have doubled each year. This term, Alli wants to move the lecture on data tables earlier in the course since last year students were confused by this concept. Although this seems like a simple change from a pedagogical standpoint, she runs into major logistical challenges. After moving the lecture, Alli also needs to update many other pieces of course materials to keep them up-to-date, such as discussion worksheets and assignments. But, making these updates requires knowledge of multiple custom infrastructure scripts and third-party software tools that her TA staff uses to generate those course materials. At the scale of Alli's course, she must not only ensure good pedagogy but also pay close attention to variations in hundreds of assignment files, manage autograder software configurations, and debug programming assignments as the underlying software libraries evolve.

This example scenario highlights the reality that large courses, especially on technical topics such as programming and data science, require instructors to manage a tremendous amount of *invisible behind-the-scenes logistical work*. All of that work takes valuable time and energy away from what they actually want to do: teach students. For instance, one of our paper's co-authors said the following during a discussion of the challenges we faced in maintaining and updating our large courses:

> *"I had 14 to 15 hour work days, every day. It was a lot of work, and NOT the fun kind of work because I didn't get to interact with students during any of it."*

Over the past five years, we (this paper's co-authors) have created and updated a set of eight courses for an undergraduate data science program that serves over 2,000 students per year. We have witnessed firsthand the tremendous growth in enrollments to data science and computing majors over the past half-decade [4, 19, 23] – some of our course enrollments have *nearly doubled every year.*

To handle such increasing scale, instructors of technical courses like programming and data science adopt automated workflows to support large class sizes, such as creating software infrastructure to automatically distribute and grade assignments [36]. While these innovations have benefits, they also require lots of work to manage and debug. ***What logistical challenges do instructors face when maintaining and updating large technical courses?***

Challenge	Description	Representative Example
Intricate dependencies between course materials	Course material depends on each other; updating a single lecture can cause worksheets and assignments to become out-of-date.	Adding hypothesis testing concepts to a course created a cascade of unforeseen updates, even for material that was not directly related to the topic. (Section 5.1).
Maintaining consistent variants of course materials	Even single pieces of course content require careful management of multiple file variants and computing environments.	For many assignments, we have an instructor version, a version released to students, and student submissions. However, student Python versions sometimes differed from the autograder's Python version, causing subtle bugs and student confusion (Section 5.2).
Writing ad-hoc software infrastructure to manage scale	Each course has its own scripts and software written by course staff to automate tasks like autograding. However, every piece of infrastructure code requires work to maintain.	One course's infrastructure, which started out as a single file of LaTeX macros, grew to become a collection of scripts that stitched together multiple programming languages (LaTeX, Python, bash) and software tools (Section 5.3).
Cannot easily reuse software written by others	Most software development tools do not fully enable instructors' desired workflows, so instructors still need to write custom code and pay close attention to software updates.	When a third-party software tool released important bug fixes, it also added backwards-incompatible changes that broke our existing course infrastructure (Section 5.4).

Table 1: From our experiences teaching large data science courses, we discovered four main sets of challenges when maintaining and updating course content. This logistical complexity hindered our ability to make pedagogically useful improvements.

This question is important because dealing with logistical challenges takes time away from the pedagogically-meaningful work that instructors want to do. By understanding the complexities that instructors face in their daily workflows, we can move toward streamlining the process so that they can focus on the substantive parts of actually teaching well.

To investigate this question, we present four short case studies drawn from our teaching experiences over the past five years. Each highlights a specific challenge we faced when updating technical courses. Table 1 summarizes each challenge along with representative examples: 1) There were intricate dependencies between course materials, so making updates to one part of the course would require updating many other parts; and oftentimes these dependencies were invisible, which led to inconsistencies that frustrate students. 2) We needed to generate several variants of course materials, such as versions of assignments with and without solutions, then keep those consistent with related materials. 3) We wrote large amounts of ad-hoc custom software infrastructure to manage course logistics such as running student code on servers, accepting homework submissions, doing both automatic and manual grading, and maintaining variants of course materials. 4) We could not easily reuse software written by others, such as off-the-shelf learning management systems or even code written by prior terms' instructors.

Research contributions to Learning@Scale: This paper is, to our knowledge, the first to characterize the challenges that instructors face when doing maintenance and update work for large technical courses. We focused on data science courses in particular since they combine concepts from multiple technical disciplines – mathematical theory, applied statistics, computer programming, data visualization, and the social sciences (e.g., the ethical and social context of data use). Thus, we believe that parts of Table 1 can generalize across other types of technical courses. More broadly, we hope our case studies open up a dialogue about how to support instructors of large university courses, who are often temporary lecturers, teaching-track professors, and graduate students [39] who must bear the brunt of this invisible behind-the-scenes labor.

In sum, this paper's contributions to Learning@Scale are:

- Case studies that highlight four sets of logistical challenges in maintaining and updating large technical courses.
- A call to develop better practices and domain-specific tools to support instructors in managing these challenges so that they can focus more of their time on teaching.

2 RELATED WORK

To our knowledge, our paper is the first to describe the challenges that instructors face in maintaining and updating materials for large-scale technical courses.

The closest related papers describe the implementation of specific *components* of large-scale courses. For instance, Sharp et al. described the code submission, execution, and autograder framework used in Harvard's large intro. programming course and MOOC (CS50) [36]; lead instructor David Malan also documented its server sandbox environment [29] and the entire suite of open-source software infrastructure [28] used to manage CS50 logistics. Basu et al. [5] and Sridhara et al. [38] presented innovations in automated code grading and feedback systems deployed in UC Berkeley's intro. programming course. Others have documented how instructors use GitHub to manage course materials and to enable students to submit coding assignments [14, 42]. Many such tools are focused on programming-related courses, but some are topic-agnostic: Gradescope facilitates AI-assisted grading of handwritten assignments

and exam questions [37]. PeerStudio [26] and Talkabout [25] facilitate peer feedback and small-group discussions, respectively, in large-scale MOOCs with open-ended learning activities. And ProjectLens helps instructors and students manage group project logistics in HCI MOOCs [7]. Similarly, we discovered that assignment submission and grading infrastructure were hard to maintain. But in contrast to these related papers, which each focus on the technical details of individual software components, our case studies instead focus on *instructors' holistic experiences* in maintaining and updating course materials in the face of increasing scale.

