Working with Large Code Bases: A Cognitive Apprenticeship Approach to Teaching Software Engineering

Anshul Shah*
ayshah@ucsd.edu
University of California, San Diego
USA

Jerry Yu*
jiy066@ucsd.edu
University of California, San Diego
USA

Thanh Tong
ttong@ucsd.edu
University of California, San Diego
USA

Adalbert Gerald Soosai Raj
asoosairaj@ucsd.edu
University of California, San Diego
USA

ABSTRACT

Prior work has highlighted the gap between industry expectations for recent university graduates and the abilities those recent graduates possess. These works have even specifically recommended that students be given the opportunity to work on large, pre-existing code bases in their undergraduate career. This paper presents our experience teaching a newly-created course called *Working with Large Code Bases*. Guided by a Cognitive Apprenticeship approach to provide an authentic classroom experience that emphasizes the implicit processes and techniques involved in real-world software engineering, the course serves as a practical introduction to the skills and workflow involved in navigating and understanding a large code base. The goal of this experience report is to provide the motivation for key course design decisions, an overview of the course content, and a detailed description of key course components. We present student feedback indicating improved confidence in navigating a large code base and course outcomes related to specific tools and techniques students used in the course. Finally, we provide the full set of course materials we used and actionable recommendations for instructors to administer this course at their own institution, even with limited TA support.

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**.

KEYWORDS

large code bases, Cognitive Apprenticeship, program comprehension, project-based learning

ACM Reference Format:
Anshul Shah*, Jerry Yu*, Thanh Tong, and Adalbert Gerald Soosai Raj. 2024. Working with Large Code Bases: A Cognitive Apprenticeship Approach to Teaching Software Engineering. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024), March 20–23, 2024, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626252.3630755

This work is licensed under a Creative Commons Attribution International 4.0 License.

SIGCSE 2024, March 20–23, 2024, Portland, OR, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0423-9/24/03.
https://doi.org/10.1145/3626252.3630755

1 INTRODUCTION

Traditional computer science and programming courses typically involve students writing their own programs from scratch, with fairly minimal starter code supplied. Consequently, these courses do not offer students the experience of working with an existing code base that has been built upon by many different engineers, and many students finish their undergraduate computer science curriculum without any exposure to techniques involved in working with large code bases. In fact, over 15 years ago, recommendations provided by Begel and Simon, after an observational study of new CS graduates, implored universities to provide students with the opportunity to work on a large, pre-existing code base before they join the workforce [5].

Motivated by these concerns, we designed a course called *Working with Large Code Bases*—a software engineering course that provides an authentic industry-like learning environment for students. We specified 7 learning objectives that were shared with students on the first day of class:

By the end of the course, students should be able to:

(1) Setup a development environment to build and work on software tools from source.
(2) Effectively navigate through a large code base.
(3) Read, understand, and modify parts of a large code base.
(4) Test and debug issues in a large code base.
(5) Understand the workflow to contribute to a large code base.
(6) Effectively communicate how a code base works to others.
(7) Use documentation and online resources to learn just-in-time.

This paper details the design of our course and the rationale behind some of the key design decisions. We share a summary of our students' learning outcomes, as well as specific recommendations for instructors who wish to teach this course but may have limited TA support or instructional resources. We have made all the course materials for this course publicly available at https://cse190largecodebases.github.io/ .

*These authors contributed equally to this work

2 THEORETICAL FRAMING: COGNITIVE APPRENTICESHIP

The motivation to teach this course derives from the theory of Cognitive Apprenticeship presented by Collins et al. [6]. Cognitive Apprenticeship aims to bring the traditional apprenticeship model that is used to teach observation-based tasks, such as cabinetry and blacksmithing, into the classroom to teach reasoning-based processes, such as reading comprehension or essay-writing [6]. Specifically, Collins et al. list two specific things instructors must do to teach students with a Cognitive Apprenticeship framework:

(1) Experts must "identify the processes of the task and make them visible to students."
(2) Experts must "situate abstract tasks in authentic contexts so that students understand the relevance of the work."

