Accountability in an Era of Agentic GenAI Coding: Rethinking the Early University Computing Experience

Kathi Fisler
kfisler@brown.edu
Dept of Computer Science
Brown University
Providence, RI, USA

Shriram Krishnamurthi
shriram@brown.edu
Dept of Computer Science
Brown University
Providence, RI, USA

Michael Littman
mlittman@brown.edu
Dept of Computer Science
Brown University
Providence, RI, USA

Abstract

GenAI coding agents now enable people to create software artifacts that are significantly beyond their own manual coding skills. This becomes a cause for significant concern in artifacts with real-world impacts. As educators, we must help students who leverage agents for exploration and productivity to learn to be accountable for their work. What does this look like?

This paper describes our experience from an agentic coding course aimed at 2nd-4th semester university students with at least one semester of programming. We gave students projects that were just at or beyond what we expected they could have written on their own. Students were expected to use agents heavily, but also to defend the quality of what they produced. After each project stage, we taught them software-engineering practices that could have been useful and expected them to use these on subsequent assignments. Students kept reflection journals throughout each project. Students clustered into four broad strategies for taking accountability for their work. We describe these and other observations about teaching early-stage university CS students to program collaboratively with agents.

CCS Concepts

• **Social and professional topics → Computer science education**.

Keywords

agentic coding, developing student accountability, course design

ACM Reference Format:
Kathi Fisler, Shriram Krishnamurthi, and Michael Littman. 2027. Accountability in an Era of Agentic GenAI Coding: Rethinking the Early University Computing Experience. In *Proceedings of the 58th ACM Technical Symposium on Computer Science Education V.1 (SIGCSE TS 2027), February 17–20, 2027, Sacramento, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3857222.3860178

1 Introduction

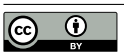
With the rise of agentic programming, CS educators are increasingly calling for students to learn how to be accountable for their code. This involves proposals to teach testing of agent-produced code in CS1 [24], teach reasoning about code [18], and to have students engage in code reviews early on [3]. These suggest that techniques typically taught in intermediate or advanced software-engineering courses are now also relevant for novices. We couldn't agree more.

What does "accountability" look like in lower-level university computing courses and how do we teach it? We can ask students to write test suites, for example, but modern coding agents can do that, just as they write code. As we rethink our course designs to account for the ways our students are using coding agents (whether or not we "allow" it), we need narratives and pedagogies that motivate and teach first- and second-year students how to meaningfully stay in the loop while building software.

In Spring 2026, we ran an experimental course [8] on coding with agents for students who had between one and three prior semesters of CS coursework (generally, 2nd-4th semester students). At the outset, we had three goals: (1) to see how students would engage with agents while building increasingly difficult programs; (2) to see which software-engineering practices the students would adopt in light of their experiences; (3) to identify our (instructors') own expert blind spots [21] on what the students could build with agents while staying in the loop. Students kept reflection journals throughout the semester. This experience report describes our course design and our takeaways about what it means to teach students to be accountable for products they build with agents.

What is "Agentic GenAI Coding"? We distinguish "agentic" coding with generative AI (GenAI) from "prompting". To us, prompting is an activity where the iterative loop of improving and fixing software is done by a human: they write a prompt, get a program, deal with any parsing or language errors, run it, evaluate it, revise the request to an LLM, and so on. In contrast, *agentic* coding is driven by a program, which takes human prompts and some notion of "fitness" (which can be a suite of tools of increasing complexity from parsers to type checkers to test suites to more sophisticated checkers), but then autonomously generates code, modifies it, tries to run it, checks it for fitness, takes the feedback, and iterates internally. This leads to a significant improvement in quality, because LLMs on their own are poor at checking their own work and need an external arbiter [27]. In contrast, the agent keeps iterating until it satisfies the checkers (or times out). The verifier remains the weak point, however. Because agents "reward-hack"—satisfying the checker without doing the task, even tampering with the evaluation harness that grades them [29]—incomplete or under-specified verification leads to the "specification gaming" long anticipated in Reinforcement Learning [15].

This paper uses the term *agentic coding* to refer to producing programs with an agentic system that runs the outer loop. We use the term *coding agent* or *agent* for the corresponding agentic



This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGCSE TS 2027, Sacramento, CA, USA
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3075-7/2027/02
https://doi.org/10.1145/3857222.3860178

system. Having students work with coding agents raises the stakes for instructors. When students merely prompt an LLM for code, instructors can ask students to write the tests. Even if students prompt an LLM for those tests, they are still in the loop. Coding agents can take them out of the loop entirely, especially on problems suitable for early CS courses. Instructors thus have to motivate them to step back in, perhaps by engaging them in accountability.