The closest analogues to the activities that we describe in this paper come from software engineering research. Specifically, software maintenance [6] and evolution [16] are classic topics of study that date back to the birth of computer software in the 1960s. Unlike physical systems, software does not physically degrade over time, but it does need to be updated in light of changes to the surrounding environments that it runs upon (e.g., operating systems, software libraries, hardware). Similarly, course materials do not physically degrade but also need constant updating in light of changes to their 'environment' such as new developments in the field. Also, since our courses involve lots of programming, our materials include software components that must be routinely updated, such as incorporating new versions of Python libraries for data science.

Dependencies also make software maintenance more challenging since it is hard to change one component independently from the dozens or hundreds of code libraries that it depends upon [9]. We faced similar issues with dependencies amongst different intricately-linked components of our course materials, which made updates fraught with unexpected surprises. For details, see Section 5.1: Intricate dependencies between course materials.

More broadly, our work relates to collaborative peer production of online resources, such as groups of writers convening to update Wikipedia [17] and other Wiki knowledge bases [20], and programmers convening to produce open-source software [10, 12]. While not as large-scale as some of these projects, the courses we describe in this paper often have a dozen or more staff members collaboratively updating materials that consist of dozens of lecture slide decks, lab handouts, code examples, and homework assignments. This collective work is reminiscent of Geiger et al. describing the "invisible and infrastructural work" involved in keeping open-source software projects running, such as "uncredited" behind-the-scenes tasks like updating documentation [15]. In our case, instructors spend large amounts of time on invisible infrastructural work to keep their course materials maintained and up-to-date.

3 METHODS

This paper investigates two main research questions:

- Why do instructors need to make updates to large-scale technical courses that are already well-established?
- What challenges do instructors face when maintaining and updating these large technical courses?

We addressed these questions by reflecting on our own experiences as data science instructors and synthesizing them into four short case studies. Each case study presents a challenge we faced along with a reflection of its broader implications for our research questions. The main benefit of this case study method is that we can candidly introspect on our own experiences and discuss them as co-authors in a way that is not as feasible with external interviews or surveys. (But see Section 3.3 for limitations of this method.)

Course Topic	Students	TA	Terms	Instructors
Intro. to Data Science	280	16	16	5
Data Structures	250	5	11	2
Theory for Data Science I	150	9	14	5
Theory for Data Science II	150	9	13	2
Algorithms for Data Science	150	10	13	3
Applications of Data Science	100	9	12	3
Data Science in Practice	550	10	17	4
Capstone Project	220	6	7	1

Table 2: The courses that this paper's authors created for a data science major at a U.S. university. Students = approximate enrollment per term, TA = number of TAs this term, Terms = how many terms has this course been taught. Instructors = how many different people have taught this course.

3.1 Case Study Participants

For a case study to be effective, its participants must be representative of the general population that its research questions aim to target. In our case, the co-authors of this paper collectively helped create the data science major at a large public Ph.D.-granting institution in the United States. We designed and taught large data science courses for the major, including *all six required lower-division courses*, a popular elective course, and a capstone project course. These courses cover foundational topics in data science, including data manipulation, data structures, visualization, and statistical modeling. We mostly use widely-adopted Python data science tools like `pandas` [30], Jupyter notebooks [34], and `scikit-learn` [33]. Thus, we believe that we are well-positioned to reflect on our first-hand experiences to highlight the challenges of updating technical courses.

Methodologically, our approach follows the precedence set by prior research papers at Learning@Scale that are *case studies of the authors' own firsthand experiences*. Some notable examples include lessons learned from simultaneous deployments of a MOOC and residential course [22], reflecting on five years of Georgia Tech's online masters program [18], a case study of an HCI MOOC with group-based design projects [7], and six years of reflections on an internationalized CS curriculum [35] (Best Paper Award in 2020).

3.2 Overview of Courses

Table 2 summarizes the eight data science courses that we created and updated over the past five years. Average course sizes ranged from 150 to 550 students. We hired large teams of teaching assistants (TAs) to help manage these courses, with three hiring 10 or more TAs each term. TAs held a wide variety of responsibilities, including hosting discussion sections, holding office hours, attending staff meetings, updating and releasing assignments, updating the course website, and grading assignments. We hired both undergraduate and graduate students as TAs, although the majority were undergraduates who did well in prior offerings of the course [31].

Seven of these courses are required for data science majors at our university, and one is a popular elective. Typically, each week we presented three hours of lecture and an hour-long discussion section. Weekly homework assignments consisted of a mix of math, computer programming, and open-ended narrative write-ups.

Moreover, all these courses were well-established, with the majority having been offered in at least ten past terms (see 'Terms' column in Table 2). There are four quarter-long terms per year at our university. Most courses had multiple instructors teaching it in the past ('Instructors' column); for instance, our intro course had 5 instructors teaching 16 offerings over the past five years.

These courses are large-scale and complex to manage not only due to high student enrollments but also because of large staff sizes (often with over 10 TAs), and because they are taught by many instructors over multiple past offerings. Thus, we feel they are appropriate to use as the settings for our case studies in this paper.

3.3 Limitations

The usual limitations of a self-reflective case study apply here. This paper's co-authors all work at the same large public university in the United States. This left out other settings where data science is taught, such as bootcamps, MOOCs, and online workshops [23]. On the other hand, having multiple instructors within the same data science program captures complementary perspectives from different instructors of the same courses. For instance, we often rotate between teaching each other's courses. During an academic year, an instructor typically teaches between 3-5 courses within the program. We acknowledge the limitations of this self-selected sample, so to compensate we tried to estimate how often the issues we encountered came up for peers who taught at other types of universities and reported issues that were more likely to generalize. In the future, interviewing or surveying our colleagues who work at other institutions can help reduce these limitations.

Another limitation is that we only teach data science and programming courses, but other technical courses across STEM fields, such as math and physics, also have large enrollments. We chose to focus on these fields since data science and intro programming are amongst the fastest-growing and highest-enrollment courses at major U.S. universities and MOOCs [4, 19, 23]. We also believe that data science courses are good representatives of technical courses *since they combine concepts from multiple STEM domains*: our courses in Table 2 cover mathematical theory, applied statistics, computer programming, data visualization, and written narrative assignments about the social and ethical context of data. Thus, studying these courses could shed light on problems that courses from other technical domains may encounter at scale. However, our findings may not generalize to all types of technical topics, especially those that do not involve as much math or programming (e.g., biology).

