A Berkeley View of Teaching CS at Scale

Kevin Lin

Over the past decade, undergraduate Computer Science (CS) programs across the nation have experienced an explosive growth in enrollment as computational skills have proven increasingly important across many domains and in the workforce at large. Motivated by this unprecedented student demand, the CS program at the University of California, Berkeley has tripled the size of its graduating class in five years. The first two introductory courses for majors, each taught by one faculty instructor and several hundred student teachers, combine to serve nearly 2,900 students per term. This report presents three strategies that have enabled the effective teaching, delivery, and management of large-scale CS courses: (1) the development of autograder infrastructure and online platforms to provide instant feedback with minimal instructor intervention and deliver the course at scale; (2) the expansion of academic and social student support networks resulting from changes in teaching assistant responsibilities and the development of several near-peer mentoring communities; and (3) the expansion of undergraduate teacher preparation programs to meet the increased demand for qualified student teachers. These interventions have helped both introductory and advanced courses address capacity challenges and expand enrollments while receiving among the highest student evaluations of teaching in department history. Implications for inclusivity and diversity are discussed.

Contents

1 Introduction

> Computer science classrooms are overflowing at colleges and universities across the United States. Enrollments are rising quickly, not only for majors, but also for non-majors who recognize the importance of computing skills in today's economy. This enrollment growth puts enormous pressure on computer science departments, which have not been able to expand to keep pace. [41]

In the decade between 2008 and 2018, CS departments across the United States have experienced double-digit undergraduate enrollment increases while "the overall growth in teaching capacity woefully lags the growth in students [with] the vast majority of departments [reporting] increased difficulty in managing the situation" [55]. From 2006 to 2015, the average number of CS majors in large departments (25 or more tenure-track faculty) increased from 341 to 970 and for small departments from 158 to 499 majors [8]. While growth varies between programs, the data make it clear that "significant growth is under way at many institutions," and that "the conditions exist for continued growth in the demand for CS and related jobs, degrees, and courses" [26].

In this report, we present three strategies which have enabled the effective teaching of large enrollment CS courses at the University of California, Berkeley (UC Berkeley):

Automation The autograder infrastructure and online platforms which are now able to provide instant feedback, minimize manual grading, and deliver courses at scale.

Support The expansion of student support networks through changes in teaching assistant responsibilities and the development of several near-peer mentoring communities.

Preparation The expansion of undergraduate teacher preparation programs to meet the increased demand for student teachers.

These strategies support recommendations previously published by the Association for Computing Machinery [45], Computing Research Association (CRA) [8], the National Academies [26], and other research universities [3, 15, 16, 17, 18, 22, 23, 24, 35, 38, 42]. In light of the national CS capacity crisis and the increasing size of CS courses, this report identifies automation as the force which has driven subsequent changes in support and teacher preparation practices.

1.1 National CS Capacity Crisis

> Current pressures on computer science units are extremely difficult to manage and will also intensify if enrollments continue to grow. Institutional administrators need to work with computer science units to find sustainable approaches to meet the student demand, accounting for important factors such as (1) lack of space for classes and units, (2) academic support required, (3) the limited pool of qualified teaching faculty, (4) the goals and needs of nonmajors taking CS classes, (5) the effect of class size on the course experience, and (6) the desired retention of both students and faculty. [8]

According to the 2017 CRA Enrollment Survey, "66% of the 134 responding doctoral-granting units reported that the enrollment growth is having a big impact (i.e., causing significant challenges) on their unit," with more than 50% of doctoral-granting institutions citing 6 significantly increasing problems due to growing enrollments: classroom space shortages, insufficient numbers of faculty/instructors, insufficient numbers of teaching assistants (TAs), increased faculty workloads, office space shortages, and lab space shortages [8].

In response to these challenges, more than 50% of doctoral-granting institutions have already taken 4 actions to manage student enrollments: significantly increase class sizes, increase the number of academic year sections, increase summer offerings, and reduce low-enrollment classes. More than 65% of doctoral-granting institutions have already taken 4 actions to increase teaching capacity: use undergraduate TAs and tutors, use more adjuncts or visitors as instructors, use graduate students as instructors, and increase the number of teaching faculty [8]. A survey of 78 CS professors from 65 different institutions identified the following three most common approaches for addressing the capacity crisis: (1) altering course offerings by increasing class sizes, offering more sections, and reducing elective offerings; (2) hiring more faculty and TAs; and (3) restricting access to classes, directing non-majors to other classes, and "weeding out" students [30].

Research suggests that certain interventions can significantly affect the recruitment and retention of women and underrepesented minorities (URM) in CS [5, 7, 8, 14, 19, 25, 26, 27, 39, 45].

> The underrepresentation of women and people from groups underrepresented in computing raises concerns for a variety of reasons, including (1) issues of equity and fairness, (2) the economic and competitive imperative of ensuring a large and diverse U.S. workforce, (3) the fact that better solutions are developed by teams with a diversity of people and perspectives, and (4) the increasing interdependency between American democracy and the ability to understand and navigate the presentation of information through technology. [45]

Course-level and department-level policies can directly affect which students pursue the major or have access to advanced coursework. More than 40% of doctoral-granting institutions limit enrollments in high-demand courses, advise less-successful students to leave the major, and require that students are in a major or minor in order to enroll in an advanced course [8]. Even nominally objective policies such as restricting access to the major based on GPA can disproportionately disadvantage URM students [14, 45]. Furthermore, "imposing such restrictions makes the relationship between faculty and students adversarial, causing students to become more competitive and, in many cases, angry," with students concluding that they aren't wanted and perpetuating the idea that "computer science [is] competitive and unwelcoming" [29, 30, 40].

> In the face of increasing enrollments institutions would do well to take lessons from the past. The share of CIS [Computer and Information Science] and CS bachelor's degrees going to women decreased precipitously beginning in the mid-1980s, and again during the dot-com bust. These drops coincided with past peaks in CS degree production, suggesting that high-enrollment conditions or the actions taken by institutions in response to these surges may have contributed to the decrease in representation of women in undergraduate CS during these times. [26]

Indeed, many of the actions taken by universities today mirror the actions undertaken in the earlier enrollment surge in the 1980s which included (1) increasing teaching loads and class sizes, (2) hiring more part-time and adjunct faculty, (3) retraining faculty from other disciplines, and (4) limiting enrollments and access to the major [9]. Many departments have since adopted some of the recommendations cited in the 1982 report including diversifying academic opportunities by creating teaching-track faculty positions and using technology to make education more efficient.

CS education has only recently advanced to the national agenda [1], slowing the adoption of these ideas and practices. "There are few researchers with CS education PhDs, and right now few or no active formal CS education PhD programs," [11] stymieing the development of pedagogical methods and computer science education as a discipline. "Teaching large computer science courses has become a more specialized endeavor," which grows capacity in impacted lower-division courses but results in an increase in student demand for upper-division courses without necessarily solving the underlying instructional bottleneck [40]. There are simply not enough CS teachers. Furthermore, this capacity crisis is occurring at a time of institutional disinvestment due in part to "administrators who are convinced that they [...] know when students will next lose interest" but whose "very decision ensures a capacity collapse" [40] in spite of evidence pointing to the opposite: "While there will probably be fluctuations in the demand for CS courses, demand is likely to continue to grow or remain high over the long term" [26].

1.2 UC Berkeley Case Study

This report presents a case study of three strategies for teaching CS at scale as developed in the Department of Electrical Engineering and Computer Sciences (EECS) at the University of California, Berkeley. Some historical context is necessary to understand the undergraduate CS education program which allowed the development of these strategies. The effectiveness of implementing them at other institutions will vary based on factors such as the institution's size and values [26].

There are two paths into the CS program at UC Berkeley:

EECS The Electrical Engineering and Computer Sciences major in the College of Engineering, to which students apply directly in their application to the university, with a cohort of about 400 students matriculating in 2018.

LSCS The Computer Science major in the College of Letters and Sciences, where students are admitted into the college without declaring a major. Letters and Sciences students can declare the CS major after meeting the requirements for the major. During periods of high student demand and low supply, the LSCS declaration process can be very selective [2, 40].

In 2019, the LSCS major is a capped major, admitting any student with an average 3.3 GPA across three introductory courses with an appeal process for students near the threshold. In 2018, about 800 students were accepted into the LSCS major. Based on current introductory CS course GPA trends, on expectation, about half of students who take the required courses will be eligible to declare the LSCS major, though these enrollments also include EECS majors and a large number of non-majors. Tracking students by their interest in the LSCS major, between 60–70% of interested students successfully declare the LSCS major each year. However, this leaves an estimated 470 interested students unable to declare the LSCS major, including 170 women. This also leaves out students who do not even consider CS due to its reputation as a selective, capped major.

In 2018, between EECS and declared LSCS majors, the undergraduate CS program included over 3,200 majors, representing over 10% of the university's undergraduate student population. In recent years, enrollment pressure has increased not only due to a growth in the number of majors, but also a growth in the number of courses students take per semester. The average number of upper-division EECS courses taken by a CS major throughout their undergraduate degree has recently increased from 5 courses to 7 courses. Students are taking more EECS courses to fulfill major requirements rather than electives offered by other departments. At the same time, the average time to graduation is only 7.89 semesters, as more students in the program are completing their degree in 3 or 3.5 years. Taken together, CS majors are choosing to take more upper-division CS courses in a shorter period of time, inflating enrollment pressure and demand for courses.