Collins et al. argue that when domain knowledge is learned "in isolation from realistic problem contexts and expert problem-solving practices," then it is often not applied in the situations it is more useful for [6]. By "making their thinking visible," experts can facilitate the transfer of expertise to the learner [6]. Because the purpose of our course is to teach students the techniques and tools for working on a large code base—an authentic real-world application of students' programming knowledge—we used the Cognitive Apprenticeship approach when designing our course.

3 RELATED WORK

Our learning objectives were developed based on literature on the academia-industry gap [5, 15]. Specifically, Radermacher and Walia conducted an extensive literature review that showed gaps between industry expectations and students' abilities in technical areas such as testing, use of tools, and design and interpersonal skills such as communication and teamwork [15]. Similarly, Begel and Simon specifically recommended that students have the chance to work on a large, pre-existing code base in university curricula [5].

Many previous works describe advanced, project-based software engineering courses [1, 3, 4, 13, 16–18]. In many such works, the projects require students to design and build their own system throughout a term [3, 4, 13, 16, 18]. The motivation for these courses, justifiably, is for students to build a more complex project than the shorter, well-defined programming assignments they completed in introductory courses. The students learn to adapt to changes in project requirements, work in teams, and use soft skills as they create a project [4, 18]. These courses, while certainly relevant to students' career readiness, typically involve students *creating* their own project rather than contributing to an existing large code base with potentially millions of lines of code, which is what is really expected of software engineers in industry [17].

We found two similar experience reports on courses where students work on large code bases. First, Shepherd et al. introduced "project-sized scaffolding" for an advanced software engineering course in which students work on an existing code base to make simple bug fixes or feature additions with hints and documentation to help students if they get stuck [17]. Second, Tafliovich et al. designed a project-based software engineering course based on the free open-source package `matplotlib` [21]. This course is perhaps the most similar experience report to ours that we found. In their report, the authors describe the team-based project which lasts the whole term, where students learn about important principles such as software modeling, verification, project planning, and technical writing [21]. The goal of the class is for students to work in a team to create a minor feature or bug fix that can meaningfully contribute to the open-source software [21].

Our course adopts a similar approach to Tafliovich et al. by providing an experience for students to work with open source software (though not with the developers of the OSS) and to complete authentic industry tasks. However, we note two key differences of our course that may contribute to the broader set of works regarding project-based, authentic software engineering courses. First, due to budget constraints, our course included only two teaching assistants (TAs) for course of 50 students. Some of the previously-mentioned works have a lower student-TA ratio (Tafliovich et al.: 5-6 students per TA) or leverage industry connections ([3, 13]). Second, to our knowledge, the courses described in these prior works did not (or even, could not) *encourage* students to use recently-popularized large language models such as ChatGPT and Github CoPilot. Therefore, we aim to provide two key contributions to the work surrounding software engineering courses:

(1) All course materials (including lecture recordings, project descriptions, code, etc.) for a course that teaches students relevant techniques and tools to work with a pre-existing, large code base. These materials can be found at https://cse190largecodebases.github.io/.
(2) Specific recommendations for how instructors can teach this course with less TA support, including our experience with encouraging students to use large language models.

4 COURSE DESIGN

A key goal of the instructional staff, based on the Cognitive Apprenticeship theory, was to an create authentic experience of working on a large code base for students. In this section, we discuss how the different course components contributed to this environment.

4.1 Selecting the IDLE (CPython) Code Base

A substantial amount of early effort during the creation of the course was spent on finding the right code base for the course that would be the subject of demonstrations in lectures and all project works. We ultimately selected the Python IDLE (Integrated Development and Learning Environment) code base [8]. IDLE is a lightweight code editor included in every CPython build (i.e., is included in each Python download by default) and is written entirely in Python. We present the criteria we used to select the IDLE code base:

(1) **Students should be able to focus on understanding the code base, rather than the programming language itself.** Given the short time frame of only 10 weeks, we wanted to minimize the time needed to teach students the language of the code base and instead focus our attention on understanding and navigating the code base as a whole. Given the pooled experience of the course staff, we prioritized finding a code base in either Java or Python.
(2) **The code base should be thoroughly documented for students to rely on as they complete their projects.** Throughout the course, students and staff members may

encounter issues that are difficult to tackle just through reading the code itself. Therefore, we wanted a well-documented code base maintained by a large community of open-source developers. For example, we felt the Images-to-PDF code base was not well-documented enough [20].

