Teaching Novice Computing and Programming in the Agentic AI Era

KATHI FISLER, Brown University, USA
SHRIRAM KRISHNAMURTHI, Brown University, USA
MICHAEL LITTMAN, Brown University, USA

AI programming agents have profoundly disrupted computing education. Rather than treating this as a threat, we argue that it is an opportunity to refocus early curricula. We introduce the "flaky compiler" metaphor to frame what AI programming enables and what it demands of its users. We argue that introductory computing courses can and must become early software-engineering courses, with the objective of teaching students to take responsibility for the work they produce with AI. This induces the need for—indeed, demands—research into how to rethink teaching program specification.

CCS Concepts: • **Social and professional topics** → **Computing education**; • **Software and its engineering** → *Software development techniques.*

Additional Key Words and Phrases: computing education, AI programming agents, software engineering, introductory programming, curriculum design

ACM Reference Format:
Kathi Fisler, Shriram Krishnamurthi, and Michael Littman. 2026. Teaching Novice Computing and Programming in the Agentic AI Era. 1, 1 (January 2026), 6 pages. https://doi.org/10.1145/3838277

1 Introduction

Grace Hopper won. She said, of producing the first compiler, "I had a running compiler, and nobody would touch it because, they carefully told me, computers could only do arithmetic; they could not do programs";[1] yet we all depend on compilers now. Later, Hopper's FLOW-MATIC (which inspired COBOL) was marketed on "the use of English words" [10]. This idea too was widely mocked—for decades. Today, we can literally write programs using natural languages.[2] This is a remarkable technological arc.

Yet, instead of celebration, much of the conversation we hear about computing education in the era of AI-programming agents exudes panic under siege: learning attacked as students submit AI-generated solutions, faculty lost about how to respond, frantic searches for AI-proof programming assignments. Computing education is on the defensive. As educators, however, we must do more than design "defeat devices". As we strategize for the huge challenge of the present moment, we should also embrace its tremendous opportunity. We argue that introductory computing can (and should) be deeply infused with the practices of (currently upper-level) software engineering, with adaptations to support the learning needs of novice computing students building programs with agents.

[1] https://www.computinghistory.org.uk/det/5487/Grace-Hopper-completes-the-A-0-Compiler/
[2] https://x.com/karpathy/status/1617979122625712128

Authors' Contact Information: Kathi Fisler, kfisler@brown.edu, Brown University, USA; Shriram Krishnamurthi, shriramc@brown.edu, Brown University, USA; Michael Littman, mlittman@brown.edu, Brown University, USA.

Manuscript submitted to ACM

We frame this proposal through a metaphor that should be helpful to experienced programmers: we think of an agent as a *flaky compiler*. Together, these words capture both what AI programming enables and what it requires; they also point to an existing knowledge base about computing and education from which we can go forward.

What is Enabled (It's a Compiler!) An AI programming agent is clearly a *compiler* from natural language to running code. In the 1950s, early programming languages freed people from the drudgery of assembly and raised the level of what a novice could build, making programming approachable and empowering to a vastly broader audience. Now the level has risen even higher, empowering possibly orders of magnitude more people to write useful programs.

What is Required (It's Flaky!) The compiler is also *flaky*, for numerous reasons. Traditional compilers "preserve meaning", but the "meaning" of natural language is fuzzy and ambiguous. Prose can also be sloppy in ways that traditional compilers simply refuse to accept. The models that translate prompts have weaknesses and problems of their own. The system is non-deterministic. Every one of these aspects creates burdens that traditional compilers do not impose. Whereas traditional compilers go to great lengths to provide a clean abstraction, the flaky compiler does not.

2 Is This New?

On the one hand, this flakiness sounds like (and is) a problem. On the other hand: how new is this, really? Humans aren't perfect at implementation, either. We program with colleagues who make mistakes. Sometimes the colleague making a mistake is actually ourself, from a little while in the past. Our requirements are ambiguous, or we misinterpret them. Requirements change, and the act of building a system itself alters our understanding of its requirements. Brooks argued 40 years ago that there is no silver bullet for developing software [2] because the essence of creating software lies in conceptual design and moving from semi-formal ideas to formal representations in code; agents don't make that problem go away.