Several factors contribute to this growth. Most upper-division CS courses have a short prerequisite chain, usually only requiring the introductory CS sequence, so students can easily switch into another upper-division CS course if enrollment in their first-choice course is full. Furthermore, the program does not require a capstone project which, at many other institutions, introduces an individual advising requirement upon the faculty and consumes student attention in their final year of study. Department surveys show that students' post-graduation plans are increasingly focused on working in software engineering roles, so students value the technical expertise gained from taking technical, CS courses over breadth or personal interest courses.

Demand from non-majors has also increased. Despite the fact that enrollment preference is given to students in the CS program, an increasing number of non-majors are enrolling in upper-division CS courses with two of the most popular courses, Introduction to Artificial Intelligence, and Efficient Algorithms and Intractable Problems, enrolling about 25% non-majors in 2018. Adjacent major programs including Data Science; Cognitive Science; Applied Mathematics; Engineering Mathematics and Statistics; Statistics; and Industrial Engineering and Operations Research either explicitly require or credit certain CS courses towards their undergraduate major degrees. This increase in non-major interest in CS courses mirrors the broader, national trend.

It is this context of external and internal demand for computer science that foreshadowed the Data Science undergraduate program, the "fastest growing program in the history of Berkeley," [2].

> Berkeley's Data Science education program aims at a comprehensive curriculum built from the entry level upward to meet students' varied needs for data fluency. It includes a diverse constellation of connector courses that allow students to explore real-world issues related to their areas of interest and continues with intermediate and advanced courses that enable them to apply more complex concepts and approaches.[1]

The Division of Data Sciences connects the School of Information, the EECS Department, the Statistics Department, the Berkeley Institute for Data Science (BIDS), and faculty, staff, and students from across campus. Introductory data science courses have been developed with lessons learned from introductory computer science [46], and upper-division courses are also designed to scale. Starting as a pilot course with 100 students in Fall 2015, enrollment in the introductory data science course, The Foundations of Data Science, reached 1,500 students in Spring 2019, exceeding enrollments in introductory CS that semester. Core data science courses are commonly co-taught by Statistics and Computer Science faculty while connector courses are offered by many departments across campus to meet the diversity in demand for computational literacy and data skills.

[1] `https://data.berkeley.edu/education`

1.2.1 Course Format

The typical introductory computer science course uses the following model:

Lecture 3 hours per week introducing concepts to the entire class, led by the instructor.

Lab 1–2 hours per week of hands-on exploration activities, led by a TA, with around 30 students.

Discussion 1–2 hours per week of group problem-solving, led by a TA, with around 30 students.

Office Hours A drop-in space for students to ask questions and get help with course concepts and assignments, normally offered on a regular basis by the instructors and TAs.

The typical upper-level computer science course consists of 3 hours of lecture and 1 hour of discussion section per week. The EECS Department has experimented with other course formats as well. Data Structures and Programming Methodology is offered during summer session in lab-centric instruction format that consists of 1 hour of lecture and 6 hours of lab per week [49].

In part due to a shortage of large lecture halls, almost all CS courses have begun webcasting lecture. The campus information technology group records live lecture and posts the video online a few hours afterwards. Many students prefer webcasts over live lecture as they can speed-up, slow-down, pause, and rewind the video, so live lecture attendance in large courses rarely exceeds one third of the true class enrollment by the middle of the semester. Furthermore, lab and discussion section attendance is often not mandatory. Students are typically encouraged to participate, and the course policies may provide incentives for attendance, but attendance is rarely required.

As a consequence, there is a significant number of students enrolled in CS courses who rarely attend lecture but still learn all of the course content by watching webcasts. In 2018, this format was officially adopted by Introduction to Database Systems, which was offered to matriculating UC Berkeley students with the regular lecture sessions replaced entirely by professional recordings on the same content by the professor. To keep students on track, the course expects them to submit short, weekly quizzes on basic lecture concepts. Students are encouraged to attend discussion sections and office hours to clarify concepts from the webcast, build problem-solving skills, and collaborate with other students in the course. While relying on online resources frees demand for resources such as seating in lecture, these students often utilize other components of the course such as discussion, lab, office hours, study groups, and tutoring where learning occurs in smaller group environments. Scaling capacity in these activities has become an increasingly important focus for the department, a theme which is revisited throughout this report.

2 Automation

For decades, CS courses at UC Berkeley have used command-line interface autograding utilities such as the `grading` package. This package allows students to submit files (e.g. programming assignments) to the instructor's UNIX account for grading. Autograding is available through makefiles defined by the instructor for each assignment, and feedback can be automatically emailed to students once the autograder finishes execution. More recent autograding solutions utilize container technology to improve reliability and web frontends to improve the user experience.

In large CS courses, much of the feedback on program correctness is provided by autograders. This has both practical and pedagogical benefits as it reduces the workload of grading student work while supporting students to make progress through independent debugging. These autograding solutions have traditionally been supported by a single computing server provided by the department dedicated to supporting the grading needs of each course. However, when hundreds of students request autograding resources (often in the hours before assignment deadlines), student submissions are placed in a long grading queue. This is an especially important consideration because many UC Berkeley CS courses rely on automated feedback to provide assistance to a large number of students, so a long grading queue is an educational denial of service.

Furthermore, large courses often require additional assistance to manage the flow of information, students, and staff. Beyond the work of grading and returning graded work to students assisted by autograding software, there is a need to improve organization of students and teachers in office hours, and to distribute announcements, assignments, and learning materials to students as the number of students grows beyond traditional classroom capacities. As with autograding, student-facing infrastructure needs to scale as student demand can reach thousands of requests per hour, course rosters can grow beyond 1,000 students long, and there can be tens of thousands of forum posts per semester. These challenges, which can easily become immense administrative burdens at very large scale, have led to the development of specialized software to reduce the administrative overhead of running large courses. For instance, administering exams to over 1,600 students spread across as many as 20 rooms on campus has required the development of specialized software as well as more flexible, student-friendly policies and procedures to enable efficient support of hundreds of exam exceptions and accommodations each semester.

2.1 Grading and Feedback

Two of the most well-known grading and feedback web apps developed at UC Berkeley are Gradescope and OK. These two web apps stand out as particularly unique in how they have enabled new pedagogies and practices.

2.1.1 Gradescope

In Spring 2012, Pandagrader was conceived by the course staff teaching Introduction to Artificial Intelligence to streamline the process of grading paper exams. As instructors from other courses and institutions rapidly adopted the tool, in 2014, Pandagrader was incorporated as Gradescope.[1]

The exam grading workflow begins with scanning and uploading exams to Gradescope. For very large courses with two, high-throughput copy machines, scanning all of the exams can take between 1–3 hours but yields significant time savings during the actual grading through Gradescope's fast online grading interface. Exam scores can then be turned around to students without returning physical papers: students view their graded exams online and see exactly which rubric items were applied. All grading and regrade requests are done over Gradescope's web interface which normally hides the identity of the student and the instructor from each other to minimize bias. The grading process is simpler for homework assignments as students submit assignments online themselves.

By moving the grading workflow online, course staff can increase their grading efficiency. Instead of shuffling papers, course staff assign rubric items to student submissions with only a single click or keystroke, and advance to the next submission with just another click or keystroke. Grading assignments according to an instructor-defined rubric improves transparency to students, helps ensure consistency between multiple graders, and makes it easy for the instructor to adjust point allocations or rubric item description post-hoc. Certain types of questions such as multiple choice or short answer blanks can be graded even more quickly with machine learning-assisted answer grouping where student submissions are automatically categorized into unique answer groups, allowing instructors to grade each group rather than each individual submission. Grading progress and statistics are computed on-the-fly, providing insight into aggregate and individual student success data. Gradescope's distributed grading system is also helpful for grading weekly problem sets. Students submit their work directly to Gradescope. Submissions can be graded online, enabling students to receive feedback the same week or even the day after submission depending on the length of the assignment and the amount of grading support.

[1] `https://gradescope.com`

The fast turnaround time also makes Gradescope a platform for providing formative feedback to students in two introductory CS courses at UC Berkeley. In these courses, students take a short paper quiz in their discussion or lab section during the day. After the section ends, the section TA scans their quizzes and uploads them to Gradescope. In the evening or later in the week, the course instructor and a handful of graders then grades all of the quizzes on Gradescope, identifying the entire class's common misconceptions, and returns personalized feedback to students via email. In end-of-semester course evaluations, students appreciated the additional feedback and found the system as a whole beneficial for their learning. Instructors and section TAs gain a detailed view of student performance previously unavailable in large courses while minimizing additional grading responsibilities. Using the misconceptions collected through weekly quizzes, instructors have released additional, targeted tutorials, practice materials, and study guides to help students improve before taking a high-stakes exams.

In 2017, Gradescope added support for autograding programming assignments. Gradescope's autograding platform is designed on top of container virtualization technology, allowing servers to start new autograder instances as needed and spread load across multiple machines. This ensures that students do not need to wait in a queue: new autograder instances can be provisioned as soon as students submit their assignments and, in times of greatest demand, servers can be dynamically added to the computing resource pool. This approach benefits instructors as they have full control over their choice of grading language and environment in the container. But it also presents a new cost as time needs to be invested in designing small programs or scripts which produce outputs following a specific autograder specification. Manual grading is also possible with grading rubrics and inline comments through a workflow similar to that of the homework and exam assignment types. The recent integration of a system for detecting software similarity has the potential to additionally simplify the workflow for identifying and understanding cases of over-collaboration.