(3) **The code base should be able to be built within a few minutes without requiring extensive configuration.** While learning about build systems is a part of our course, we wanted to avoid subjecting students to a complex build process that might cause problems due to the potentially limited performance and heterogeneous nature of students' personal devices. For example, we felt the Eclipse code base was too large and complicated to build [12].

(4) **The size of the code base should be large enough to warrant code navigation techniques but small enough for students to reasonably understand within one term.** This criteria was somewhat challenging to fulfill. For example, we thought the Jarvis code base [19] was not complex enough for code navigation techniques but was a reasonable size, written in Python, and sufficiently documented.

4.2 Lectures

The key topics covered in lectures include code comprehension, unit testing, git workflow (including code reviews), and project management. Table 1 depicts the content within each unit. Each item listed in the description section (Code Navigation on VSCode, Using a Debugger, etc.) roughly maps to one lecture. We also invited three guest lecturers from the software engineering industry to speak with students about aspects of their job, such as code reviews, project management, and documentation.

Table 1: Lecture topics throughout the course.

Concept	Lecture Topics
Code Comprehension	Code Navigation on VSCode, Using a Debugger, Diagramming, Using Online Sources (ChatGPT, Google, Documentation), Reading Test Cases, and Making Experimental Code Changes.
Unit Testing	Designing Unit Tests, Understanding Unit Tests, Test Coverage.
Git Workflow	Git Branching, Merge Conflicts, Code Reviews, Continuous Deployment.
Miscellaneous	Introducing Open Source Software (OSS), Project Management Tools, Guest Lecture on Navigating Code Base via Commandline, Guest Lecture on Code Reviews

In line with the theory of Cognitive Apprenticeship, the lectures in the course aimed to demonstrate an expert's approach to realistic and relevant tasks for software engineers. Subsequently, we prepared a similar task for students to complete on their own or in pairs. For example, in the lecture regarding how to use a debugger, an instructor showed students how to use the debugger to identify the line of code where the IDLE Pyshell prompt was set in order to change the prompt from ">>>" to a custom prompt and verbalized their thought process as they demonstrated their approach. Following the demonstration, the students completed a short task that asked them to use the debugger to investigate a different part of the code base. During the 10-15 minutes while students worked on the task, the course staff—one professor and two teaching assistants—walked around the room, offered guidance to students who were stuck, and gave feedback to students who were already done with the activity. Following the activity, the instructors asked for student volunteers to share their approach and demonstrated a correct approach to the activity.

4.3 Projects

All projects revolved around the IDLE code base. For the first half of the term, the students worked on three individual projects where they built, modified, and added to the code base. In the second half of the term, they proposed their own project in teams of three.

4.3.1 Course Policies Regarding Projects. All projects were administered via GitHub Classroom, where each student had their own Git repository that instructors could monitor. Each student used the same repository for all individual projects. For each project, they were required to make regular, descriptive commits and made their project submissions via a pull request. We found GitHub Classroom to be an effective tool to administer this course.

From the start of the course, we encouraged students to use online resources, including large language models such as ChatGPT and Github Copilot. We explicitly demonstrated using resources like Stack Overflow, Google searches, ChatGPT, and Github Copilot during lecture demonstrations as the instructors were solving tasks in front of students. For example, one teaching assistant showed a Stack Overflow post they had made previously when working to understand the code base. We expand on the value of this course policy in Section 7.