4 WHAT KINDS OF UPDATES ARE NEEDED FOR LARGE TECHNICAL COURSES?

First we describe the kinds of updates that are typical for large-scale technical courses such as our data science courses (Table 2).

We all taught established courses that have each existed for at least three years. Thus, every term we all had the option of "replaying" the course verbatim—we could have reused the material that already existed without changing any of it. In fact, this material was often written by ourselves in a prior offering of the course. Yet, all of us shared a common desire to update our course materials despite the effort required to do so. Here are the three main kinds of updates and common reasons for making them:

1) Improving lecture materials: The most common reason for updating existing lecture materials was observing students struggle with an important concept from the prior term. In response, we often trimmed, expanded, and moved topics. For example, at times we allocated more lectures for basic statistics concepts or decided to move them earlier in the course to give students time to practice. Larger classes often involve more frequent lecture edits to accommodate more students with varying levels of prior experience.

We also updated the examples used in lectures to keep up with current trends. For example, most of us felt students were more engaged when lectures incorporated personal interests or data related to current news events. Related, since data science and programming technologies (e.g., software libraries and APIs) change rapidly year-to-year, the code examples used in lectures must be continually updated or else we risk teaching outdated content.

2) Updating assignments to keep them fresh: In established courses, assignment solutions inevitably get leaked online after several offerings, which increases the odds of cheating. Table 2 shows that most of our courses had been taught over ten times before. Many of us felt that students were negatively impacted by assignment solution leaks, to the point that we avoided repeating an assignment in consecutive offerings. Thus, assignments must be constantly updated to help maintain academic integrity.

Another reason for assignment updates is that data science courses often rely on the latest data or real-time data streams to provide timely and motivating assignments for students. Some assignments used election data or real-time Twitter feeds, so they had to be updated to incorporate the latest data changes.

3) Adding more content as enrollments grow: Student demand for our courses has been at an all-time high due to more people wanting data science and programming jobs. Most of our courses in Table 2 have doubled or tripled in size over the past few years.

We had to create more content as enrollments increased. First, we needed to write more internal documentation for our TAs. To ensure that even new TAs (who are mostly undergraduates) can deliver high-quality instruction, we had to write more detailed TA guides to provide structure for TA sections and create more detailed grading rubrics. We felt that these additional teaching materials were required to maintain a consistent student experience at larger scales. For instance, when our data structures course grew from 60 to 250 students, we made discussion worksheets so that students could get useful practice no matter which section they attended.

Larger courses also mean a greater diversity of student backgrounds; in our experience, this manifested most prominently in the amount of prior computer programming experience that students had when coming into our courses. When we taught the programming-heavy parts, we found that many students either found programming too easy or too difficult. To address this wide variance, we created supplemental course content to help students who had less programming background. For instance, one of us

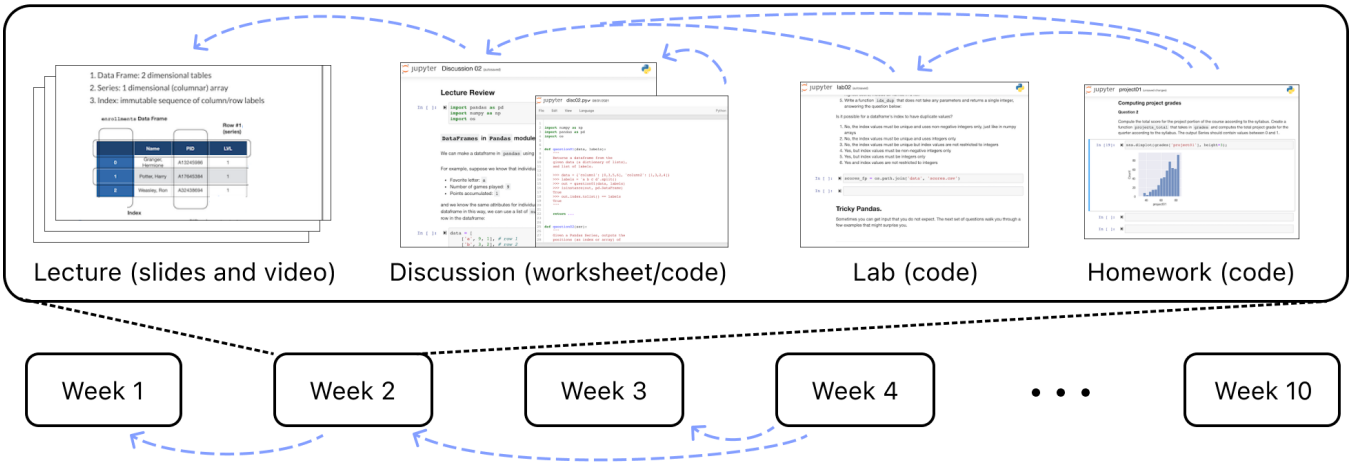

Figure 1: Even within a single week in a course, instructors must manage multiple pieces of course content with intricate dependencies (depicted as dotted blue arrows in this figure). In this example from one of our courses, the discussion Jupyter notebook reinforces topics from lecture. The worksheet is accompanied by a separate Python script which must stay up-to-date with the content in the notebook. Lab and homework also depend on the discussion and lecture content. At a higher level, dependencies exist between subsequent weeks. An instructor who wishes to update a single lecture must ensure that all the dependent material is also updated across the entire course. Instructors now do this manually, which is tedious and error-prone.

created a separate problem-solving guide for the data science theory course. We also created a separate track of content for more advanced students who wanted additional depth for a particular topic, for instance by inviting guest lecturers or by adding more lectures that covered recent advances in the field.

5 WHAT CHALLENGES DO INSTRUCTORS FACE WHEN UPDATING THEIR COURSES?

We distilled our collective experiences of updating our course materials over the past few years into four main challenges, summarized in Table 1. For each challenge, we use a small case study to illustrate a representative example from our experiences. Each case study provides background context, the difficulties we faced in making that course update, and a reflection on broader implications. We conclude each one with a list of other kinds of course updates that resulted in similar challenges.