2.1.2 OK

OK[2] is an online autograding platform developed by the course staff teaching introductory computer science at UC Berkeley in Fall 2014. Its primary users are introductory computer science and data science courses at UC Berkeley, each regularly serving enrollments 500–1,500 students per semester. The OK platform provides an alternative to Gradescope's autograding for programming assignments, though its approach favors much deeper integration with the course. In addition to a web interface for students to submit programming assignments, OK provides a Python client that boasts three key features over Gradescope's server-side grading.

[2] `https://okpy.org`

Student-Side Autograding

With student-side autograding, students can run a suite of tests on their computer at any time, providing instantaneous code correctness feedback without needing to formally submit their assignment. These test suites are written in Python doctest format, a format students are familiar with from their practice using with the interactive Python shell, which makes the tests and results easier for students to interpret than traditional unit tests. Complex integration tests can be written in separate files that are seamlessly integrated into the system [44], appearing only after the submission passes targeted doctests. OK presents a simple interface to this feature so that students can autograde any part of the assignment without memorizing a complicated command.

Student-side autograding has resulted in a qualitative change in the way students approach problems as they now often use the autograder as part of a continuous edit-test feedback loop. For every change to a piece of code, students will re-run the autograder to see how the change affects the program. The immediate feedback then informs future planning, implementation, and debugging behaviors. This real-time feedback reduces frustration and builds student confidence by helping them make progress where they would normally get stuck. But there is an inherent risk to this approach. While students may be able to make more progress with the help of the student-side autograders, they may also grow more dependent on the feedback. Several introductory CS courses have experimented with velocity limiting which limits autograder usage (either student-side or server-side) to a certain number of attempts which resets after a fixed amount of time. Students are encouraged to instead try out different debugging and problem-solving techniques such as developing their own examples and running through it on paper to check their understanding. This automated feedback has become an important cornerstone of introductory CS at UC Berkeley.

Test Unlocking

One of the risks of student-side autograding is that students can access tests without thinking through the problem which can lead to a dependency on this development cycle. With test unlocking, the expected output of each doctest is stored as an encrypted string until it is successfully unlocked. Before students can run the student-side autograder, they must explore the problem by unlocking each automated test and determining the expected outputs by hand. Test unlocking reduced the number of conceptual questions (misunderstandings or clarifications of the problem), allowing instructors to spend more time assisting students with more involved questions: unlocking the tests helped students better understand the problem specifications and "work through the thought process" [6]. This result has been cited to provide metacognitive scaffolding [36]. Students later have the opportunity to write and run their own tests, reinforcing program understanding.

Automatic Backups

Each time the student-side autograder is invoked, student work is automatically backed up to the OK server along with metadata on their current progress. This submission and metadata collected by the automatic backup mechanism helps instructors answer questions about student learning. Together with the code backup, the OK client program records analytics such as the current question the student is working on, number of times the student-side autograder has been invoked thus far, and correctness. In addition, because the student-side autograder is run as part of a continuous edit-test feedback loop, the OK server aggregates a large number of intermediary student submissions. Instructors can then track student progress in aggregate and at the individual level to a high degree of detail. Frequent intermediary code snapshots have also benefited the research community as it has generated a massive dataset for education researchers to model student learning and performance [20, 33, 51], deploy learning interventions at scale [31, 44], understand excessive collaboration [54], and propagate feedback at scale [13, 53]. Students have the option to disable automatic backups and metadata collection but still run the student-side autograder by running the OK client with the appropriate command-line arguments.

2.2 Managing Student Learning

Beyond grading and feedback, many other components of a traditional lecture-format course such as office hours, exam scheduling, and discussion and lab scheduling are more difficult to organize at scale as they often require coordinating a large number of students.

2.2.1 Office Hours

Drop-in office hours in computer science courses have historically been organized as single-instructor events. The instructor announces their office hours time and place, and students visit office hours to interact with the instructor, often asking questions about assignments, concepts from lecture, as well as topics not directly related to the course such as undergraduate research opportunities and their own personal interests. However, as enrollments have increased, this model has proven increasingly difficult to scale especially as the number of homework questions has grown with the number of students. Questions left unanswered during discussion and lab section are clarified in office hours, leading to more demand for teaching assistance in office hours.

Until recently, this model was also true for teaching assistant office hours. In TA office hours, students mostly ask questions related to their assignments. TA office hours often used the same model with one TA walking around and answering each question. As more students arrive at office hours and the number of students waiting increases, students sign themselves up on a queue to reserve their place in line, ensuring that students who have waited the longest get helped next. While this model works up until about 10 students in office hours, as more students join the queue, there is pressure on the course staff to resolve all of the questions on the queue or risk ending office hours with several disappointed students. Overcrowded office hours are one of the highest-visibility issues for large lecture courses and a common source of student complaints.

Double, triple, and quadruple-staffing office hours has helped to alleviate supply and demand mismatches. Instead of office hours simply being the responsibility of a single TA, multiple TAs staff each office hour to meet student demand. However, one of the side effects of increasing the supply of TAs is that office hours can quickly grow out of control and unmanageable. Students may jump ahead in the queue and get help from multiple different instructors which not only takes teaching resources away from other students, but can also potentially shortcut the student's learning. Locating the next student in the queue is often a mess as instructors shout names across the room and need to navigate a maze of students to reach the next student on the queue.

Software can be used to streamline drop-in office hours by matching TAs to the next unresolved student help request through an online first-come, first-served queue. The Office Hours Queue web app, developed by undergraduate teaching assistants in the EECS Department on top of the OK platform, presents an interface for students and TAs to interact with the queue in much the same workflow as before. Students login and request help through the web app, specifying the assignment, question, location, and a brief description of the help request. TAs access the web app on their smartphones to view students on the queue. When a TA chooses to help the next student, the system marks the request as currently being helped, displays the name of the student to the TAs, and sends a notification to the student's device. The TA then works with the student to help resolve their question and make progress on their assignment. Finally, the TA marks the help request as resolved, or, if the help request was not resolved, returns the help request back to its original place on the queue. Similar systems have been deployed at peer institutions [21, 43]. An introductory computer science student commented that,

> The additional office hour staff and expanded hours during [the final project] was great! I remember having to wait 2–3 hours and not being able to receive help for projects and assignments at the beginning of the school year because office hours were so packed and there wasn't really an orderly way to identify students who needed help.

Other institutions have taken this model even further by moving office hours into highly-frequented, open spaces on campus for the convenience of students and course staff [21]. In spite of these improvements, office hours remains oversubscribed, and finding spaces large enough and flexible enough for office hours remains a challenge.

This model for drop-in office hours treats student help requests as discrete, individual tickets. However, other models for office hours, such as The Tao of TALC, can also be highly productive. Instead of directly assisting students, The Tao of TALC encourages instructors to be guides wherein "students drive as much as possible" and the instructor acts "more as a facilitator between the confused student and their 'peer instructors'" [4]. This has been used successfully both by instructors of small-group office hours and by TAs running large-group office hours. This group learning model has informed one of the features of the office hours queue which enables a single instructor to help a group of students working on the same question all at once.

The use of online office hours queues has generated a large amount of data on office hours usage patterns [43]. Data dashboards for the office hours queue give instructors a view of which assignments students are asking questions about and when. Using this information, course staff have been able to shift resources and staff the most heavily-impacted office hours with more TAs where they were needed the most, further reducing student wait times. We have found that the majority of office hours are utilized by a very small minority of students: in a class of over 1,400 students, 50 students asked half of all the questions in office hours that semester.

2.2.2 Online Course Delivery

Course delivery is the process of offering a course to students. Office hours, lab, and discussion section are but a few ways students learn in large lecture courses. CS courses at UC Berkeley emphasize solving problems which can occur through lecture, office hours, lab, and discussion section, but is often further emphasized in homework assignments and programming projects which combine multiple concepts presented in the course. As enrollments for several lower-division and upper-division CS courses regularly exceed the capacity of the largest lecture halls on campus, lectures in most CS courses are now webcasted. Making attendance at lecture, lab, and discussion section optional has had profound impacts on the way large lecture courses are run and has required special attention from the instructor.

In webcasted courses, it is much easier for students to fall behind and out-of-sync with content from lecture. Course policies need to be designed such that webcasted courses demand enough attention and consistent effort from students so that they stay on track. The assignment of frequent, small quizzes checks that students are keeping up with lecture. At UC Berkeley, simulcasts are not offered so live lecture recordings are often delayed by at least two hours. Nonetheless, even with interventions to keep students at pace with the course, it is often the case that many students will be at least a few days behind schedule at various times during the semester due to commitments from other courses taking priority.

In order to keep asynchronous classes up-to-speed, most large CS courses at UC Berkeley deliver their content via course websites or online discussion forums such as Piazza. The largest introductory CS courses update the front page of their website frequently with announcements so that students treat it as the definitive authority for the course, reducing student questions about assignments, deadlines, and exams. All materials in the course, including lecture videos, lecture notes, assignments, and readings, are always posted online which enables students to choose how they would like to learn: either with a long-distance learning model, or by attending activities in a more traditional classroom setting.

Experienced TAs have also devised workarounds to make Piazza more robust for large courses.

> Piazza will routinely be flooded with questions, especially near exams. Managing Piazza effectively does not necessarily mean keeping up with this flood of questions; it means directing the flow of questions efficiently so that students with similar questions can easily find previous students' questions and avoid asking repeat questions.
>
> Each week, keep a pinned Piazza thread for each separate homework question. This is the most important rule to follow, as homework questions will comprise the majority of all Piazza questions. Also, we recommend creating separate threads for each lecture note and discussion. Once these threads are in place, aggressively mark student questions as duplicates and move them to the appropriate threads so that all similar questions are in one place.
>
> To help students find these various threads, create a pinned master index which contains links to all of the above threads, as well as threads containing homework grade distributions, TA resources, weekly posts, events (guerrilla sections, exam review sessions, etc.), and other announcements. The index requires maintenance, so either consciously assign the management of the index as a TA duty, or make it understood within the team that each person who posts new material on the Piazza is also responsible for adding a link in the index.