2 Related Work

Calls for first-year students to learn software-reliability techniques are not new to the agentic coding era. In the early 2000s, Edwards advocated teaching testing starting in CS1 [5], arguing that "a student should be given the responsibility of demonstrating the correctness of his or her own code" [4]. In the same period, the introductory textbook *How to Design Programs* (HtDP) [6] emphasized testing as part of its design process. Back in 1993, Sitaraman and Gray called for specifications and contracts to be part of first-year CS [26].

Post-LLMs, Porter and Zingaro advocate testing as a way for students to check the code produced by agents [24]; their textbook [23] teaches students to write tests *after* using LLMs to generate programs. Our paper, in contrast (and in line with Edwards and HtDP), calls for thinking about testing *before* producing the corresponding code. In this way, tests become an independent representation of the problem, rather than one derived from the code being tested. Such independence is essential if tests are to serve as a verifier, regardless of whether a human or an agent generated the code.

Many other pedagogic ideas aim at getting early-stage CS students to produce reliable code: examples include teaching with contracts [20] and using robust type systems [11]. These works do not use the human-accountability framing that is central to this paper. There are also efforts around social accountability but these focus on design choices rather than accountability for final code.

3 Course Context

Instructors. The first author was the primary instructor. The other two authors/instructors brainstormed topics and assignments and led a few of the lectures. Our collective expertise spans software engineering, programming languages, AI, and computing education. All of us had experience teaching introductory CS, but to different audiences (non-majors, students with little to no programming experience, and students with substantial programming experience). Two had extensive experience teaching testing in introductory courses. Each of us had been experimenting with our own agentic coding projects over winter break in 2025-26 using Claude Code.

Students. We advertised the course as an exploration of agentic coding and how it might shape our early CS curriculum. As we were unsure of whether students might need to be able to read or modify generated code, we required at least one semester of CS taught at our institution. 78 students applied for 20 seats. The selected students represented each of the (multiple) introductory courses offered in our department, had between one and three semesters of programming experience, and indicated an interest in the education aspects of the course rather than just in learning how to use coding agents. All but two were in their first or second year; we also accepted a senior who hadn't programmed much in two years and one senior with more experience but a lot of interest in education. Seven students had never generated code with an agent; only two had generated code with multiple components. The rest had only generated small programs akin to an intro CS exercise.

Format. There were two 75-minute sessions per week. Attendance was required. Sessions involved a combination of small-group exercises, small-group design critiques, instructor-led design or code reviews, class-wide discussions, and some lectures.

4 Course Design

We designed the course to see what sorts of software engineering and intermediate programming techniques we could get students to learn and embrace through agentic coding. "Learn" meant that they could use the techniques when instructed to, while "embrace" meant that they would express value in and use the techniques when they had choice of how to approach projects. We view both as key to developing a practice of accountability.

Our general pedagogic strategy was to give students a problem for which a coding agent (Claude, in our case) could produce a seemingly correct solution, but which had subtleties that could surface in design or testing. Those subtleties became the entry points for discussing software and programming techniques. Table 1 summarizes the major projects, our conceptual objectives for each, and the stages in which students worked on them. The stages were important because students would do the first stage before we taught the corresponding techniques. In class, we engaged in code review and small-group discussions around how the solutions approached the underlying issue we wanted to teach. This motivated a structured lecture by the instructors on mitigations for the issue. In essence, we were following a model of Productive Failure [13] through the first-stage tasks, along with ideas from Variation Theory [19] and Contrasting Cases [2] to help students focus on the issues before we explicitly taught about them.

For us to understand how students were experiencing and learning from the assignments, and thus uncover our expert blind spots, we required students to keep a (human-written!) reflection journal per assignment. We asked them for four kinds of entries, as summarized in Table 2 (implementation details are in Section 6). We also asked them to upload their chat transcripts for each assignment (though this did not happen consistently). We hoped that the journals and transcripts would help us identify issues that did not surface during class discussion.

Framing Accountability. We framed accountability as knowing something was right and taking responsibility for it. Each assignment corresponded to a stakeholder-facing product; we asked students how they would convince someone to pay them for it. The assignments either had subtleties or were under-specified so that students would need to make decisions along the way. We hoped the professional framing would provide a concrete motivation for testing (beyond "it's a good thing"), while also giving students a target audience to whom they would justify their work.