4.3.2 Individual Project 1: Building a Code Base. Before asking students to read and modify the code base, we first tasked students with configuring and building the development environment. For this project, students had to download the CPython repository and compile the code according to the setup instructors provided by the CPython developers [9]. We purposely provided very brief instructions to students that directed them to official documentation instead of showing them the step-by-step build process. After this first project, all students should have a working version of Python IDLE on their devices, and a way to view, edit, and run the source code. One issue we encountered in this assignment was the difficulty in installing dependencies on Windows computers, so we recommended students to use Windows Subsystem for Linux (WSL 2), which allows users to run a Linux environment [11].

4.3.3 Individual Project 2: Modifying an Existing Feature. In the second project, students had to improve the *Go To Line* feature in IDLE. The original implementation prompted the user to enter a line number and then moved the cursor to that line number. Unlike more powerful editors such as VSCode, standard IDLE did not support negative indexing and also showed a somewhat confusing prompt that read *Enter 'big' for end of file*, which meant that the cursor would be placed at the end of the file if the user entered a number greater than the total number of lines in the file. Students were

required to modify this feature to a) support negative indexing and b) improve the prompt message to read "Enter a number between 1 and N or -1 and -N", where N is the number of lines in the editor. One possible solution required students to modify two functions in separate files (the *query.py* file and the *pyshell.py* file). However, the main challenge students reportedly encountered was locating the correct place in the code to make those changes.

4.3.4 Individual Project 3: Adding a New Feature. In the third project, students were asked to add a feature to IDLE called *Show Code Outline*, which would be a new menu item in the IDLE "Format" settings. When invoked by the user, the application should display an outline of the user-selected region (or default to the entire file) by showing a hierarchy of all classes and functions defined in the selected region. A particular challenge in this assignment was the feature requirement of writing some output from the code editor window to the accompanying shell window. The difficulty lied in understanding how to invoke a method belonging in one class from another, which meant understanding the rather complex relationship between these two classes in the code base. However, based on their process journals (Section 4.4), students ultimately overcame this obstacle using either office hours, exploring similar parts of the code base for ideas, or discussing with ChatGPT.

4.3.5 Code Walkthrough Presentations. After the first two projects in which students built and modified the code base, students were placed in teams of three to give an brief in-class presentation on a small part of IDLE of their choice. In this presentation, students had to use two techniques demonstrated in class (Debugger, Diagramming, Experimental Code Changes, Code Navigation on VSCode) to teach fellow students about a specific part of the code base. This gave students an opportunity to enhance their technical communication skills,and introduced the class to parts of the code base that may be useful to their group projects.

4.3.6 Group Project. In the group project, teams of 3 students were required to implement *and test* a new feature in the IDLE code base. Each team proposed two detailed feature additions along with an estimate of how much time and work would be required to implement each idea. The course staff reviewed each team's ideas and selected one of the proposal as the group's project. We factored project diversity into our decision-making: When multiple teams proposed the same idea (*e.g.*, supporting tabs), we only assigned the project to one of them. In the rare case where both proposals by a team were rejected, either due to scale or feasibility, we reassigned one of the other teams' unused proposals to that team.

Following the principle of incremental development, we required students to submit periodic pull requests to their GitHub repository when working on the group project so that their code changes could be reviewed by teammates. This gave students further practice reading and reviewing code written by other engineers. In our final assessment of each team's project, their history of pull requests and the quality of these code reviews were visible on Github Classroom and affected their overall group project grade.

At the end of the course, each team presented their final project to the class by demonstrating their finished product, explaining their implementation at a high level, and giving an overview of their test coverage for the code they added. After the presentation, students in the audience were required to ask in-depth questions about the project. Each team's performance during their presentation and Q&A factored heavily into their group project grade.

4.4 Project Assessment

To assess project work, our focus was not solely on if a student's final product functions as intended/proposed, but also on their process. Therefore, we manually reviewed each student's final pull request for each project to mimic the code reviews performed in industry settings. In addition, we asked students to produce two documents to be submitted alongside their code changes: a process journal and a design document.