5.1 Challenge 1: Intricate dependencies between course materials

Figure 1 shows that in technical courses even a single week can involve many pieces of course material that must remain consistent with each other. For example, a typical week might contain two to three lectures. Each lecture has both a set of slides and a Jupyter computational notebook [27, 34] that the instructor presents to show live code and data examples in class. Students also attend discussion and lab sections during the week to reinforce lecture content; those sections each require their own Jupyter notebooks or printed paper handouts. Each assignment also requires its own Jupyter notebook and supplemental files like datasets.

Due to the intricate dependencies shown in Figure 1, if an instructor wants to update any piece of course content (e.g., to fix a bug or improve an example), they must also update a batch of related content to keep it all consistent. And although each piece of content depends on others, these dependencies are hidden and must be maintained manually by instructors and TAs.

5.1.1 Case Study: Substituting or Moving Course Topics. The first course that all students take in our degree program is an introduction to data science. For this intro course, we adapted UC Berkeley's openly-available Data 8 curriculum [11]. However, Data 8 is a 15-week course, so we had to modify it for our university's shorter 10-week academic term. Data 8 is split into three parts: computation (Python), inference (hypothesis testing), and prediction (machine learning). To prune this course down to ten weeks, we did not include inference when we first offered our version. However, a year later we updated our course to make the opposite choice—we removed topics in prediction to teach inference instead.

Why make a change? The motivation for making this change actually came from other instructors in our data science program. Specifically, one of the more advanced data science courses teaches students to do end-to-end data analyses, including hypothesis testing. However, the instructors for that course felt that teaching hypothesis testing from the ground up took too much time away from their core topics, which made us realize it was important for students to learn it first in our intro course. Thus, we decided to replace prediction topics with inference (hypothesis testing) since other courses in the program already covered prediction.

Why did we not include hypothesis testing in the original version of our intro course? Because it was the very first course we created for our data science program, so these later courses did not even exist back then. Although we knew the basic high-level descriptions for later courses, it was not possible to know exactly what would be most important to cover until those courses were actually taught

for the first time and we received student and instructor feedback. Thus, during the first offering we decided to teach prediction instead of inference because we felt that a prediction project related to machine learning would be more interesting for students. But once the subsequent courses were launched, we found that having inference in our intro course was better for students. This is an example of updating our course in the face of changing external circumstances such as new downstream courses being created later.

How did we update our course? We adjusted the lecture schedule to emphasize inference instead of prediction. As we updated course materials, we repeatedly encountered the challenge of intricate dependencies: switching each lecture's topic required updating the discussion worksheet, lab assignment, and homework for that week. We also realized that we could adjust the first half of the course to better prepare students for inference. For instance, our old content did not teach students how to draw random samples from a data table, an important skill for inference. Thus, we updated content not only for the weeks that were directly focused on inference but also previous weeks in the course so that students could be better prepared for inference later. This is an example of an update triggering additional updates of dependencies.

Reflection. This case study illustrates how intricate dependencies exist between materials within a course. But at a higher level, it also illustrates how dependencies exist *between courses*. In our case, an advanced downstream course motivated us to swap out prediction topics for inference. Once we made this change, other courses in the program also needed to adjust since their students would have more practice with inference content but less practice with prediction. For instance, courses that previously assumed that students understood how to fit a classification model in Python could no longer do so. This poses a big challenge for instructors: Although we want to make pedagogical updates to improve our courses, it is difficult to predict how much work these updates will actually take to implement. One update could cause cascading changes to many other pieces of material, but we do not know which other pieces of material will become out-of-date until we review them manually.

5.1.2 Other examples. Here are other times where we encountered the problem of intricate dependencies between course materials.

- Updating references to past assignments in text (e.g. "In the last assignment, we covered these topics.")
- Changing a core software package for working with data tables in Python.
- Moving graph algorithms from an earlier course to a later course in our data science program.
- Removing geospatial data from a course to cover natural language models in more depth.

5.2 Challenge 2: Maintaining consistent variants of course materials

In the prior section we showed how updating one piece of course content often results in needing to update many other pieces of content that depend on it. Here we present a related problem: even maintaining a *single* piece of content in isolation (without worrying about dependencies) requires instructors to manage multiple *variants* of that content. For instance, a typical assignment in one of our courses has at least three variants: 1) an instructor version that contains both questions and solutions, 2) a student version that contains only the questions (not the solutions), 3) each student completes the assignment to produce their own personal variant (a completed assignment) to submit for grading.

5.2.1 Case Study: Reacting to Unexpected Python API Changes. One of our courses contains lessons on text data, regular expressions, and natural language models. We give students an assignment where they write Python code to download books from Project Gutenberg [2] and tokenize the text. However, some students ran into unforeseen problems after we upgraded the course Python version from 3.6 to 3.7. This led us to update the assignment.

Why make a change? In the midst of the term we got unexpected reports from students and TAs about an issue where for some students, their code would run correctly on their personal computers but not on the autograder server that grades assignments. After looking into this, we realized that this problem affected only students who had *not* upgraded to Python 3.7. In 3.7, Python changed the behavior of regular expressions. Before, splitting the string `'ba t'` using `r'\s*'` would result in `['ba', 't']`. But in 3.7, the same regular expression produces `['', 'b', 'a', '', 't', '']`. The challenge here was that some students had installed Python 3.6 from a previous course while others had installed Python 3.7. Our autograder system used Python 3.7, so students who wrote perfectly correct code for Python 3.6 could still fail the tests for 3.7.

Unfortunately, we did not catch this subtle issue until we had already released the assignment. Why not? Our staff-written solution to the assignment did *not* rely on the specific regular expression syntax affected by the Python update, so the staff solution still produced the correct output using either Python version; in contrast, many good student solutions (that were still correct) used a regular expression that broke in 3.7. Also, the webpage that documented the changes in Python 3.7 buried this tiny backwards-incompatible change within a long list of other changes [1]—if it were printed out, this change would be on page 38 within a 42-page document.