Automation has also benefited online course delivery by streamlining administrative processes. In order to facilitate rapid updates, course websites are deployed using static website generators such as Jekyll for easy updating by course staff. Small programming questions and math problems are checked into a version-controlled question bank consisting of every question ever developed in the course so that developing a homework assignment can be as simple as specifying which questions to include and running a makefile to deploy the assignment to the website. TAs have even developed scripts to automatically extract screenshots and compose Piazza threads for each question. GitHub Classroom allows instructors to easily provision private GitHub repositories for student work and automatically configure instructor permissions. Some courses which require more advanced configuration have developed their own custom web scripts for provisioning private repositories. By automating these tasks, more time can be spent supporting student learning.

2.2.3 Exam Administration

Administering exams at scale is a particular pain point for large courses. One of the reasons for this is the number of students who need to be organized together, on-campus, at one time. In the largest introductory CS courses, it is common for exams to be simultaneously proctored across all of the largest lecture halls on campus, as well as spread across a dozen smaller classrooms. Traditionally, the course staff assigns students to exam rooms based on name or student identification number ranges. While, in theory, this system should allow the staff to distribute students however they like, large courses are hampered by the number of students requesting exam accommodations. In the largest introductory CS courses, there can be as many as 100 students requesting accommodations which makes coordinating exam seating a challenge of its own.

Undergraduate teaching assistants in the EECS Department developed the Seating web app to solve this problem. The exam seating tool allows instructors to design seating charts, populate them with students based on their individual preferences, and send personalized emails to students with their particular exam room and exact seat location. Seating charts can be specified to assign students to every-other seat, or to skip entire rows of seats to make it easier for TAs to answer questions from students in the middle of a long row in a packed lecture hall.

Adopting this software has enabled large courses to provide better accommodations to all students, both students with and without documented disabilities. Students who are left-handed, or those who simply prefer to sit on a particular side of the room, near the aisle, or near the front can mark their preferences in a form. The Seating app will then take those preferences into account when randomly assigning a seat to the student. Since seats are assigned, it is easy to account for missing students and produces a paper trail making it more difficult for students to cheat on exams. Before, during, and after the exam, instructors can lookup any student on the seating chart and immediately identify adjacent students. When used with Gradescope, the Seating app makes it easier to compare suspicious student work against other students sitting in their vicinity.

Peer institutions have developed specialized computer-based testing facilities [28, 52] and restricted computing environments for test-taking [32] which have demonstrated benefits for student learning while reducing the amount of course staff resources spent administering exams.

3 Support

First popularized in the 1980s [38], undergraduate students have been increasingly utilized in teaching positions at institutions of all sizes [10, 12, 34, 37, 42, 45]. In the EECS Department at UC Berkeley, undergraduate teaching assistants (TAs) have been utilized since before 1987.

> Undergraduates have the potential to provide significant help and reduce the workload for graduate TAs and for faculty and other instructors. Undergraduates are typically paid per hour of effort, which can cost significantly less than graduate TAs. In addition, the shared experience of undergraduates may make them particularly attuned to understanding the problems and the challenges facing their peers around specific content. Furthermore, from a pedagogical perspective, peer teaching and evaluation can be valuable learning experiences in and of themselves, and can help empower students and build their confidence with the material. Finally, the undergraduate pool is larger than those of graduate students or faculty, and typically more diverse, presenting an opportunity for a more diverse set of instructors, which could contribute to a more inclusive culture. [26]

One of the defining features of the undergraduate teaching program at UC Berkeley is the culture of student-directed innovation. Tools such as Gradescope and OK were developed by teaching assistants to solve grading and feedback challenges. Undergraduates have different opportunities to engage with the teaching community and receive feedback on their teaching. Situated within this existing infrastructure, the role of the instructor is part role model and part leader with the goal of fostering a productive undergraduate teaching culture.

Beyond course staff, a constellation of extracurricular opportunities for students has grown over the past few years. Dedicated staff have developed several services for the CS student community such as the CS Scholars program in which cohorts of 30 students take classes together for three semesters. With rising enrollments, more students than ever before are involved in EE and CS student organizations. The programs developed by staff and students for recruiting and retaining women in CS have been recognized by the National Center for Women & Information Technology.[1]

[1] https://www.ncwit.org/2019-ncwit-extension-services-transformation-next-award-recipients

3.1 Undergraduate Teaching Assistants

In the literature, it is common for the term Undergraduate Teaching Assistant to describe an undergraduate student teaching in a non-traditional role such as providing assistance to students in drop-in office hours or offering optional small-group or one-on-one tutoring. Notably, this definition refers to students who have a set of responsibilities which are distinct from graduate TAs. At UC Berkeley, undergraduate students hired as teaching assistants are responsible for the same set of duties as their graduate student counterparts: they teach weekly lab and discussion sections, grade assignments and exams, and assist with course delivery. These undergraduate students are officially hired under the title Graduate Student Instructor (GSI), though they are sometimes referred to as Undergraduate Student Instructors (UGSIs) to more accurately describe their academic standing.

EECS PhD students have a teaching requirement of 30 hours of service as a GSI, including 20 hours of service in an undergraduate course. However, the demand for TAs far exceeds the supply of graduate students. This is exacerbated by the fact that the vast majority of EECS PhD students are supported by research grants or fellowships that enable them to focus on their research. The hiring situation is particularly impacted in lower-division CS courses as graduate students often prefer to teach courses in their specific research area and taught by their faculty research advisors. As such, most introductory CS courses at UC Berkeley are taught with primarily undergraduate teaching assistants despite the fact that graduate TAs receive priority for these positions.

As the number of declared CS majors has increased, the demand for TAs in upper-division courses has also exploded beyond the supply of graduate TAs. Compared to peer institutions, the design of the undergraduate CS program makes it easier for the EECS Department to identify qualified undergraduates to staff upper-division courses and meet demand in spite of a shortage of interested graduate students. CS upper-division coursework is relatively flat with short prerequisite chains. Courses such as Introduction to Artificial Intelligence, Operating Systems, or Computer Security do not require any courses beyond the introductory course sequence. Students often satisfy the core introductory courses within two or three semesters so that, by the end of their second year, many students will have taken a couple upper-division CS courses that they can then teach over the remaining two years of their undergraduate degree program. Additionally, students are not required to complete a capstone project, leaving them more time to commit to coursework or extracurricular activities such as teaching or research.

In 2011, the largest introductory CS courses at UC Berkeley would hire 10 teaching assistants to serve classes with enrollments of about 350 students. Typical TA duties included both teaching and administrative responsibilities, usually split evenly across the entire course staff.

Teaching Leading lab and discussion sections each week; holding weekly office hours; advising students; preparing for these teaching activities; and participating in weekly staff meeting.

Administrative Developing handouts, lab exercises, homeworks, and projects; grading assignments; handling accommodations for exceptional circumstances; managing announcements and student questions on the course forum; and proctoring and grading exams.

In recent years, however, grading and feedback tools such as Gradescope and OK have automated or streamlined many grading tasks. Tools have been developed to simplify traditionally expensive processes such as exam administration and assignment extensions. Furthermore, as course enrollments increase, the workload for certain aspects of course administration remain fixed. For example, the number of assignments and exams in the course is generally independent of the number of students enrolled in the course. In contrast, the teaching load grows linearly with respect to the number of students in the course.

Course staff composition has changed to reflect this new context. In a class of over 600 students with more than 20 TAs, there might be only a handful of 5–10 head TAs who are responsible for all of the course's administrative tasks. Instead of splitting tasks evenly, these 5–10 head TAs are each assigned one or two administrative responsibilities in addition to their regular teaching duties. There may be one or two TAs responsible for developing handouts, lab exercises, homeworks, and projects, which allows them to become domain experts in developing assignments for the course and maintaining a high quality of assignments with fewer bugs and greater consistency. This shift allows the remaining TAs to focus on teaching their students as effectively as possible. The number of these teaching-focused TA positions can be scaled at the same rate as course enrollment without significantly affecting course administration activities. Managing this greater number of teaching-focused TAs has become an administrative responsibility in and of itself so there may also be a head TA whose duty is to manage the course staff, communicate expectations, announce upcoming activities, and improve the quality of teaching.

3.2 Center for Student Affairs

The EECS Department served over 27,000 student enrollments across all course offerings during the 2018 academic year. Managing this number of students presents an administrative challenge for operating the program at scale, and can easily create feelings of anonymity among students, harming recruitment and retention efforts. The Center for Student Affairs (CSA), an EECS staff unit that provides several functions for undergraduate and graduate CS education, has developed a number of programs and solutions to tackle these challenges.

Starting Fall 2013, the CS Scholars Program,[2] based on student retention theories of first year college students [47, 48] and minority engineering students [50], is one such solution.