In the *process journal*, each student had to detail their approach to the problem, including mentioning the struggles they encountered, which tools or techniques they used to overcome these difficulties, and their general thought process as they worked through the project. The goals of requiring the process journal were twofold. First, the process journal elucidated how the class in general approached each project so that course staff could evaluate whether students actually utilized the skills introduced during lecture. Second, the process journal allowed students to reflect on and articulate their own processes—two Cognitive Apprenticeship teaching methods to help students develop metacognitive control over the techniques used to complete tasks.

The second document is the *design document*, written after the completion of each project. This document should contain the key design considerations for the changes, a detailed explanation of the new code, and the effects of the code additions on other parts of the code base. Students were required to include at least one diagram to contextualize their code changes within the broader system.

Overall, students were graded on the completion of project requirements and quality of their design documents and process journals. However, much like a code review conducted in industry, students were given an opportunity to address the feedback given by TAs within 3 days of the initial code review to get points back.

4.5 Coverage of Learning Objectives

The course projects aimed to cover the 7 learning goals listed in Section 1:

- **Goal 1** of students learning to build a development environment from source was targeted by Project 1.
- **Goal 2** of students navigating a large code base was targeted by each project in which students needed to explore a new part of a code base in each project.
- **Goal 3** of students understanding and modifying parts of a large code base was also targeted in each individual project after Project 1 since students made changes to the existing code after needing to comprehend it.
- **Goal 4** of students testing and debugging issues in the code base was emphasized in the group project since students wrote unit tests and provided a test coverage report.
- **Goal 5** of students understanding the workflow of contributing to the code base was covered in all course projects since students used Git branching, made pull requests on Github, and received and provided code reviews.

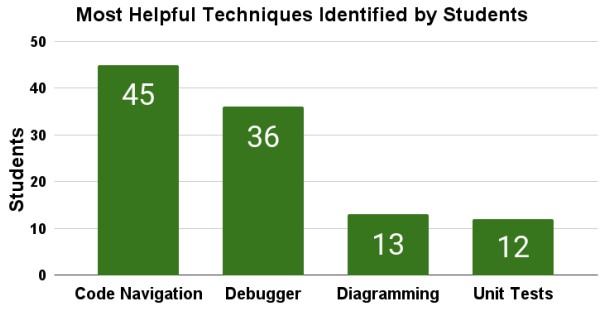

Figure 1: Student responses (n = 46) to the question "What were the most helpful techniques you learned to help you better understand a code base? Choose all that apply."

- **Goal 6** of students communicating how a code base works to others was emphasized in the design document, code walkthrough presentations, and final group presentations.
- **Goal 7** of students using resources to learn "just-in-time" was accomplished through students self-reported usage of documentation, ChatGPT, Stack Overflow, and collaboration.

5 PARTICIPANTS

In total, 51 students filled out the pre-course survey given to students to understand their background and experience with large code bases. Of the 51 students, 44 (86.3%) self-reported using he/him/his pronouns, 5 (9.8%) self-reported using she/her/hers, and 2 (3.9%) preferred not to specify. We expand on this significant gender disparity in Section 7.2. Further, 35 (68.6%) of the students were in their senior (fourth) year of their undergraduate career and 16 (31.4%) were in their junior (third) year.

The prerequisite for our course was a "Tools and Techniques" course in which students learn about commandline tools, Git/Github, and file systems. Students were also encouraged to have completed the core software engineering course in which students build a small application from scratch, but they were allowed to take that course concurrently with our course. To gain a deeper understanding of students' computing background, we asked students about their prior experience with Python (the language that IDLE is written in) and working with large code bases. Only 9 (17.6%) students reported to have worked with a medium to large code base in Python, 12 (23.5%) students reported using advanced Python features such as generators and context managers, and 23 (45.1%) students had worked with a virtual environment in Python. In terms of experience with large code bases, 18 students said they have worked only on small projects before (few files and less than 1000 lines of code), 21 students reported that they have worked on moderately-sized code bases (such as small open-source projects), and 12 students reported that they have worked on a large code base (such as a large open-source project or enterprise code bases).