How did we update our course? When we found out about this issue, we wanted to update the assignment so that the test cases would detect and notify students when they were using the problematic syntax. To do this, the course staff added test cases to the instructor version of the assignment and regenerated the blank student version of the assignment. However, we then ran into the problem that many students had already downloaded and made progress on the old version of the assignment—some students had even finished the assignment before we discovered this issue. If a student wished to use the new version of the assignment, they would have to download a blank copy of the assignment and copy all of their code over. However, we felt that this was too burdensome for students since there were many pieces of code in this assignment, and we only needed to update one question.

Instead, we sent an email announcement to students to upgrade to Python 3.7 immediately. We also instructed our teaching assistants to make sure students in their discussion sections ran their code using Python 3.7, not 3.6. For future iterations of the course, we emphasized that students needed to install and use the same Python environment as the autograder.

Reflection. This case study highlights the challenges of having multiple variants of course materials. In this example, a single assignment had an instructor file (with both the questions and the solutions) and a blank student file (with no solutions) that needed to stay consistent with each other. But each time a student downloaded the assignment and worked on it, they created yet another file that could become inconsistent with the instructor version. When the course staff fixed a question in the instructor version, all previously-downloaded student copies of that assignment (which could be hundreds in a large course) became out of sync.

Another related challenge with variants of course material is that these variants might be used in different computing environments. In this case study, one version of the assignment ran on the instructor's computer and another ran on each student's computer. And once students submitted the assignment, their code would run on the autograder server, which had its own distinct Linux environment. In other words, instructors must not only ensure that variants of course materials stay consistent but also that the *computing environments for these variants stay consistent as well.* One way to cope with this problem is to have all students use a cloud-based coding environment. Some courses do this, but others give students the flexibility to work locally on their own computers.

5.2.2 Other examples. Here are other times when we faced the challenge of maintaining consistent variants of course materials:

- Changing the dataset used for an assignment often requires rewriting many questions and accompanying autograder tests, then testing those tests to make sure they are robust.
- Creating alternate versions of programming assignments after noticing that the current solutions were posted online.
- Updating lecture and assignment code because a package version updated, for example `pandas` releasing version 1.0.

5.3 Challenge 3: Writing ad-hoc software infrastructure to manage scale

Another challenge of teaching large technical courses is that we find ourselves spending lots of time working as software developers and sometimes as software development managers, even though our primary job is supposed to be teaching. We have to either write lots of *software infrastructure* ourselves or supervise TAs who write the code to manage the complexity of course logistics. See Figure 2 for an example of some software infrastructure that we created.

5.3.1 Case Study: Handling Math and Programming Problems. After students take the introductory courses in our data science program, they then take a theory course on algorithms. When we first launched this course, students completed a weekly assignment in LaTeX. To make an assignment, we created a single LaTeX file that contained all the problems and solutions. Then, we used a library of LaTeX macros to strip out solutions in the student version of the assignment. This system was relatively simple, since we could create the student versions of the assignment using a single shell command. However, this course infrastructure software grew significantly more complex over time as we iterated on the course.

Why make a change? As is typical for new courses, the topics we taught in this course changed many times over its first few years. For instance, the course initially spent several weeks on statistical concepts. However, we later decided to move those topics to another course in the program and focus on algorithm analysis instead. Whenever we added a new topic or changed the order of existing topics, we also needed to update the course assignments to match the lecture schedule (as discussed in Section 5.1). The process of updating assignments was prone to introducing syntax errors or pedagogical bugs since we had to manually copy-paste LaTeX problem descriptions between files.

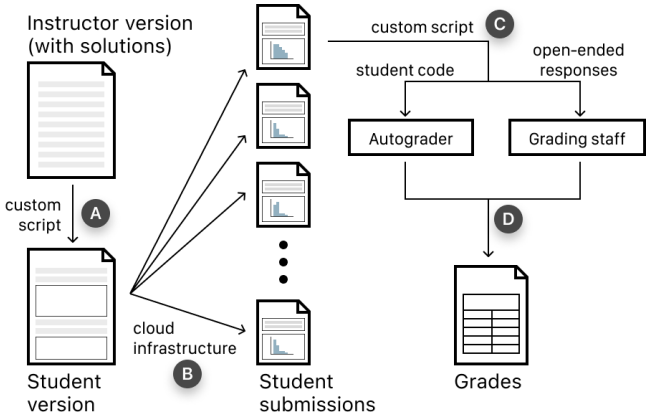

Figure 2: We wrote custom software infrastructure to manage assignment workflows. (A) Instructors use custom scripts to strip out solutions and test cases to generate a student-ready assignment file. (B) Students write and submit assignments to a cloud-hosted computing environment. (C) Another custom script runs an autograder on student code and splits out the open-ended responses for manual grading. (D) Grades are collected into a single score and released to students.

Also, after several course offerings we accumulated a growing "library" of assignment problems for every course topic. For instance, we wrote problems on big-O notation for each offering, so after a few years of running the course we had a number of possible problems to include in an assignment. We wanted a way to choose a few problems from our problem library for each assignment, but there was no existing tool available to do this task.

How did we update the course? Updating course topics meant that we also had to update the ad-hoc infrastructure code that we had written earlier to manage assignment creation, submission, and grading (summarized in Figure 2).

First, we refactored our existing assignments by moving every problem to its own file. We then wrote a script that could concatenate problem files together into a complete assignment. This script started as a simple set of shell commands but grew more complex as we augmented it to handle other tasks. For instance, we added a feature to automatically generate a blank LaTeX starter template for students to write in their solutions so that submissions had a consistent style to make them easier to grade.

We also wanted to give students programming questions in the course, which required a series of extensions to the existing infrastructure that managed our lecture and assignment materials, which were based on Jupyter notebooks. We wrote scripts that

could remove our solution code from Jupyter notebooks, concatenate notebooks together, bundle both code and LaTeX files together for students, and generate configuration settings for the course autograder server. At the time of writing, this course infrastructure consists of an ad-hoc mix of shell scripts, LaTeX macros, and Makefiles that manage LaTeX files, Jupyter notebooks, and Python scripts for assignments.

Reflection. Our software infrastructure plays an important role in helping us manage the scale of our courses—every task that the infrastructure currently handles used to be manual work for the course staff. For instance, we no longer worry about accidentally leaving solutions in the student versions of assignments since our scripts handle this automatically. This is especially useful for helping us manage the many variants of course materials (Section 5.2).