5 Observations

5.1 The Power of Tetris

Tetris is a classic video game in which blocks fall downward from the top of the screen one at a time. The player presses keys to move

Assignment	Objective	Staged Tasks
Tetris Web-based version of the classic game of filling rows of a grid with falling blocks	Illustrate what agents can and cannot do easily; Motivate model-view-controller and testing	(1) Implement the classic version; (2) Reverse the direction so blocks float upward instead of falling; (3) Add a "gravity switch" mode through which the player changes the direction of block movement between rising and falling
Airport Weather Create a data display that relates two real-world datasets on weather and flight delays	Use new language and libraries; Test data analysis (vs model-view-controller architectures); Learn how to peer review; Motivate databases and SQL	(1) Create initial version on a 5-day sample of the two datasets; (2) Extend program to support 3-month datasets; (3) Peer-review testing plans
Requirements Checker Check whether a given set of courses satisfies given degree requirements (e.g., for a CS major), reporting discrepancies	Separate configurations (requirements) from program logic; Motivate constraint solvers; Experience tests used to check other students' code	(1) Develop design and testing plans; (2) Revise design and testing plans after lecture on externalizing configurations; (3) Implement the checker; (4) Peer-review implementations; (5) Revise post-review and conform to an instructor-provided API for use in testing
Music Tour Planner Design (but don't implement) a program that creates a performance schedule for a touring music band	Synthesize content (from lectures) on (data)types, externalization, databases, and constraints	A single-stage assignment on computing a tour schedule based on cities where a band planned to perform. Students proposed how to organize (data structures) and store needed information (CSV files vs. a SQL database); they also proposed a set of constraints on what would make a generated schedule valid
Chat Client Build a chat client; also call a local LLM to provide personalization or summary features	Practice API design, learn about client-server programming, and use local LLMs to add client-side features; this was the final course project	(1) Propose an API for a given set of desired chat operations (e.g., reading threads, creating threads, adding users to threads); (2) Compare classmates' proposals, then map their own to the newly-revealed Matrix API; (3) Develop a test-suite to identify a working Matrix server from six candidates; (4) Implement chat client against a working server; (5) Oral code-review with instructors

Table 1: Summary of the design and implementation assignments. Full handouts are available online [8].

Category	Prompts
Reflection	I'm working on; I'm feeling; I noticed; I learned; I'm wondering; Next time, I would
Bug	Description of bug; I detected it when; My hypothesis about the cause; The actual cause; Lesson/takeaway
Knowledge Note	When I was doing; I relied on knowing
Misc Note	

Table 2: The Reflection Journal Prompts

the block side to side or drop it onto the pile below. Completely filled rows in the pile disappear to make room for additional blocks.

Tetris was a terrific first assignment. Simply prompting a coding agent with "tetris" obtains a working JavaScript version in under a minute. Our students—many of whom had implemented Tetris in their intro course—were stunned. We then asked them to produce two more versions: one in which the blocks rose from the bottom instead of falling from the top, and one in which pressing the "g" key (for "gravity") would reverse gravity, switching the blocks between falling and rising mid-game.

The first variation yielded some hiccups in game play (often as blocks ran into other blocks or the edges). Students were surprised and confused: they saw the first variation as simple edits to arithmetic operations (swapping plus/minus operations and computing block starting locations). Their second variations were full of bugs: blocks lost their definitional shapes, overran one another, or disappeared entirely. The desired behavior wasn't even clear: should the stationary blocks float to the opposite edge? Should a falling block get trapped by shifting stationary blocks or remain in play?

Instructor Observation: With this simple, well-understood example and its variations, we motivated many of the upcoming course themes: the value of a model-view-controller architecture so we could test the program beyond simply playing the game, the need for clear behavioral specifications, the value of designing data structures and program concepts (e.g., whether blocks were cohesive structures or just color patterns on individual cells), and the challenges of defining good test cases. Contrasting the variations also let us discuss a bit about how LLMs generate code: many classic Tetris implementations would be in the training data and LLM prediction is sensitive to the training data distribution. This was a good opening to explain that agents generate code rather than retrieve it. While some students' prior experience implementing Tetris enhanced the initial shock factor, those coming from a different intro were still sufficiently familiar with the game to have a rough mental model of how it might work in code. We expect this core structure could be replicated with other examples.