6 COURSE OUTCOMES

To understand the learning outcomes of our course, we used an end-of-course survey to ask students to reflect on the course content and their own abilities. In total, 46 students (95.83%) filled out the end of course survey out of 48 students who completed the course.

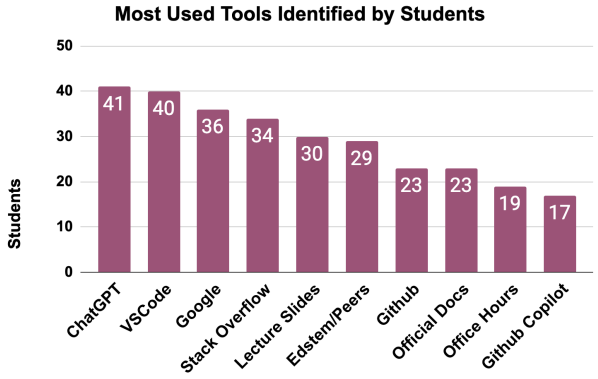

Figure 2: Student responses (n = 46) to the question "What tools and/or resources did you use to help navigate large code bases? Choose all that apply."

6.1 Tools and Techniques

Figure 1 summarizes students' perceptions of the most useful techniques shown in class. Code navigation, which refers to VSCode features and shortcuts such as `Go to Definition`, `Find All References`, etc, and the IDE-based debugger were the two most helpful techniques identified by students. Similarly, Figure 2 shows the tools students used the most to help navigate the IDLE code base. Interestingly, ChatGPT was the most common selection (selected by 89.1% of students) followed by VSCode (selected by 87% of students). To gain a more complete picture of *how* students used the tools and techniques, we asked students "After taking this course, have you developed a general approach towards navigating a new codebase?" One response reads:

> "First, I would look at the file structure and read the main classes... Then, when I want to work with a specific feature, it would be useful to use code navigation to get a general understanding of how the files are connected to each other and where things are located. Finally, when I start writing code for specific features, or when I do not understand what specific lines of code are doing, I would use the debugger or ask [C]hatGPT to see what exactly is going on."

Upon inspection, student answers to this question typically mentioned techniques such as code navigation on VSCode, the IDE-based debugger, and using resources like Stack Overflow or ChatGPT. Through a Cognitive Apprenticeship lens, these tools and techniques are precisely the *implicit processes* of working with a large code base that students may not know how to use effectively before being explicitly shown. We hypothesize that the combination of explicit instructor demonstration (i.e., "making their thinking visible" [6]) combined with an authentic application of such techniques allowed students to see the value of the tools and techniques we demonstrated. Indeed, one student wrote: "I frequently used [VSCode navigation] features and they saved me a lot of time."

6.2 Improved Student Confidence

Figure 3 shows students' self-reported confidence levels in their ability to navigate a large code base at weeks 3, 5, 7, and 10 of

the course. In Week 3, after only completing Project 1 to set up a development environment, only 40% of students responded with a confidence level of 4 or 5. However, by completion of the course, nearly 80% of students reported a confidence level of 4 or 5, with zero students reporting a confidence level below 3. This finding is encouraging, as prior work by Akram et al. showed that improvements' in students self-assessed ability correlates with greater intentions to persist a career in computing [2].

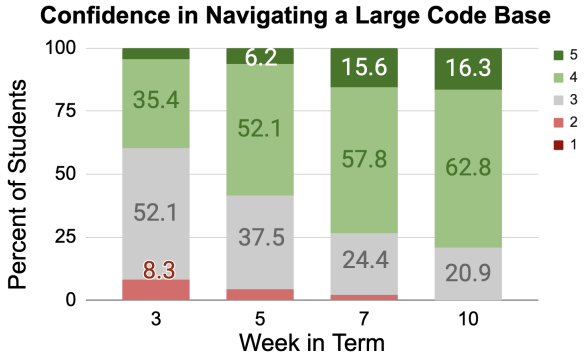

Figure 3: Student responses to the question: "At this point in the course, how confident do you feel about your ability to navigate a large code base?"