> CS Scholars is a first-year student support program intended to serve those from under-represented communities who have had little or no exposure to Computer Science. A learning community, CS Scholars integrates several components of support to meet the academic, social, and developmental needs of students intending to study Computer Science. Those components include:
>
> - Cohort-style course discussions
> - CS Scholars only seminars for personal and professional development
> - Solidarity and community building activities
> - Dedicated CS Scholars Advising

Data analysis has shown that the CS Scholars cohorts outperform students in the general population by 10–20%, and students maintain a higher GPA than the overall class. In earlier cohorts, among students who identify as having no prior programming experience, CS Scholars had a 0.3 GPA advantage over non-scholars, and a greater difference for students who self-identified as female. The CSA-led EECS Resiliency Project is another retention initiative, which draws attention to stories from students and faculty who struggled with computer science at some point in their lives but persevered through those experiences of failure.

To diversify participation in and access to research experiences and graduate work in computer science, the CSA developed the Summer Undergraduate Program in Engineering Research at Berkeley (SUPERB),[3] an NSF-funded Research Experience for Undergraduates (REU) program. Participants include junior and senior undergraduate students at Berkeley or elsewhere, and each participant receives faculty mentorship, graduate student support, and graduate school advising. 95% of the students who participated in SUPERB continued to graduate school in STEM fields [2].

Due to the large number of students in the EECS major or considering declaring the LSCS major, most undergraduate advising is provided by professional EECS and LSCS major advisors. The advising staff assists with student questions and concerns including those related to the CS degree programs, coursework, undergraduate research, as well as students' broader plans and how they might fit into their life or career goals. The advising staff also manages a team of undergraduate peer advisors.

[2]https://eecs.berkeley.edu/cs-scholars
[3]https://eecs.berkeley.edu/resources/undergrads/research/superb

Getting into CS courses has become an often-cited grievance for undergraduates enrolled in universities, both large and small, across the nation. One of the CSA's functions is to coordinate between faculty and students to ensure that teaching supply is properly calibrated to meet enrollment demands, so the CSA has dedicated staff members for managing course scheduling and enrollment. Course capacity in the EECS Department is primarily limited by availability of classrooms and teaching assistants. Allocation of most discussion classrooms and large lecture halls involves close cooperation with central campus administrative staff. However, campus spaces do not provide enough capacity for all CS courses, so space often needs to be found within EECS-managed buildings. This is complicated by the fact that many spaces are already earmarked for strictly research purposes or strictly academic purposes. The challenge of efficiently allocating the remaining shared spaces is further exacerbated by enrollment growth as research functions compete with rapidly-growing academic functions. Increasing enrollments has also increased the number of teaching assistant hires, which has resulted in a significantly enlarged payroll that, unfortunately, does not increase at a rate sufficient to meet teaching needs, let alone match enrollment trends. Additionally, hiring more TAs requires additional coordination with campus training for first-time GSIs, as well as department-level and course-level preparation (chapter 4).

3.3 Near-Peer Student Mentors

As of 2018, there are 42 student organizations officially registered with the EECS Department, many of which host events and provide services to the broader CS community, such as:

Mentorship Student organizations provide mentorship opportunities by hosting one-on-one or small-group mentoring sessions, blending academic support with a sense of community.

Invited Speakers External speakers and alumni give talks on topics including diversity in tech, overcoming adversity, and well-being, as well as workshops on bias, equity, and inclusion.

Industry Events With a student organization as their sponsor, employers can host info sessions, tech talks, or other events such as puzzle hunts or trivia nights to network with students.

One of the unusual features of the EECS Department is the amount and diversity of student-driven, near-peer mentorship opportunities available to students. In the near-peer mentor model, mentors are only a couple years more senior than their mentees. Near-peer mentoring "provides younger students with a positive, inspiring experience to learning about computing from college near-peer mentors," and "helps students feel like they belong in CS, especially if their mentors have backgrounds or experiences similar to their own" [45].

One such mentorship program is CS Kickstart.[4]

> CS Kickstart is a week-long program open to any incoming UC Berkeley students that introduces them computer science while meeting other computer science students and professionals. This program primarily targets women who are interested in the fields of science, technology, engineering, and math. Participants get hands-on experience in programming introducing them to the creativity and diversity of computer science. Participants also get the opportunity to visit tech companies in the Bay Area to see what life is like for computer scientists in industry. For several years it served 25 incoming students, but recently this doubled. It draws almost all of its support from industry and individual donors. [2]

The 2019 cohort will consist of about 50 participants. The program is organized by a group of undergraduate and graduate women in computer science, and is offered free to participants despite housing, transportation, and activity costs thanks to industry sponsors. As a result of the program, 96 percent of participants felt more prepared to take their first CS course at Berkeley, and 95 percent had a greater motivation to pursue computer science.

Once students are on campus, there are several student organizations forming communities around various identities or affinity groups, many of which offer mentorship programs of their own. Serving the woman-identifying EECS community is the Association of Women in Electrical Engineering and Computer Science (AWE).

> The AWE Mentorship Program provides a framework for EE and CS women to develop and sustain mentoring relationships by matching incoming students with upper division women. As new students, mentees connect with their mentors at the beginning of the school year, receiving personalized academic and social help when needed. Throughout the academic year, mentees receive advice, encouragement, information, and insight from experienced peers. Mentors, in turn, gain satisfaction and knowledge from guiding fellow students while fostering a sense of community.[5]

Similarly, the Society of Women Engineers provides mentorship to the broader community of women in all kinds of Engineering, and the more recent FEMTech student organization engages with the broader campus community by providing outreach and mentorship activities such as FEMTech Launch, which "provides office hours, extra help, and weekly tutoring sessions specifically

[4]https://cs-kickstart.berkeley.edu

[5]https://eecs.berkeley.edu/resources/undergrads/eecs/women/mentoring

geared towards women and underrepresented minorities in lower level CS courses."[6] Honor societies such as Eta Kappa Nu (HKN) and Upsilon Pi Epsilon (UPE) offer free drop-in tutoring to the EECS undergraduate community across a majority of the undergraduate coursework.

Computer Science Mentors (CSM)[7] is a student organization which, like other programs offers academic support together with the community-building benefits of near-peer mentorship, but is offered at large scale, serving nearly 2,000 students per semester across 6 introductory computer science and electrical engineering courses. A typical mentoring group consists of 4–6 students and 1 near-peer mentor. The mentor facilitates student discussions and group work with a focus on mastery learning. Mentors adapt each session to meet the group's needs, drawing on additional examples to clarify concepts and build student confidence. Over the course of the semester, the mentor gets to know each student on an individual basis, and students grow more comfortable with each other too. The development of these relationships makes it easier for the mentor to keep in touch with their students by setting up individual check-ins in addition to the group sessions, sharing their experiences and study advice, and referring students to free tutoring services offered by other members of the EECS community. Participation in CSM small-group mentoring has been shown to have a significant positive association with exam scores. Organizing, preparing, and mentoring the mentors has become a challenge of its own (chapter 4).

[6] `https://femtechberkeley.com/index.php/education/`
[7] `https://csmentors.berkeley.edu`

4 Preparation

Utilizing undergraduates in teaching positions is not without its risks.

> Undergraduates who are unclear on the material may cause confusion among their peers. In addition, not all undergraduates have the knowledge or maturity to successfully teach, assess, or mentor their peers, or understand conflict-of-interest situations. If poorly implemented or not properly supervised, this approach can place additional strain on course instructors. [26]

Preparation is especially important as the program expands in size and hires more undergraduate TAs to support large enrollment courses in both lower-division and upper-division courses.

4.1 Introduction to Teaching Computer Science

> [CS 370: Introduction to Teaching Computer Science] is a course designed help aspiring teachers hone their teaching skills, become a part of the teaching community here at UC Berkeley, and expose them to the foundations of computer science pedagogy. Students in this class will receive first-hand experience through one-on-one tutoring and an enriched teaching knowledge through research-based pedagogical studies.
>
> CS 370 has three key components that distinguish it from other pedagogical courses. First, we cover student interactivity and teaching in one-on-one settings. This is applicable to all levels of teachers [...] since one-on-one interactions are a critical component of all teaching experiences. Next, we cover group teaching through in-class demonstrations, as mastering pacing and understanding the individualities of students in a group setting is key to being a successful TA. Last, we socratically discuss current issues in CS pedagogy, including atmosphere-related questions such as: underrepresentation, stigmas associated with computer science, the issue of prior experience, and how these factors heavily influence student learning.[1]

[1] `http://inst.eecs.berkeley.edu/~cs370/policies.html`

Students are introduced to pedagogical concepts during an 80-minute seminar each week that includes discussion of ideas and reflection on their teaching experiences in small groups. Outside of the classroom, students read scholarly articles on the practice and theory of teaching computer science, host three, hour-long one-on-one tutoring sessions per week, and reflect on their tutoring as part of a weekly written assignment. Combining theory and practice together helps students learn and retain material, treating topics taught in class as a frame for questions brought up in self-reflections on their teaching experiences. Group discussions are facilitated by experienced TAs whose experience students identify with and more closely relate. To facilitate discussion, these weekly seminars are held in an active learning classroom with students seated around tables and facing each other rather than the front of a lecture hall. Between the weekly seminar, tutoring, tutoring preparation, tutoring reflection, and weekly assignments, CS 370 is a total commitment of 9 hours per week.

CS 370 was designed originally as a course to prepare and engage new teachers which influenced its decision to use one-on-one tutoring as the context for teaching practicum. Unlike other programs at peer institutions, a large number of aspiring undergraduate student-teachers take the course before they become TAs. This results in a diversity of students composed of first-year students who just recently took the courses they want to someday teach as well as older, second or third-year students, which makes for engaging conversations as their different experience levels provide greater opportunities to learn from each other. More-experienced student-teachers in the group and the experienced TA facilitators can chime in and provide nuanced viewpoints to questions less-experienced teachers might have about teaching one-on-one or leading small groups.