5.2 Boundaries for Working with Claude

Coding agents raise the abstraction level at which we program to that of human language. This opens considerable accuracy risks due to under-specification, ambiguity of human language, and the (mis)alignment of a task with training data [15]. Programmers who use agents successfully describe prompts, practices, and features that constrain what agents do to raise the likelihood of getting reliable code in return [25, 30]; this is an example of what Karpathy (who also coined the term "vibe coding") calls *agentic engineering* [1]. In other words, programmers specify and constrain the abstractions through which they write agentic code. This has clear implications for accountability: humans frame the solution space for the agent, and can then focus their audits within that space.

For our second project (Airport Weather), students had Claude generate test suites. They were initially impressed when Claude created and ran dozens of tests automatically. Upon inspection, however, most were basic tests of API calls to existing Python libraries, not tests of the program logic. We used this to motivate creation of *testing plans*, high-level descriptions of what issues should be tested and what nuances or edge cases might arise. At first, many students had Claude generate these plans and the corresponding test suites with minimal human input. Once we began peer-reviewing the testing plans, however, most students got more involved, creating structured plans with sections for different kinds of tests, explicit links between these sections and the names of the corresponding actual tests, and a focus on the plans being human readable. They still used Claude, but they took a strong role in creating the plans.

Overall, testing and design plans (with datatype and architecture ideas) became the intermediate representations through which students communicated with Claude and each other (for peer review). As we read the journals, we noticed that students tended towards one of four clusters around the use of these files for accountability; we describe them as variations on the maxim "trust but verify":

- **Don't trust, work in small enough chunks to verify:** These students described being overwhelmed or discouraged by the volume of code or tests that Claude would produce if left unchecked. Their design and testing plans guided implementation to be done in stages that aligned with their perceived cognitive capacity for reviewing the output.
- **Plan carefully, verify plan consistency, then trust:** These students developed ideas on their own before using Claude to refine them into design and testing plans. They spoke of having Claude check consistency between the planning documents. Once this was done, they handed off to Claude, using distinct sessions for implementation and testing, hoping to avoid leaking information through hidden context.
- **Never fully trust, not confident they can verify:** These students got less comfortable working with agents as the assignments got harder. They appreciated how Claude could save them time on rote tasks, but felt it took away too much of the thinking they personally needed to do to be confident in their work. They valued the plans for helping them think through the problem, but did not view the plans as giving them trust in Claude's work.
- **Point the agent in the right direction, then trust and let the agent verify:** These students viewed the design and testing files as guides to help Claude find and remove bugs. They skimmed the plans that Claude produced, then let it work from there. They didn't discount the need to be accountable per se; they just trusted Claude to be good enough that they could put their time into other courses.

Overall, two students were in the last group, while the rest of the students distributed fairly evenly across the others (the first group was a touch smaller). The distribution did not correlate with students' numbers of prior semesters of CS coursework.

Instructor Observation: In hindsight, we should have more explicitly discussed the design and testing plans as forms of abstractions and setting constraints on solutions (as opposed to simply a way to document human thinking or give instructions). We also should have gone into more detail about how the elements of those plans could be useful for agents. In course evaluations, several students expressed a desire for more guidance on how to make robust and useful plan documents. We wonder whether this would have made a difference particularly to the "never fully trust" group.

Instructor Observation: Design plans typically included feature descriptions, constraints (prose, not formal), and descriptions of data or file organization. Just past the halfway point of the semester, we did an in-class exercise on using Jackson's software concepts [12] to capture some of these aspects in a semi-structured way. Next time, we would teach software concepts earlier and assess their value and suitability for helping students and coding agents communicate through specification. While specification is generally challenging for early-stage (or later!) CS students [16], semi-structured approaches may be viable.

5.3 A (Server) Bridge Too Far

The final course project focused on client-server programming, which was new to most of our students. We wanted them to build a chat client with simple features akin to those in Discord or WhatsApp: creating user accounts, creating threads, posting messages, and giving permissions on threads. To revisit concepts about data structure and API design from Music Tour Planner, we first had students propose a data model and API for such a system. Following in-class review of the many ways one could conceptualize the model, we gave them a fixed API for the rest of the project. Specifically, we used a decentralized communication protocol called Matrix [28].