7 REFLECTIONS

7.1 The Value of Modeling and Scaffolding

Rather than designing a broad, project-based learning class for software engineering, which may not include experts modeling the implicit processes for completing the project, our Cognitive Apprenticeship approach centered the implicit processes of working with a large code base. Because students were given the opportunity to practice these implicit processes in lectures, such as using VS Code navigation or the debugger, before applying them in open-ended situations, students may have seen the value of such processes in a large code base. We also note that the element of authenticity likely contributed to the effectiveness of the course. By applying the implicit processes *on authentic industry tasks*, students may have gained self-confidence in their ability to work on a large code base going forward—a key learning outcome for our students.

7.2 The Gender Disparity

As noted in Section 5, women accounted for just 10% of our students. However, we note that the instructor and both TAs were male-identifying. In future iterations of the course, we resolve to include female-identifying people on the course staff. Nonetheless, we believe this low representation in our course is part of a systemic problem of poorer retention of women compared to men throughout an undergraduate CS program. As per US Department of Education statistics in 2016, just 18% of CS graduates were women [14]. Another study showed that within four years after college, men were retained in a computing profession at double the rate than women were [7]. This is an incredibly important problem to address. Given the initial, optimistic findings regarding improved self-confidence, we hope that a course such as ours can help women persist in computing.

7.3 Teaching with Limited TA Support

In our course, an instructional staff of one instructor and two graduate TAs supported 50 students. Unlike prior courses that include only a group work component [1, 4, 18, 21], our TAs were able to provide individual feedback for three projects per student *in addition to* a final group project. We wholeheartedly agree with the recommendations provided by Tafliovich et al. about the importance of efficient TAs with strong technical familiarity with the code base [21], and we provide two additional recommendations for educators to teach this course with limited TA support.

Encourage independent help-seeking, especially through use of resources such as ChatGPT, Google, and StackOverflow. In previous courses, students may have been discouraged, or even forbidden, from using online tools and large language models for academic integrity concerns. However, AI tools such as Github Copilot and ChatGPT are increasingly used by developers and businesses [10]. Therefore, encouraging use of online resources not only helped provide an authentic experience for students, but it also likely reduced the instructor and TA workloads. Our course staff believes these tools were vital for students to resolve issues independently, thereby mitigating student help requests for TAs. In fact, 45 (97.8%) of the 46 end-of-course survey respondents mentioned at least one of ChatGPT, Github Copilot, or Stack Overflow as a tool they used to help navigate the code base.

Utilize peer code reviews. Code review proved to be the activity on which TAs spent most of their time on, so a reduction in workload in this area could help staff accommodate more students. From our experience, students were providing high-quality code reviews of pull requests in their project teams later in the course. Therefore, an approach to address this issue is to teach the code review process to students early in the term and assign students to provide code reviews to peers on individual projects. Course staff could use these peer reviews as a "starting point" for code reviews or rely on these code reviews altogether, thereby limiting the time TAs spend reviewing pull requests.

8 CONCLUSION

With a Cognitive Apprenticeship-inspired course design, we aimed to provide an authentic software engineering experience that taught the implicit techniques to work with a large code base. By the end of the course, students reported generally high confidence in their ability to work with a large code base. Through this experience report, we aim to provide two contributions to the existing literature on project-based software engineering courses. First, we provided a course design that uniquely addresses the implicit skills of working on a large code base in an authentic software engineering context and second, we provided explicit recommendations for scaling such a software engineering course to serve a broader, more diverse set of students despite limited TA support.

ACKNOWLEDGMENTS

This work was supported in part by NSF award 2044473. We would like to thank Bill Griswold for his feedback in the writing process.