However, this software infrastructure itself requires constant maintenance and debugging since it is a complex piece of ad-hoc software created by educators, not professional software developers. Many courses in our program hire TAs who specifically monitor the course infrastructure so that problems can be quickly addressed.

This infrastructure can grow complex over time. For instance, Figure 2 depicts a typical part of our infrastructure for generating assignments, handling submissions, and autograding them on a server. This software not only needs to be written but must also be fine-tuned every term as assignment details get updated and the underlying run-time environment (e.g., operating systems, library versions, cloud hosting services) change year-to-year.

5.3.2 Other examples. We have also used our infrastructure for:

- Extracting free-response questions from student submissions for manual grading.
- Hosting a cloud-based computing environment so that students all use the same Python and package versions.
- Handling grading special cases like late submission penalties.

5.4 Challenge 4: Cannot easily reuse software written by others

The prior section described how we spend significant amounts of time writing and maintaining custom software infrastructure for our courses. A reasonable question is: *Why don't we purchase off-the-shelf software or use freely-available software written by others?*

First, some of us do use a generic LMS (learning management system) to keep track of logistics like student enrollment lists and gradesheets. But such software is best-suited for static content like posting PDFs of homework assignments. Technical courses like the ones we teach have lots of *dynamic* content such as coding assignments that need autograding infrastructure and different variants of course materials that interleave code, data, and exposition (see Section 5.2). Generic LMS software such as Blackboard, Canvas, or Moodle cannot handle that kind of dynamic content on their own.

We also use free tools such as GitHub and software development toolchains like the classic Unix 'make' [13] for programmatically compiling different variants of course materials. However, we still have to write custom code to wrap around these tools since they were not originally developed with educational use cases in mind. And even though we use tools designed for educational use like Gradescope [37], we still need custom code since each tool uses differently formatted data.

In sum, we have found it challenging to reuse instructional software written by others, as illustrated by the following case study.

5.4.1 Case Study: Handling Software Tool Updates. In several of our courses we rely on an open-source software package for autograding student code in Jupyter notebooks called Otter Grader [3]. This tool was specifically made for educational use, so it has several convenient features. For instance, it can automatically generate test cases from `assert` statements in Jupyter notebooks. However, using this package comes with its own maintenance requirements—when we update this package to use the latest release, it often requires us to also update our course infrastructure and materials.

Why make a change? Since Otter Grader receives regular bug fixes and updates, we prefer to use the latest package version when possible. But these updates also come with their own backwards-incompatible changes. In one notable instance, an old package version could take over 45 minutes to generate student variants of assignments; the new package version fixed this bug so it ran much faster. But when we updated to the new version, we found that other parts of our infrastructure code stopped working since our code depended on old functionality that was no longer available.

How did we update our course? To make our infrastructure code work again, we needed to edit files in every single assignment for the course. We also needed to teach the course staff how to use the changed infrastructure since we updated our existing scripts in the process of updating to the latest Otter Grader package version.

As another example, we added special markup to every assignment that tells Otter Grader what parts of the assignment to convert into test cases. However, in an upcoming package version the markup format will change in a backwards-incompatible way. Thus, when new versions are released and we update our infrastructure, we expect that we will need to manually go through every single assignment file and change the markup to match.

Reflection. This case study highlights the challenges of relying on external software tools when running a course. Even software tools that are specifically designed for instructional use require manual effort to update. Since we lack a single tool that handles every single instructional use case, we cobble multiple tools together in our course software infrastructures. For instance, we use Otter Grader alongside scripts that we wrote ourselves, and student code is graded in another third-party tool called Gradescope that also needs to be configured and hooked up to our other software.

In theory, we could have designed a one-size-fits-all infrastructure that works for all the courses in our data science program, since presumably programming-based courses involve similar sorts of workflows comprised of code submission, autograding, and assignment file variants. But despite these surface-level similarities, in our experience it was still more effective to develop our own software that could be customized for the needs of each course rather than reuse another course's software infrastructure.

In fact, several of us have actually tried to reuse another course's software, only to encounter mismatches that were time-consuming to fix—for instance, a course infrastructure for Jupyter notebooks might not be able to handle LaTeX files. When these moments

arose, we did not want to ask other instructors to spend even more time adding features to their custom infrastructure since it would unfairly take time away from their course.

Also, making software generally reusable by others requires significant time investment beyond simply making the software functional—for instance, software without good documentation is hard for others to use. Since our primary jobs are to teach, not to write and maintain software, our code ends up being highly specific to our course and is not designed for others.

Finally, writing software ourselves makes it much easier to adjust it on-the-fly in response to surprises that arise throughout the term. Many of us value flexibility in our software since our courses change frequently. And when the course's TA staff has expertise in their own software it is much more feasible to fix bugs quickly, which we value especially during the run-up to assignment deadlines.

5.4.2 Other examples. Here are other cases where we encountered difficulties in reusing software written by others:

- Running into merge conflicts when using the Git version control tool, which then prevent Jupyter notebooks from loading.
- Needing to write scripts to transfer grades between multiple grading systems, like the course autograder, manually-graded exams, and our university's gradebook.

6 DISCUSSION

Our case studies shed light on the fact that running, maintaining, and updating large-scale technical courses involve a tremendous amount of invisible infrastructural work [15] that is not directly related to pedagogy. In light of these findings, we now discuss how to design better tools to support instructors in coping with such complexities.

6.1 Instruction at scale lacks the tools that make open-source software projects successful

To reflect on our findings as a whole, we draw an analogy between instruction at scale and open-source software development [12]. In both, multiple stakeholders collaborate to produce a useful product that is shared with a large audience.

Successful open-source projects gain more users and maintainers but must also deal with a growing code base that fills with technical debt [24]. This progression mirrors our experience as instructors since our courses are rapidly-growing not only in numbers of students but also in numbers of TAs and amount of course material that we need to maintain. Open-source software projects also face the challenges that we as instructors encounter with intricate dependencies, infrastructure, and distribution of knowledge. Like instruction at scale, large software projects depend on other libraries that are frequently updated. A single set of source code may generate multiple output variants, for instance to build executables for different operating systems. They also rely on ad-hoc infrastructure via custom build scripts and Makefiles. And complex software contains too many features for one person to track in detail, so project knowledge is distributed among multiple maintainers.