To help students understand the API, we had them map (in prose) their original proposed API to Matrix. We had them write a test suite to capture the behavior of the Matrix server. To help them assess the quality of their test suites, we provided local URLs to six versions of a Matrix server. One was correct (the Matrix reference implementation); we made the other five by breaking the behavior in semantically-meaningful ways (e.g., the "add-user" command failed to actually add the user). A high-quality test suite would be able to determine which of the six servers was correct [31].

Server testing was far bumpier than we had expected. Part of this was our fault: the instructor who set up the servers had no prior experience doing so and mis-configured the rate limits. As a result, students' efforts to test the servers kept timing out, with multi-minute time gaps until they could try again. In theory, a testing harness could deal with this given multiple small testing files,

but this was an advanced testing issue that we had not discussed. Furthermore, some students were prompting Claude to *identify* the incorrect servers, rather than using Claude to help *produce a test suite* that could identify the servers (as we had intended). The former approach did not work, even after the rate limits were fixed.

Instructor Observation: This project countered oft-heard claims that LLMs make it easy for people to teach themselves. Knowing our students were new to client-server programming, we gave an overview in lecture, then told students to work with an LLM as needed to understand this in the context of Matrix. Many students lacked a mental model and vocabulary to understand those LLM conversations. This steered some students towards simply letting Claude do more of the project work. Accountability took a hit.

5.4 Code Review of Chat Clients

Students also built clients against a working Matrix server. This part of the project went smoothly. To help students learn how to call out to local LLMs within programs, we asked them to include a feature to summarize a thread and some sort of sentiment-based analysis to annotate individual posts. Students produced many creative designs: some built workplace-focused apps with annotations for meetings and to-dos; some focused on casual chats that annotated every post.

As a final check on their approaches to accountability, each student did an individual, in-person, live demo and code-review with the instructors. We asked each student (a) to imagine we were potential investors and to explain why we should trust the quality of their work, and (b) what they saw as their human value-add beyond whatever work they asked of Claude. Nearly every student showed their testing and design plans and talked about how they developed and reviewed them (most said they had anticipated this question based on our emphasis on accountability throughout the semester, but two seemed caught off guard). When we asked questions about how certain parts of the system were implemented or tested, several students navigated their code quickly, having directory structures and smaller files that reflected the structural decomposition of the system. Others spent a lot of time scrolling instead.

Instructor Observation: We had hoped that students would externalize the configuration of icons and their corresponding sentiments or post contents. Nobody had. Students either hard-coded a dictionary or simply asked the LLM to return a reasonable icon. When we asked, it was clear externalization hadn't occurred to them. Those who simply called out to an LLM had not considered that there might not always be a suitable icon. We wonder whether students missed the core motivation for externalization, or whether this was too small a usage for them to have caught it.

Instructor Observation: Students often made basic UI mistakes such as showing users options that they were not permitted to do. Because it has become so easy to generate GUIs, and these mistakes can also have security consequences, we believe future iterations may need to consider putting more time into UI design.

5.5 What Students Said They Learned

Near the end of the course, we had students annotate their original Tetris journals to describe things they might do differently now, after a semester of agentic coding experience. They also wrote end-of-semester reflections on (a) their habits or practices to work effectively with Claude, (b) times when they felt fatigued, and (c) times when they felt particularly productive.

In the Tetris annotations, every student commented about making design plans, testing, and being more involved in guiding Claude's work. One annotation read "I felt I was designing a prompt, not a plan". Several mentioned realizing the need to limit Claude: not letting it control testing and debugging, not letting it add unrequested features, and scoping which files it can read per session. One-third explicitly said that they no longer trusted Claude's claims about how well it had tested a program. The end-of-semester reflections contained similar themes about planning and guiding Claude, but also revealed the nuances that we described in Section 5.2.

6 Course Tools and Infrastructure

Coding Agents. Students used $20/month Claude accounts. Our university covered the costs to support the pilot (long-term, these costs could be akin to course fees for consumables like reactants for chemistry labs). We chose Claude Code because the instructors were using it themselves. We intentionally limited students to the $20/month plan partly to force them to space out their work and partly to prevent them from spending too much time overall with the tool. These plans would let students work for a couple of hours before locking them out for a few hours. Multiple students reported that they started assignments much earlier than normal as a result and felt this was a good thing. We ourselves had noticed how enticing these agents can be, leading to overuse, and felt a responsibility to ward off those consequences for the students.