This design of CS 370 has a number of consequences which has made it particularly well-suited for preparing undergraduate TAs. First, the outline of topics includes concerns which are especially important for teaching at the undergraduate level such as diversity, unconscious bias, and tackling misconceptions. CS 370 is complemented by CS 375 which is geared toward a graduate student audience and, notably, includes coverage of topics such as developing course syllabi, exam problems or rubrics, and student surveys, all of which are tasks that concern head TAs responsible for the administrative component of a course but not necessarily teaching-focused TAs. CS 370, CS 375, as well as other pedagogy courses at peer institutions have also found success running the course in a workshop style with a significant portion of the materials presented at the beginning of the semester to maximize their effect on teaching, and then later fading away to more infrequent check-ins later in the semester [42].

4.2 Mentoring at Scale

Near-peer student mentors, such as the students who lead small-group sessions for CSM, are organized into a family system to prepare for their weekly group sessions and build community. Like TA families, mentors are grouped into families of about 6 mentors, each consisting of two experienced senior mentors and about four less-experienced junior mentors. In addition to providing feedback, checking-in, and bonding over social events, mentors also meet together regularly for one hour each week to prepare for the upcoming week's group sessions with mentees. Family meetings are a mix between active problem-solving, 3-minute teaching demonstrations, real-time critiques, and moments of written self-reflection. In these weekly family meetings, senior mentors lead and facilitate group discussions with junior mentors about the challenges and pitfalls of upcoming concepts, and assist mentors in personalizing their session to meet their mentees' needs. In order to make this hour effective, junior mentors prepare for the family meetings in advance by spending half an hour reviewing concepts in advance and preparing a mental outline of the lesson they have in mind for their session.

Most mentors only lead one or two group mentoring sessions. Since these mentoring sections only consist of 4–6 students each, for larger classes, over 100 sections are offered each week. In order to support this structure, CSM delegates the task of organizing mentors to the course coordinators, highly-experienced mentors who manage the entire operation. Course coordinators play a similar role as Section Leader Coordinators and Meta-TAs implemented at peer institutions [37, 38, 42]. They hold weekly meetings with all of the senior mentors to prepare content for the family meetings and group mentoring sessions and assist the senior mentors in preparing for facilitating their own family meetings.

4.3 Course-Specific Preparation

Large classes make it harder for instructors to provide individual feedback to students. Likewise, large course staffs make it harder for instructors to provide personalized mentorship to their TAs. As course staffs have grown beyond 20, 30, 40, and even 50 TAs, several CS courses have begun grouping their course staff members into smaller families as well. Each TA family consists of 4–6 TAs with a mix of experienced and inexperienced teachers. As part of their preparation duties, TAs are occasionally expected to shadow and provide feedback to other family members to improve their teaching. Mirroring mentoring families, lead TAs check-in with their family members throughout the semester and organize occasional social outings with the entire group, building a community between undergraduate TAs.

In addition to the formal CS 370 pedagogy course and the more informal family system, undergraduate student-teachers also receive support and mentorship at the course level. The instructor of record and more-experienced TAs will often share their preparation materials, refine assignment guides, discussion walkthroughs, and other documents designed to support newer teachers. Discussions handouts are often reused between semesters so course staff share potential ways of teaching the concepts. Assignment guides provide answers to frequently-asked questions, identify common student bugs and their fixes, and suggest relevant connections to previous concepts and prerequisites to bridge knowledge gaps.

It is also common for course staff to run their own preparation sessions at the beginning of each semester to on-board new course staff members, set expectations, and provide course-specific guidance. Topics include preparing for discussion, lab, and office hours; modeling behavior and setting student expectations for the course's pace, format, and recommended learning strategy; course-specific resources and policies that need to be shared with students early in the semester; and upcoming changes for the current offering of the course. The course staff set four ground rules for one-on-one interactions in office hours:

1. If you don't know what to do, ask.
2. Be sensitive because learning computer science can be hard.
3. Let the student drive.
4. Do not give away the answer, if you can help it.

These conversations are continued throughout the semester during weekly staff meetings where the entire course staff meets to make decisions on open administrative questions, give and receive feedback on new ideas or proposals, and plot out the next few weeks' content in the course. At the final meeting, time is set aside for course staff to reflect on the entire semester as a whole and determine where improvements can be made to assignments, teaching, policies, and the overall design of the course.

In addition, experienced TAs propose an idealized assignment help workflow to normalize lab and office hours expectations across the course staff. The goal of this workflow is to reduce the risk of providing too much assistance, which can harm students as they grow dependent on the guidance and are unable to solve problems on their own [43]. For programming courses, this workflow starts with understanding the question since students often miss important details when focused on the problem. The TA is directed to sit down beside the student, ensuring that their eye-lines match, and introduce themselves and learn the student's name. These practices help to build trust and rapport between the student and the TA, particularly if the student and the TA meet again in lab or office hours. The next step is to ask the student to describe their problem in their own words. This gives the TA time to skim the student's code and verify that the student's explanation matches their code, and later work with the student to identify the source and cause of the problem. After the student gains an understanding of the problem, the TA works with the student to formulate a plan to resolve the problem, and then gives the student time to solve it on their own. After about 10 minutes, during which the TA helps another student, the TA returns to check back in on the student's progress. These last few practices give the student space to work on the problem on their own and encourages them to build independence. Rather than sitting with the student and solving their problems for them, the TA's goal is to have the student in a better position to solve the problem independently.

5 Discussion

While automation, support, and preparation have enabled CS courses at UC Berkeley to scale to meet both CS major and non-major student demand, the system of incentives, particularly the LSCS 3.3 GPA cap, strains relationships between instructors and students. Implementing student-friendly course policies, designing collaborative assignments, and encouraging students to take advantage of mentorship opportunities can significantly improve the student learning experience but is ultimately contrary to the message sent by the GPA cap. There is little room for failure: students who struggle in one or two of the introductory courses face an uphill battle to make it to the GPA cap. This is in spite of the limited evidence that, when given a second chance, students are able to make a remarkable improvement. In Spring 2016, students who were given the option to receive a failing grade in introductory Data Structures and retake the course in a later semester made an average improvement of +2.54 grade bins over the grade they would have received in Spring 2016.

The EECS Department has made significant gains in improving the gender diversity of its undergraduate student population, receiving recognition by the National Center for Women & Information Technology (NCWIT)[1] as well as local news media[2] for its achievements. However, there is still much work to be done to encourage participation from a broader population of students. For some students, the time, energy, and stress necessary to meet the GPA cap makes the major unattractive. Other students may not have the confidence to pursue the major despite interest and academic preparation. In order to grow capacity while maintaining an inclusive student culture, it is important to take into account the entire system of incentives and punishments. Policies such as the GPA cap have ripple effects as student perceptions of the program on campus are shaped by its reputation of being highly rigorous, demanding, and stressful. Students may not feel comfortable if they see the program—including its faculty, staff, and students—as competitive in spite of their best efforts to design collaborative and supportive learning experiences. When

[1]https://www.ncwit.org/2019-ncwit-extension-services-transformation-next-award-recipients
[2]https://www.mercurynews.com/2018/04/16/forget-techs-bad-bros-stanford-berkeley-boost-female-computing-grads/

a potential student's sensibilities do not line up with these perceived values, students may feel excluded from CS even if they could otherwise be successful computer scientists.

Even with technology and a large number of highly motivated support staff, faculty teaching load remains a significant burden, particularly for new tenure-track assistant professors who also need to balance their research output and tenure priorities with teaching. In particular, the faculty find larger enrollments have resulted in greater administrative workload, one that has not yet been fully displaced by head TAs despite all of the preparation and software in place to support them. Furthermore, courses which rely on individually-personalized projects or deep-feedback assignments do not easily fit into this framework of automation solutions.

Additionally, the use of some automation has the potential to impact student behavior in unexpected ways. The office hours queue, for example, compartmentalizes assignment help and student questions into individual tickets which are then resolved one-by-one. This model works in introductory courses due to the large number of TA office hours, and because assignments are typically scaffolded to help students make progress and facilitate efficient resolution in office hours. But the kinds of debugging and self-regulation practices acquired through these introductory CS office hours don't necessarily prepare students for upper-division coursework where the questions are more open-ended and the debugging processes much less clear.

Designing a CS program that scales requires cooperation from all levels of campus, including the students, staff, faculty, the department, the college, and the administration. While this report focuses on recent developments, a culture of innovation by undergraduate TAs has long existed in the EECS Department since their introduction in the 1980's, and by graduate students even before then. Each class of students is supported by the preceding class of students who teach section each week and serve as role models. Staff have worked closely with students and faculty to develop novel solutions to challenges of teaching CS at scale while advising triple the number of students from just a decade ago. Faculty teaching undergraduate courses have made sacrifices to teach at this scale, often going far beyond their expected teaching responsibilities. The College of Engineering, Graduate Division, and campus administration have supported the program by committing additional faculty slots, expanding advising support, and helped to expand course enrollments by funding additional TAs.

However, the program still faces a number of budgetary shortfalls particularly in the area of course budgets as campus Temporary Academic Support (instructional support) has not kept up with the unprecedented growth of the program. Contributions from private and industry donors have enabled the department to continue funding more TAs, opening more sections and reaching more students in spite of structural deficits and budget cuts at the university and state level. This external investment has fueled the development of software solutions, processes, and initiatives which have made the UC Berkeley EECS Department a national model for teaching CS at scale.

Bibliography

1. *2018 State of Computer Science Education.* 2018. URL: https://advocacy.code.org/.