REFERENCES

[1] Zahra Shakeri Hossein Abad, Muneera Bano, and Didar Zowghi. 2019. How Much Authenticity Can Be Achieved in Software Engineering Project Based Courses?. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training* (Montreal, Quebec, Canada) *(ICSE-SEET '19)*. IEEE Press, 208–219. https://doi.org/10.1109/ICSE-SEET.2019.00030

[2] Bita Akram, Susan Fisk, Spencer Yoder, Cynthia Hunt, Thomas Price, Lina Battestilli, and Tiffany Barnes. 2022. Increasing Students' Persistence in Computer Science through a Lightweight Scalable Intervention. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 526–532. https://doi.org/10.1145/3502718.3524815

[3] Zakarya Alzamil. 2005. Towards an Effective Software Engineering Course Project. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 631–632. https://doi.org/10.1145/1062455.1062575

[4] Craig Anslow and Frank Maurer. 2015. An Experience Report at Teaching a Group Based Agile Software Development Project Course. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 500–505. https://doi.org/10.1145/2676723.2677284

[5] Andrew Begel and Beth Simon. 2008. Struggles of New College Graduates in Their First Software Development Job. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '08)*. Association for Computing Machinery, New York, NY, USA, 226–230. https://doi.org/10.1145/1352135.1352218

[6] Allan Collins, John Seely Brown, Ann Holum, et al. 1991. Cognitive apprenticeship: Making thinking visible. *American educator* 15, 3 (1991), 6–11.

[7] Christianne Corbett and Catherine Hill. 2015. *Solving the Equation: The Variables for Women's Success in Engineering and Computing*. ERIC.

[8] Python Developers. 2023. *IDLE*. https://docs.python.org/3.12/library/idle.html

[9] Python Developers. 2023. *Python Developer's Guide: Setup and Building*. https://devguide.python.org/getting-started/setup-building/

[10] Thomas Dohmke, Marco Iansiti, and Greg Richards. 2023. Sea Change in Software Development: Economic and Productivity Analysis of the AI-Powered Developer Lifecycle. arXiv:2306.15033 [econ.GN]

[11] Windows Subsystem for Linux. 2023. What is Windows Subsystem for Linux | Microsoft Learn. https://learn.microsoft.com/en-us/windows/wsl/about

[12] Eclipse Foundation. 2023. *Eclipse Git repositories*. https://git.eclipse.org/c/

[13] Steven M. Hadfield and Nathan A. Jensen. 2007. Crafting a Software Engineering Capston Project Course. *J. Comput. Sci. Coll.* 23, 1 (oct 2007), 190–197.

[14] US Department of Education. 2016. https://nces.ed.gov/programs/digest/d16/tables/dt16_322.50.asp?current=yes

[15] Alex Radermacher and Gursimran Walia. 2013. Gaps between Industry Expectations and the Abilities of Graduates. 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, 525–530. https://doi.org/10.1145/2445196.2445351

[16] André L. Santos. 2015. Collaborative Course Project for Practicing Component-Based Software Engineering. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '15)*. Association for Computing Machinery, New York, NY, USA, 142–146. https://doi.org/10.1145/2828959.2828972

[17] David C. Shepherd, Felipe Fronchetti, Yu Liu, Daqing Hou, Jan DeWaters, and Mary Margaret Small. 2022. Project-Sized Scaffolding for Software Engineering Courses. In *Proceedings of the First International Workshop on Designing and Running Project-Based Courses in Software Engineering Education* (Pittsburgh, Pennsylvania) *(DREE '22)*. Association for Computing Machinery, New York, NY, USA, 27–31. https://doi.org/10.1145/3524487.3527362

[18] Maurício Souza, Renata Moreira, and Eduardo Figueiredo. 2019. Students Perception on the Use of Project-Based Learning in Software Engineering Education. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering* (Salvador, Brazil) *(SBES '19)*. Association for Computing Machinery, New York, NY, USA, 537–546. https://doi.org/10.1145/3350768.3352457

[19] sukeesh. 2023. *Jarvis*. https://github.com/sukeesh/Jarvis

[20] Swati4star. 2023. *Images-to-PDF*. https://github.com/Swati4star/Images-to-PDF

[21] Anya Tafliovich, Francisco Estrada, and Thomas Caswell. 2019. Teaching Software Engineering with Free Open Source Software Development: An Experience Report. https://doi.org/10.24251/HICSS.2019.931