Reflection Journal Template and Parser. We created a journal template in Google Docs with a custom drop-down menu for the four types of entries from Table 2. Selecting a menu option put preformatted headers for that entry at the end of the document, along with a date stamp. We designed the prompts to inspire and structure the sort of information we wanted towards our experimental goals (Section 1). Students did not have to respond to all prompts when making an observation, but they could not add prompts or change section structure. This enabled us to write a journal parser (in Google Apps Script) that created a Google Sheet with one row per reflection entry. We scanned the Google Sheet to learn what students were experiencing on each assignment.

Submission of Work. We maintained a GitHub organization with the students as members. We created assignments in GitHub Classroom; students submitted simply by pushing to their repos. We collected reflection journal links separately through a Google Form.

Peer Review. We developed our own peer-review system atop GitHub. We created a script that (1) pushed a copy of each student's repo under a unique four-letter code for anonymization, and (2) randomly assigned each student as a GitHub collaborator/reviewer on two anonymous repos. Students completed their reviews in a Google Form that asked for the code and was structured around a concrete rubric. A second script posted each review as a GitHub issue back to the original (non-anonymous) repo.

Cross-Student Test Runner. We developed scripts to run students' tests against each other's implementations [10]. We intended to

give students extra points for detecting issues in each other's (and our own!) implementations (especially on Requirements Checker). The basic infrastructure was straightforward, but accounting for ambiguity or underspecification in the problem statement was not. We had not left ourselves enough time to figure out how to handle that systematically. We also had not planned for students submitting work late, which had other negative interactions with the basic setup. We definitely want such a tool for future iterations. Students were both excited by and a touch nervous about this activity, both of which have positive implications for accountability in testing.

Note on instructors' AI usage. We occasionally uploaded student submissions to our university-provided Gemini accounts (which are cleared for use with sensitive data) to identify patterns in student work for class discussion or to format submissions for (manual) analysis. Grading and review were done manually by one of the instructors. We used Claude to create our course tools.

7 Looking Ahead

As instructors, our experiences from teaching this course and doing agentic coding ourselves have crystallized many ideas about university-level early CS education.

Recovering Signs that Something is Wrong. Before LLMs, instructors had multiple signals for when assignments weren't working, including questions on discussion boards and in office hours. When students use coding agents, whether permitted or not, there is no such thing as an assignment that is too hard for "them" to finish (if they have the vocabulary for it). Instructors need new signals as we work to keep students in the program-development loop.

Accountability vs Ownership. We began the course and this paper thinking about developing students' accountability for what they create with coding agents. Students' journals and Chat Client code reviews suggest that this framing is too limited: "accountability" is an *external* motivator, while "ownership" is a related *internal* motivator. Our students experienced productivity, excitement, and engagement with projects that were sufficiently challenging to make personal investment worthwhile. We hypothesize that a sense of ownership will help students stay in the loop. Personally-relevant projects are only part of the story; ownership needs some challenge.

Leveraging What Agents Enable. Our course highlighted multiple ways that agents can enhance how we teach early CS. Now, we can let students design their own APIs for a project and still use autograders, by having agents write the translators between their APIs and ours (we did this on Requirements Checker). Now, students can quickly spin up user interfaces to help them experiment with their choices of data structures and data models. They can explore multiple data structure choices by having agents help swap out code. Such capabilities let us have students work more concretely and more deeply with conceptual and design activities. Furthermore, coding agents offer ways to engage students with basic topics in security, privacy, usability, maintainability, scalability, and testability as design concerns, rather than us having to wait for their programming skills to catch up. Obviously, we can't (and shouldn't!) cram everything into first-year courses, but coding agents do give us new options to consider.

Ethical Considerations. We acknowledge the many ethical concerns around AI use. Space permits two remarks. Local models are a viable option that protect user data and offset some impacts (and make others highly visceral). For those concerned about learning, our course *is* designed around student learning, just of different content than before, and addressing the needs of much more diverse populations (e.g., those who will not go on to study computing).

Implications for TAs. Class discussion identified two aspects of the future role of TAs in courses with agentic coding: (1) TAs must (be trained to) conduct design and code reviews, and (2) TAs may be asked to comment on wildly different approaches to the same problem (rather than rely on standardized expected solutions as our courses do now). Doing these well requires experience as much as a clear rubric and reviewing protocol. Departments (such as ours) that extensively employ undergraduate TAs face critical questions here: how do we prepare and select TAs to be effective when agents enable assignments to be less structured, less homogenous, and potentially with more subtle errors in planning and design?