2. P. Alivisatos. "STEM and Computer Science Education: Preparing the 21st Century Workforce". Research and Technology Subcommittee, House Committee on Science, Space, and Technology. 2017. URL: https://docs.house.gov/meetings/SY/SY15/20170726/106330/HHRG-115-SY15-Wstate-AlivisatosA-20170726.pdf.

3. C. Alvarado, M. Minnes, and L. Porter. "Micro-Classes: A Structure for Improving Student Experience in Large Classes". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education.* SIGCSE '17. ACM, New York, NY, USA, 2017, pp. 21–26. ISBN: 978-1-4503-4698-6. DOI: 10.1145/3017680.3017727. URL: http://doi.acm.org/10.1145/3017680.3017727.

4. O. Astrachan, N. Parlante, D. D. Garcia, and S. Reges. "Teaching Tips We Wish They'd Told Us Before We Started". In: *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education.* SIGCSE '07. ACM, Covington, Kentucky, USA, 2007, pp. 2–3. ISBN: 1-59593-361-1. DOI: 10.1145/1227310.1227314. URL: http://doi.acm.org/10.1145/1227310.1227314.

5. M. Babes-Vroman, I. Juniewicz, B. Lucarelli, N. Fox, T. Nguyen, A. Tjang, G. Haldeman, A. Mehta, and R. Chokshi. "Exploring Gender Diversity in CS at a Large Public R1 Research University". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education.* SIGCSE '17. ACM, New York, NY, USA, 2017, pp. 51–56. ISBN: 978-1-4503-4698-6. DOI: 10.1145/3017680.3017773. URL: http://doi.acm.org/10.1145/3017680.3017773.

6. S. Basu, A. Wu, B. Hou, and J. DeNero. "Problems Before Solutions: Automated Problem Clarification at Scale". In: *Proceedings of the Second (2015) ACM Conference on Learning @ Scale.* L@S '15. ACM, Vancouver, BC, Canada, 2015, pp. 205–213. ISBN: 978-1-4503-3411-2. DOI: 10.1145/2724660.2724679. URL: http://doi.acm.org/10.1145/2724660.2724679.

7. J. M. Cohoon. "Recruiting and Retaining Women in Undergraduate Computing Majors". In: *SIGCSE Bull.* 34:2, 2002, pp. 48–52. ISSN: 0097-8418. DOI: 10.1145/543812.543829. URL: http://doi.acm.org/10.1145/543812.543829.

8. Computing Research Association. *Generation CS: Computer Science Undergraduate Enrollments Surge Since 2006.* 2017. URL: https://cra.org/data/Generation-CS/.

9. K. K. Curtis. *Computer Manpower — Is There a Crisis?* Washington, D.C., USA: National Science Foundation, 1982. URL: https://cs.stanford.edu/people/eroberts/Curtis-ComputerManpower/.

10. A. Decker, P. Ventura, and C. Egert. “Through the Looking Glass: Reflections on Using Undergraduate Teaching Assistants in CS1”. In: *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’06. ACM, New York, NY, USA, 2006, pp. 46–50. ISBN: 1-59593-259-3. DOI: 10.1145/1121341.1121358. URL: http://doi.acm.org/10.1145/1121341.1121358.

11. L. A. Delyser, J. Goode, M. Guzdial, Y. Kafai, and A. Yadav. *Priming the Computer Science Teacher Pump: Integrating Computer Science Education into Schools of Education*. 2018. URL: http://www.computingteacher.org/2018.

12. P. E. Dickson, T. Dragon, and A. Lee. “Using Undergraduate Teaching Assistants in Small Classes”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’17. ACM, New York, NY, USA, 2017, pp. 165–170. ISBN: 978-1-4503-4698-6. DOI: 10.1145/3017680.3017725. URL: http://doi.acm.org/10.1145/3017680.3017725.

13. E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller. “OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale”. In: *ACM Trans. Comput.-Hum. Interact.* 22:2, 2015, 7:1–7:35. ISSN: 1073-0516. DOI: 10.1145/2699751. URL: http://doi.acm.org/10.1145/2699751.

14. Google Inc. and Gallup Inc. *Diversity Gaps in Computer Science: Exploring the Underrepresentation of Girls, Blacks and Hispanics*. 2016. URL: http://goo.gl/PG34aH.

15. P. J. Guo. “Online Python Tutor: Embeddable Web-based Program Visualization for CS Education”. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE ’13. ACM, New York, NY, USA, 2013, pp. 579–584. ISBN: 978-1-4503-1868-6. DOI: 10.1145/2445196.2445368. URL: http://doi.acm.org/10.1145/2445196.2445368.

16. J. Hug and D. D. Garcia. “Handling Very Large Lecture Courses: Keeping the Wheels on the Bus (Abstract Only)”. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE ’15. ACM, New York, NY, USA, 2015, pp. 697–697. ISBN: 978-1-4503-2966-8. DOI: 10.1145/2676723.2691867. URL: http://doi.acm.org/10.1145/2676723.2691867.

17. J. Hug and C. Lee. “Handling Very Large Lecture Courses: Keeping the Wheels on the Bus III (Abstract Only)”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’17. ACM, New York, NY, USA, 2017, pp. 725–725. ISBN: 978-1-4503-4698-6. DOI: 10.1145/3017680.3022374. URL: http://doi.acm.org/10.1145/3017680.3022374.

18. D. G. Kay. “Large Introductory Computer Science Classes: Strategies for Effective Course Management”. In: *Proceedings of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’98. ACM, New York, NY, USA, 1998, pp. 131–134. ISBN: 0-89791-994-7. DOI: 10.1145/273133.273177. URL: http://doi.acm.org/10.1145/273133.273177.

19. C. M. Lewis. “ACM Retention Committee: Twelve Tips for Creating a Culture That Supports All Students in Computing”. In: *ACM Inroads* 8:4, 2017, pp. 17–20. ISSN: 2153-2184. DOI: 10.1145/3148524. URL: http://doi.acm.org/10.1145/3148524.

20. S. N. Liao, D. Zingaro, C. Alvarado, W. G. Griswold, and L. Porter. “Exploring the Value of Different Data Sources for Predicting Student Performance in Multiple CS Courses”. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE ’19. ACM, Minneapolis, MN, USA, 2019, pp. 112–118. ISBN: 978-1-4503-5890-3. DOI: 10.1145/3287324.3287407. URL: http://doi.acm.org/10.1145/3287324.3287407.

21. T. MacWilliam and D. J. Malan. “Scaling Office Hours: Managing Live Q&#38;A in Large Courses”. In: *J. Comput. Sci. Coll.* 28:3, 2013, pp. 94–101. ISSN: 1937-4771. URL: http://dl.acm.org/citation.cfm?id=2400161.2400179.

22. M. L. Maher, C. Latulipe, H. Lipford, and A. Rorrer. “Flipped Classroom Strategies for CS Education”. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE ’15. ACM, New York, NY, USA, 2015, pp. 218–223. ISBN: 978-1-4503-2966-8. DOI: 10.1145/2676723.2677252. URL: http://doi.acm.org/10.1145/2676723.2677252.

23. D. J. Malan. “Reinventing CS50”. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. SIGCSE ’10. ACM, New York, NY, USA, 2010, pp. 152–156. ISBN: 978-1-4503-0006-3. DOI: 10.1145/1734263.1734316. URL: http://doi.acm.org/10.1145/1734263.1734316.

24. M. Minnes, C. Alvarado, and L. Porter. “Lightweight Techniques to Support Students in Large Classes”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE ’18. ACM, New York, NY, USA, 2018, pp. 122–127. ISBN: 978-1-4503-5103-4. DOI: 10.1145/3159450.3159601. URL: http://doi.acm.org/10.1145/3159450.3159601.

25. S. Narayanan, K. Cunningham, S. Arteaga, W. J. Welch, L. Maxwell, Z. Chawinga, and B. Su. “Upward Mobility for Underrepresented Students: A Model for a Cohort-based Bachelor’s Degree in Computer Science”. In: *ACM Inroads* 9:2, 2018, pp. 72–78. ISSN: 2153-2184. DOI: 10.1145/3210555. URL: http://doi.acm.org/10.1145/3210555.

26. National Academies of Sciences, Engineering, and Medicine. *Assessing and Responding to the Growth of Computer Science Undergraduate Enrollments*. The National Academies Press, Washington, DC, 2018. ISBN: 978-0-309-46702-5. DOI: 10.17226/24926. URL: https://www.nap.edu/catalog/24926/assessing-and-responding-to-the-growth-of-computer-science-undergraduate-enrollments.

27. T. Newhall, L. Meeden, A. Danner, A. Soni, F. Ruiz, and R. Wicentowski. “A Support Program for Introductory CS Courses That Improves Student Performance and Retains Students from Underrepresented Groups”. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE ’14. ACM, New York, NY, USA, 2014, pp. 433–438. ISBN: 978-1-4503-2605-6. DOI: 10.1145/2538862.2538923. URL: http://doi.acm.org/10.1145/2538862.2538923.

28. T. Nip, E. L. Gunter, G. L. Herman, J. W. Morphew, and M. West. “Using a Computer-based Testing Facility to Improve Student Learning in a Programming Languages and Compilers Course”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE ’18. ACM, Baltimore,

Maryland, USA, 2018, pp. 568–573. ISBN: 978-1-4503-5103-4. DOI: 10.1145/3159450.3159500. URL: http://doi.acm.org/10.1145/3159450.3159500.