People who maintain open-source software are all-too-familiar with the problem of cascading updates due to chains of dependencies [9], which we also faced when updating our course materials. But instructors lack analogous tools that software projects rely on. For instance, package managers [40] enable automatic dependency tracking—a developer might update a dependency by changing a single metadata file that tracks package versions, running its test cases, and then using code inspection tools to ensure the project still functions as expected. However, these off-the-shelf tools do not account for all the different settings where instructors might use code, like as screenshots embedded within lecture slides or as snippets inserted in LaTeX or MS Word assignment files. More broadly, instructors lack an way to automatically track semantic dependencies based on pedagogical concepts.

6.2 Toward instructor-centered tool design

We now synthesize our findings into ideas for future tools to address the logistical challenges of instruction at scale. As a whole, we advocate for ***instructor-centered tool design***: rather than force instructors to adopt entirely new systems and workflows, tools should acknowledge that courses already have existing workflows, that teaching staff changes frequently from term to term, and that instructors desire flexibility in handling course updates.

As a representative example, consider the dependency problem again from Section 5.1. For instance, an instructor who removes a lecture slide introducing *pivot tables* [41] needs to ensure that no future course content relies on pivot tables. These dependencies extend beyond just keeping software package versions up-to-date; thus, they are not supported by current package manager tools. A future tool might address this by explicitly representing the dependencies between pieces of content in a course. Such a tool could represent dependencies in a fine-grained way, so that removing a lecture slide on pivot tables would alert the instructor to the specific homework questions, discussion worksheets, and exam questions that also need updating. This will make it easier to keep multiple pieces of course content in sync.

However, it is unrealistic to expect that instructors will explicitly set aside time to record every single content dependency, for the same reasons that they currently do not set aside time to document their course updates. To take this into account, future tools could embed themselves directly within instructor workflows without requiring them to switch applications to record dependencies. For instance, we noticed that we often open multiple pieces of dependent material in separate windows to manually ensure that materials remained consistent. A tool for automatic dependency tracking might observe this activity at the operating-system level [32] and provide a prompt to allow the instructor to mark these materials as linked together, or even automatically record the links based on how often two files were opened side-by-side.

Relatedly, instructors do not wish to break out of their workflows in the process of making course updates and thus do not often use outside tools to document their changes. They also hesitate to create yet another piece of content with metadata that they must keep up-to-date. In software development, maintainers often rely on email threads, chat channels (e.g., Slack), and GitHub issues/commit messages rather than monolithic documentation pages. Can future

tools support similar kinds of lightweight contextual documentation for instructors who work in multiple applications and diverse file formats? Such a tool could record down useful context that the instructor provides tacitly [8] as they are working.

7 CONCLUSION

We used four case studies to present some of the complexities of updating large-scale courses related to data science and programming. These challenges include managing intricate dependencies between course materials, content variants, software infrastructure, and software reuse. The main implication of our findings is that large technical courses are like complex engineering systems with hundreds of 'moving parts' that depend on one another in subtle ways. Similar to how programmers have developed a rigorous methodology of software engineering over the past few decades, we call for the Learning@Scale community to move toward an analogous discipline of *courseware engineering* [21]. This would involve not only better design practices but also building domain-specific tools to help manage the logistical complexities of maintaining and updating courses at scale. These efforts will hopefully free up instructor time and energy to focus on what matters most: *teaching students.*

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. NSF IIS-1845900 and Grant No. NSF DGE-1735234.