The Effect on CS1. Running this course strengthened our belief that genuine software-engineering skills, more ambitious projects, and some use of agentic coding belongs in CS1. Like it or not, students are using coding agents. We absolutely need to help students learn necessary manual skills, but we should also embrace the benefits. Giving at least some under-specified projects that are beyond what students can code—but not beyond what they can design and test (letting agents generate the code)—brings specification in earlier. Designs and testing plans may be easier for peers, TAs, and instructors to critique than code [22]. They offer alternative forms of success for students who initially struggle with writing code. They might also make the need for accountability visceral, while simultaneously creating an environment for students to take ownership. Most of our students saw value to bringing agentic coding into CS1, while sharing concerns about how to discourage or avoid students outsourcing their learning [14]. The first two authors have experimental CS1 courses underway along these lines [7, 17].

A Bold Thought Experiment. We should consider an idea that some readers may find horrifying: a CS1 course that makes *heavy* use of coding agents. The abstraction through which people learn to program has changed many times before. It started with assembly, moved to languages that hid registers and provided procedures, and then on to languages that hid memory management behind garbage collection and provided rich data structures. Agentic coding moves us to a vastly higher level of abstraction. As our recent CACM Viewpoint elaborates [9], students can produce working systems from a very early stage. This both enables and forces them to confront a much richer set of topics, like usability and security, that are critical but have not found room in early curricula.

Acknowledgments

Thanks to our wonderful students for their enthusiasm! Student Hailey Coval eagerly helped analyze initial data during the course. This work was partially supported by NSF award 2208731 and DARPA Agreement No. HR00112420354. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of our funders.

References

[1] AI Builder Club. 2026. Agentic Engineering: Karpathy's New Framework. https://www.aibuilderclub.com/blog/karpathy-agentic-engineering. Accessed: 2026-07-03.

[2] Louis Alfieri, Timothy J. Nokes-Malach, and Christian D. Schunn. 2013. Learning Through Case Comparisons: A Meta-Analytic Review. *Educational Psychologist* 48, 2 (2013). doi:10.1080/00461520.2013.775712

[3] Santiago Berrezueta–Guzman, Stephan Krusche, and Stefan Wagner. 2025. From Coders to Critics: Empowering Students Through Peer Assessment in the Age of AI Copilots. In *2025 International Symposium on Educational Technology (ISET)*. 66–71. doi:10.1109/ISET65607.2025.00022

[4] Stephen H. Edwards. 2003. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing* 3, 3 (2003). doi:10.1145/1029994.1029995

[5] Stephen H. Edwards. 2003. Rethinking computer science education from a test-first perspective. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Anaheim, CA, USA) *(OOPSLA '03)*. Association for Computing Machinery, New York, NY, USA, 148–155. doi:10.1145/949344.949390

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs*. MIT Press.

[7] Kathi Fisler. 2026. CSCI 0111E: Data-Centric Intro to Computing and Coding Agents. Course website, Department of Computer Science, Brown University. https://cs.brown.edu/courses/csci0111e/fall26/index.html Fall 2026. Accessed 2026-10-12.

[8] Kathi Fisler, Shriram Krishnamurthi, and Michael Littman. 2026. CSCI 1970KF: Agentic Studio. Course website, Department of Computer Science, Brown University. https://cs.brown.edu/courses/csci1970kf/agentic-spr-2026/ Spring 2026. Accessed 2026-10-12.

[9] Kathi Fisler, Shriram Krishnamurthi, and Michael Littman. 2026. Teaching Novice Computing and Programming in the Agentic AI Era. *Commun. ACM* 69, 11 (Nov. 2026), 37–40. doi:10.1145/3838277

[10] Michael H. Goldwasser. 2002. A gimmick to integrate software testing throughout the curriculum. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Kentucky) *(SIGCSE '02)*. Association for Computing Machinery, New York, NY, USA, 271–275. doi:10.1145/563340.563446

[11] John Hughes. 2008. Experiences from teaching functional programming at Chalmers. *SIGPLAN Not.* 43, 11 (Nov. 2008), 77–80. doi:10.1145/1480828.1480845

[12] Daniel Jackson. 2021. *The Essence of Software: Why Concepts Matter for Great Design*. Princeton University Press.

[13] Manu Kapur. 2024. *Productive Failure: Unlocking Deeper Learning Through the Science of Failing*. John Wiley & Sons.