29. E. Patitsas, M. Craig, and S. Easterbrook. "A Historical Examination of the Social Factors Affecting Female Participation in Computing". In: *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education.* ITiCSE '14. ACM, Uppsala, Sweden, 2014, pp. 111–116. ISBN: 978-1-4503-2833-3. DOI: 10.1145/2591708.2591731. URL: http://doi.acm.org/10.1145/2591708.2591731.

30. E. Patitsas, M. Craig, and S. Easterbrook. "How CS departments are managing the enrolment boom: Troubling implications for diversity". In: *2016 Research on Equity and Sustained Participation in Engineering, Computing, and Technology (RESPECT).* 2016, pp. 1–2. DOI: 10.1109/RESPECT.2016.7836180.

31. P. M. Phothilimthana and S. Sridhara. "High-Coverage Hint Generation for Massive Courses: Do Automated Hints Help CS1 Students?" In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education.* ITiCSE '17. ACM, Bologna, Italy, 2017, pp. 182–187. ISBN: 978-1-4503-4704-4. DOI: 10.1145/3059009.3059058. URL: http://doi.acm.org/10.1145/3059009.3059058.

32. C. Piech and C. Gregg. "BlueBook: A Computerized Replacement for Paper Tests in Computer Science". In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education.* SIGCSE '18. ACM, Baltimore, Maryland, USA, 2018, pp. 562–567. ISBN: 978-1-4503-5103-4. DOI: 10.1145/3159450.3159587. URL: http://doi.acm.org/10.1145/3159450.3159587.

33. C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. "Modeling How Students Learn to Program". In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education.* SIGCSE '12. ACM, Raleigh, North Carolina, USA, 2012, pp. 153–160. ISBN: 978-1-4503-1098-7. DOI: 10.1145/2157136.2157182. URL: http://doi.acm.org/10.1145/2157136.2157182.

34. H. Pon-Barry, A. St. John, B. W.-L. Packard, and B. Rotundo. "A Flexible Curriculum for Promoting Inclusion Through Peer Mentorship". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* SIGCSE '19. ACM, Minneapolis, MN, USA, 2019, pp. 1116–1122. ISBN: 978-1-4503-5890-3. DOI: 10.1145/3287324.3287434. URL: http://doi.acm.org/10.1145/3287324.3287434.

35. L. Porter, C. Bailey Lee, and B. Simon. "Halving Fail Rates Using Peer Instruction: A Study of Four Computer Science Courses". In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education.* SIGCSE '13. ACM, New York, NY, USA, 2013, pp. 177–182. ISBN: 978-1-4503-1868-6. DOI: 10.1145/2445196.2445250. URL: http://doi.acm.org/10.1145/2445196.2445250.

36. J. Prather, R. Pettit, B. A. Becker, P. Denny, D. Loksa, A. Peters, Z. Albrecht, and K. Masci. "First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* SIGCSE '19. ACM, Minneapolis, MN, USA, 2019, pp. 531–537. ISBN: 978-1-4503-5890-3. DOI: 10.1145/3287324.3287374. URL: http://doi.acm.org/10.1145/3287324.3287374.

37. S. Reges. "Using Undergraduates As Teaching Assistants at a State University". In: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education.* SIGCSE '03. ACM, New York, NY, USA, 2003, pp. 103–107. ISBN: 1-58113-648-X. DOI: 10.1145/611892.611943. URL: http://doi.acm.org/10.1145/611892.611943.

38. S. Reges, J. McGrory, and J. Smith. "The Effective Use of Undergraduates to Staff Large Introductory CS Courses". In: *Proceedings of the Nineteenth SIGCSE Technical Symposium on Computer Science Education.* SIGCSE '88. ACM, Atlanta, Georgia, USA, 1988, pp. 22–25. ISBN: 0-89791-256-X. DOI: 10.1145/52964.52971. URL: http://doi.acm.org/10.1145/52964.52971.

39. P. Rheingans, E. D'Eramo, C. Diaz-Espinoza, and D. Ireland. "A Model for Increasing Gender Diversity in Technology". In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education.* SIGCSE '18. ACM, New York, NY, USA, 2018, pp. 459–464. ISBN: 978-1-4503-5103-4. DOI: 10.1145/3159450.3159533. URL: http://doi.acm.org/10.1145/3159450.3159533.

40. E. Roberts. *A History of Capacity Challenges in Computer Science.* 2016. URL: https://cs.stanford.edu/people/eroberts/CSCapacity/ (visited on 04/01/2019).

41. E. Roberts. *Resources for the CS Capacity Crisis.* 2018. URL: https://cs.stanford.edu/people/eroberts/ResourcesForTheCSCapacityCrisis/ (visited on 04/01/2019).

42. E. Roberts, J. Lilly, and B. Rollins. "Using Undergraduates As Teaching Assistants in Introductory Programming Courses: An Update on the Stanford Experience". In: *Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education.* SIGCSE '95. ACM, New York, NY, USA, 1995, pp. 48–52. ISBN: 0-89791-693-X. DOI: 10.1145/199688.199716. URL: http://doi.acm.org/10.1145/199688.199716.

43. A. J. Smith, K. E. Boyer, J. Forbes, S. Heckman, and K. Mayer-Patel. "My Digital Hand: A Tool for Scaling Up One-to-One Peer Teaching in Support of Computer Science Learning". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education.* SIGCSE '17. ACM, Seattle, Washington, USA, 2017, pp. 549–554. ISBN: 978-1-4503-4698-6. DOI: 10.1145/3017680.3017800. URL: http://doi.acm.org/10.1145/3017680.3017800.

44. S. Sridhara, B. Hou, J. Lu, and J. DeNero. "Fuzz Testing Projects in Massive Courses". In: *Proceedings of the Third (2016) ACM Conference on Learning @ Scale.* L@S '16. ACM, Edinburgh, Scotland, UK, 2016, pp. 361–367. ISBN: 978-1-4503-3726-7. DOI: 10.1145/2876034.2876050. URL: http://doi.acm.org/10.1145/2876034.2876050.

45. C. Stephenson, A. Derbenwick Miller, C. Alvarado, L. Barker, V. Barr, T. Camp, C. Frieze, C. Lewis, E. Cannon Mindell, L. Limbird, D. Richardson, M. Sahami, E. Villa, H. Walker, and S. Zweben. *Retention in Computer Science Undergraduate Programs in the U.S.: Data Challenges and Promising Interventions.* New York, NY, USA, 2018. URL: https://www.acm.org/binaries/content/assets/education/retention-in-cs-undergrad-programs-in-the-us.pdf.

46. V. Swamy. *Pedagogy, Infrastructure, and Analytics for Data Science Education at Scale.* Technical report UCB/EECS-2018-81. EECS Department, University of California, Berkeley, 2018. URL: `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-81.html`.

47. P. T. Terenzini and E. T. Pascarella. "Toward the Validation of Tinto's Model of College Student Attrition: A Review of Recent Studies". In: *Research in Higher Education* 12:3, 1980, pp. 271–282. ISSN: 03610365, 1573188X. URL: `http://www.jstor.org/stable/40195370`.

48. V. Tinto. *Leaving college: Rethinking the causes and cures of student attrition.* ERIC, 1987.

49. N. Titterton, C. M. Lewis, and M. J. Clancy. "Experiences with lab-centric instruction". In: *Computer Science Education* 20:2, 2010, pp. 79–102. DOI: `10.1080/08993408.2010.486256`. eprint: `https://doi.org/10.1080/08993408.2010.486256`. URL: `https://doi.org/10.1080/08993408.2010.486256`.

50. U. Treisman. "Studying Students Studying Calculus: A Look at the Lives of Minority Mathematics Students in College". In: *The College Mathematics Journal* 23:5, 1992, pp. 362–372. ISSN: 07468342, 19311346. URL: `http://www.jstor.org/stable/2686410`.

51. L. Wang, A. Sy, L. Liu, and C. Piech. "Deep Knowledge Tracing On Programming Exercises". In: *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale.* L@S '17. ACM, Cambridge, Massachusetts, USA, 2017, pp. 201–204. ISBN: 978-1-4503-4450-0. DOI: `10.1145/3051457.3053985`. URL: `http://doi.acm.org/10.1145/3051457.3053985`.

52. M. West and C. Zilles. "Modeling Student Scheduling Preferences in a Computer-Based Testing Facility". In: *Proceedings of the Third (2016) ACM Conference on Learning @ Scale.* L@S '16. ACM, Edinburgh, Scotland, UK, 2016, pp. 309–312. ISBN: 978-1-4503-3726-7. DOI: `10.1145/2876034.2893441`. URL: `http://doi.acm.org/10.1145/2876034.2893441`.

53. L. Yan, A. Hu, and C. Piech. "Pensieve: Feedback on Coding Process for Novices". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* SIGCSE '19. ACM, Minneapolis, MN, USA, 2019, pp. 253–259. ISBN: 978-1-4503-5890-3. DOI: `10.1145/3287324.3287483`. URL: `http://doi.acm.org/10.1145/3287324.3287483`.

54. L. Yan, N. McKeown, M. Sahami, and C. Piech. "TMOSS: Using Intermediate Assignment Work to Understand Excessive Collaboration in Large Classes". In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education.* SIGCSE '18. ACM, Baltimore, Maryland, USA, 2018, pp. 110–115. ISBN: 978-1-4503-5103-4. DOI: `10.1145/3159450.3159490`. URL: `http://doi.acm.org/10.1145/3159450.3159490`.

55. S. Zweben and B. Bizot. *2018 CRA Taulbee Survey.* 2019. URL: `https://cra.org/resources/taulbee-survey/`.