REFERENCES

[1] 2018. What's New In Python 3.7 — Python 3.9.6 Documentation. https://docs.python.org/3/whatsnew/3.7.html#porting-to-python-37.
[2] 2021. Project Gutenberg. https://www.gutenberg.org/.
[3] 2022. Otter-Grader Documentation — Otter-Grader Documentation. https://otter-grader.readthedocs.io/en/latest/.
[4] Computing Research Association. 2017. Generation CS: Computer Science Undergraduate Enrollments Surge Since 2006. https://cra.org/data/generation-cs/. Accessed: 2021-01-15.
[5] 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* (Vancouver, BC, Canada) *(L@S '15)*. Association for Computing Machinery, New York, NY, USA, 205–213. https://doi.org/10.1145/2724660.2724679
[6] Ned Chapin, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil, and Wui-Gee Tan. 2001. Types of Software Evolution and Software Maintenance. *Journal of Software Maintenance* 13, 1 (Jan. 2001), 3–30.
[7] Hao-Fei Cheng, Bowen Yu, Siwei Fu, Jian Zhao, Brent Hecht, Joseph Konstan, Loren Terveen, Svetlana Yarosh, and Haiyi Zhu. 2019. Teaching UI Design at Global Scales: A Case Study of the Design of Collaborative Capstone Projects for MOOCs. In *Proceedings of the Sixth (2019) ACM Conference on Learning @ Scale* (Chicago, IL, USA) *(L@S '19)*. Association for Computing Machinery, New York, NY, USA, Article 11, 11 pages. https://doi.org/10.1145/3330430.3333635
[8] Anna T. Cianciolo and Robert J. Sternberg. 2018. *Practical Intelligence and Tacit Knowledge: An Ecological View of Expertise* (2 ed.). Cambridge University Press, 770–792. https://doi.org/10.1017/9781316480748.039
[9] Russ Cox. 2019. Surviving Software Dependencies. *Commun. ACM* 62, 9 (Aug. 2019), 36–43. https://doi.org/10.1145/3347446
[10] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work* (Seattle, Washington, USA) *(CSCW '12)*. Association for Computing Machinery, New York, NY, USA, 1277–1286. https://doi.org/10.1145/2145204.2145396
[11] John DeNero. 2021. Data 8. http://data8.org/.
[12] Nadia Eghbal. 2020. *Working in Public: The Making and Maintenance of Open Source Software.* Stripe Press.
[13] Stuart I. Feldman. 1979. Make–A program for maintaining computer programs. *Software: Practice and experience* 9, 4 (1979), 255–265.
[14] 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)*. 422–431.
[15] R. Stuart Geiger, Nelle Varoquaux, Charlotte Mazel-Cabasse, and Chris Holdgraf. 2018. The types, roles, and practices of documentation in data analytics open source software libraries. *Computer Supported Cooperative Work (CSCW)* 27, 3 (2018), 767–802.
[16] Michael W. Godfrey and Daniel M. German. 2008. The past, present, and future of software evolution. In *2008 Frontiers of Software Maintenance*. 129–138. https://doi.org/10.1109/FOSM.2008.4659256
[17] Aaron Halfaker, R. Stuart Geiger, Jonathan T. Morgan, and John Riedl. 2013. The Rise and Decline of an Open Collaboration System: How Wikipedia's Reaction to Popularity Is Causing Its Decline. *American Behavioral Scientist* 57, 5 (2013), 664–688. https://doi.org/10.1177/0002764212469365 arXiv:https://doi.org/10.1177/0002764212469365
[18] David A. Joyner and Charles Isbell. 2019. Master's at Scale: Five Years in a Scalable Online Graduate Degree. In *Proceedings of the Sixth (2019) ACM Conference on Learning @ Scale* (Chicago, IL, USA) *(L@S '19)*. Association for Computing Machinery, New York, NY, USA, Article 21, 10 pages. https://doi.org/10.1145/3330430.3333630
[19] Alexander C. Kafka. 2018. With Student Interest Soaring, Berkeley Creates New Data-Sciences Division. *The Chronicle of Higher Education* (Nov 2018).
[20] Aniket Kittur and Robert E. Kraut. 2010. Beyond Wikipedia: Coordination and Conflict in Online Production Groups. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work* (Savannah, Georgia, USA) *(CSCW '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. https://doi.org/10.1145/1718918.1718959
[21] Carlos Delgado Kloos, Ma Blanca Ibáñez, Carlos Alario-Hoyos, Pedro J Muñoz-Merino, Iria Estévez Ayres, Carmen Fernández Panadero, and Julio Villena. 2016. From software engineering to courseware engineering. In *2016 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 1122–1128.
[22] Joseph A. Konstan, J. D. Walker, D. Christopher Brooks, Keith Brown, and Michael D. Ekstrand. 2015. Teaching Recommender Systems at Large Scale: Evaluation and Lessons Learned from a Hybrid MOOC. *ACM Trans. Comput.-Hum. Interact.* 22, 2, Article 10 (apr 2015), 23 pages. https://doi.org/10.1145/2728171
[23] Sean Kross and Philip J. Guo. 2019. Practitioners Teaching Data Science in Industry and Academia: Expectations, Workflows, and Challenges. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. ACM, New York, NY, USA, Article 263, 14 pages. https://doi.org/10.1145/3290605.3300493
[24] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software* 29, 6 (2012), 18–21. https://doi.org/10.1109/MS.2012.167
[25] Chinmay Kulkarni, Julia Cambre, Yasmine Kotturi, Michael S. Bernstein, and Scott R. Klemmer. 2015. Talkabout: Making Distance Matter with Small Groups in Massive Classes. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing* (Vancouver, BC, Canada) *(CSCW '15)*. Association for Computing Machinery, New York, NY, USA, 1116–1128. https://doi.org/10.1145/2675133.2675166
[26] Chinmay E. Kulkarni, Michael S. Bernstein, and Scott R. Klemmer. 2015. PeerStudio: Rapid Peer Feedback Emphasizes Revision and Improves Performance. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale* (Vancouver, BC, Canada) *(L@S '15)*. Association for Computing Machinery, New York, NY, USA, 75–84. https://doi.org/10.1145/2724660.2724670
[27] 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 *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (VL/HCC '20)*.
[28] David Malan. 2021. CS50 Docs: ALL THE DOCS! https://cs50.readthedocs.io/. Accessed: 2021-01-15.
[29] David J. Malan. 2013. CS50 Sandbox: Secure Execution of Untrusted Code. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 141–146. https://doi.org/10.1145/2445196.2445242
[30] Wes McKinney. 2011. Pandas: A Foundational Python Library for Data Analysis and Statistics. *Python for high performance and scientific computing* 14, 9 (2011).
[31] Diba Mirza, Phillip T. Conrad, Christian Lloyd, Ziad Matni, and Arthur Gatin. 2019. Undergraduate Teaching Assistants in Computer Science: A Systematic Literature Review. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/3291279.3339422
[32] 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* (Québec City, QC, Canada) *(UIST '17)*. ACM, New York, NY, USA, 703–714. https://doi.org/10.1145/3126594.3126628

[33] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, and Vincent Dubourg. 2011. Scikit-Learn: Machine Learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[34] Jeffrey M. Perkel. 2018. Why Jupyter Is Data Scientists' Computational Notebook of Choice. *Nature* 563, 7732 (2018), 145–147.

[35] Chris Piech, Lisa Yan, Lisa Einstein, Ana Saavedra, Baris Bozkurt, Eliska Sestakova, Ondrej Guth, and Nick McKeown. 2020. Co-Teaching Computer Science Across Borders: Human-Centric Learning at Scale. In *Proceedings of the Seventh ACM Conference on Learning @ Scale* (Virtual Event, USA) *(L@S '20)*. Association for Computing Machinery, New York, NY, USA, 103–113. https://doi.org/10.1145/3386527.3405915

[36] 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* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 487–492. https://doi.org/10.1145/3341525.3387417

[37] 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* (Cambridge, Massachusetts, USA) *(L@S '17)*. Association for Computing Machinery, New York, NY, USA, 81–88. https://doi.org/10.1145/3051457.3051466

[38] 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* (Edinburgh, Scotland, UK) *(L@S '16)*. Association for Computing Machinery, New York, NY, USA, 361–367. https://doi.org/10.1145/2876034.2876050

[39] Beckie Supiano. 2018. It Matters a Lot Who Teaches Introductory Courses. Here's Why. *The Chronicle of Higher Education* (Apr 2018).

[40] Wikipedia. 2021. Package manager. https://en.wikipedia.org/wiki/Package_manager. Accessed: 2021-01-15.

[41] Wikipedia. 2021. Pivot table. https://en.wikipedia.org/wiki/Pivot_table. Accessed: 2021-01-15.

[42] 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* (Vancouver, BC, Canada) *(CSCW '15)*. Association for Computing Machinery, New York, NY, USA, 1906–1917. https://doi.org/10.1145/2675133.2675284