[14] Paul Kirschner. 2026. Offloading? No Outsourcing! Understanding AI: Offloading or Outsourcing Thinking? https://substack.com/home/post/p-184414235. Accessed: 2026-07-03.

[15] Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, Matthew Rahtz, Tom Everitt, Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. 2020. Specification Gaming: The Flip Side of AI Ingenuity. Google DeepMind blog. https://deepmind.google/blog/specification-gaming-the-flip-side-of-ai-ingenuity/

[16] Jeff Kramer. 2007. Is abstraction the key to computing? *Commun. ACM* 50, 4 (April 2007), 36–42. doi:10.1145/1232743.1232745

[17] Shriram Krishnamurthi. 2026. CSCI 0190: Accelerated Introduction to Computer Science. Course website, Department of Computer Science, Brown University. https://cs.brown.edu/courses/csci0190/2026/ Fall 2026. Accessed 2026-10-12.

[18] Yi-Chieh Lee, Nattapat Boonprakong, Yugin Tan, Harold Soh, Alex Potanin, Viraj Kumar, Anoop K. Sinha, Chen Qian, Paul Denny, Mennatallah El-Assady, Ian Oakley, Jake Renzella, Amy Zhang, Jat Singh, Wee Sun Lee, Hsuan-Tien Lin, Jane L. E, Anthony Tang, Margaret M. Burnett, Sowmya Somanath, Renwen Zhang, Vicky Charisi, and Alexandra I. Cristea. 2026. Reshaping Undergraduate Computer Science Education in the Generative AI Era. arXiv:2606.07545 [cs.CY] https://arxiv.org/abs/2606.07545

[19] Ference Marton. 2014. *Necessary Conditions of Learning*. Routledge.

[20] Bertrand Meyer. 2009. *Touch of Class: Learning to Program Well with Objects and Contracts* (1 ed.). Springer Berlin Heidelberg. doi:10.1007/978-3-540-92145-5

[21] Mitchell J. Nathan, K. Koedinger, and Martha W. Alibali. 2001. Expert Blind Spot : When Content Knowledge Eclipses Pedagogical Content Knowledge. In *Proceedings of the Third International Conference on Cognitive Science* (Beijing, China), L. Chen et al. (Ed.). USTC Press.

[22] Joe Gibbs Politz, Shriram Krishnamurthi, and Kathi Fisler. 2014. In-flow peer-review of tests in test-first programming. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) *(ICER '14)*. Association for Computing Machinery, New York, NY, USA, 11–18. doi:10.1145/2632320.2632347

[23] Leo Porter and Daniel Zingaro. 2024. *Learn AI-assisted Python Programming: With GitHub Copilot and ChatGPT* (2nd ed.). Manning Publications.

[24] Leo Porter and Daniel Zingaro. 2026. Generative AI Solves Introductory Computer Science Tasks. Now What? In *Degrees of Change: What AI Means for Education and the Next Generation*, Juan M. Lavista Ferres (Ed.). OReilly, Chapter 13, 171–184.

[25] John Regehr. 2026. Zero-Degree-of-Freedom LLM Coding using Executable Oracles. https://john.regehr.org/writing/zero_dof_programming.html. Accessed: 2026-07-03.

[26] Murali Sitaraman and Jeff Gray. 1993. Software reuse: a context for introducing software engineering principles in a traditional computer science second course. In *Proceedings of the Conference on TRI-Ada '93* (Seattle, Washington, USA) *(TRI-Ada '93)*. Association for Computing Machinery, New York, NY, USA, 137–146. doi:10.1145/170657.170701

[27] Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. 2024. On the Self-Verification Limitations of Large Language Models on Reasoning and Planning Tasks. arXiv:2402.08115 [cs.AI]

[28] The Matrix.org Foundation. [n. d.]. *Matrix.org: An open network for secure, decentralized communication*. https://matrix.org/

[29] Sydney Von Arx, Lawrence Chan, and Beth Barnes. 2025. Recent Frontier Models Are Reward Hacking. METR. https://metr.org/blog/2025-06-05-recent-reward-hacking/

[30] Simon Willison. 2026. Writing about Agentic Engineering Patterns. https://simonwillison.net/2026/Feb/23/agentic-engineering-patterns/. Accessed: 2026-07-03.

[31] John Wrenn and Shriram Krishnamurthi. 2019. Executable Examples for Programming Problem Comprehension. In *SIGCSE International Computing Education Research Conference*.