A (reductionist) view of the field of software engineering (which we distinguish from ad hoc "coding", which lacks a clear intellectual or evidentiary basis) is that it is a set of disciplines created to help us build impressive systems that accomplish complex tasks in the face of such obstacles. Everything from requirements analysis to specification to testing to verification and more ameliorate these weaknesses. From the very start, software engineering has worked to empower humans to make computers do what they actually want in the face of numerous sources of complexity and difficulty.

With AI-programming agents, software engineering education must start far earlier in the curriculum: from close to the first moment a student engages in AI-driven programming. An ambiguous programming language demands design and requirements engineering. A flaky compiler demands extensive testing and verification. Making work auditable requires using and documenting types, helper functions, APIs, design concepts, plans, and an overall architecture. Because LLMs tend towards the norm, they demand attending both to mistakes on out-of-distribution tasks and to societal context and responsible computing (where normative solutions can miss critical nuances).

Unfortunately, we lack a knowledge-base on how to teach software engineering to novices. Because getting code onto the page was considered hard, even basic topics like testing were often maligned as being too advanced for novices. Now that getting lines of code onto the page is too easy, topics such as these must be at the forefront. Part of that requires reconsidering what it means to be educated in computing.

3 Rethinking the Goal of Computing Education

We believe the key differentiator of a trained computing person is going to be the ability to stand behind the products they produce. Decades ago, David Lorge Parnas pointed out how, while other engineering disciplines provided guarantees, software came with disclaimers [9]—and exhorted us to do better. Until recently, this call was largely ignored. But now, if anyone can easily produce a disclaimer-riddled product, the differentiator becomes producing an engineered artifact: an object that comes with quality assurances.

These assurances will cover the many dimensions of software: not only basic correctness but testability, scalability, usability, accessibility, and more, what software engineers have referred to as the "-ilities". Understanding the context of a system's use, and being able to guarantee that the product is secure, fast, usable, and more in that context, will become valuable. Observe how each of those -ilities requires understanding many different areas of computer science. In other words, the demands of assurance will both necessitate and motivate those who design software to study computing more, not less. The power of agents enables even early-stage students both to implement interesting designs and to experiment with their consequences.

As a concrete example, consider user interfaces. Previously, they required a significant amount of programming experience to construct and modify, and hence were often an upper-level topic. Now, a novice can generate a sophisticated-looking interface effortlessly. But is it a *good* one? Students who cannot manually program an interface can still peer-review each others' and rapidly iterate on feedback, arriving at something closer to a usable product—and in the process, concretely observe the difference between visual flash and usability.

4 Redesigning Introductory Courses

Putting these experiences in the hands of novice students, and preparing them for a mindset centered around assurance, requires rethinking introductory computing education. A subtle but key feature is that the "compiler" is actually a *synthesizer*: what we write in prose are not really programs but specifications. That is, any choice not pinned down is a degree of freedom for the (flaky) agent. Running the same prompt N times can result in N wildly different implementations—perhaps even in different languages. Thus, part of the art of using agents effectively is *constraining what they do*. This insight is not novel to us. Prominent agentic programmers such as Andrej Karpathy[3] and Simon Willison[4] and educators such as John Regehr[5] have spoken and written about the intermediate specifications that they provide to constrain agents and make their outputs auditable.

We envision teaching introductory students an appropriately scaled down set of related techniques. Our emphasis would not be on "prompts". Rather, students would now focus on all the other aspects of producing software: program plans, testing plans, test suites, properties, constraints, and more. They can think hard about different forms of data organization and their consequences for what questions can be computed and with what performance trade-offs. We can define a semi-structured format for these specifications, which effectively become a meaningful (gradable!) intermediate level between students and code. The "prompt" goes from just a simple natural language description of a task to a complex, multi-faceted description that captures all these viewpoints and features. While agents can produce these specifications too, it is far too easy to sign off on what they produce without knowing for sure that the output matches the *intended* task. Both peer reviews and new forms of conceptual exam questions can help students realize these

[3] https://www.aibuilderclub.com/blog/karpathy-agentic-engineering
[4] https://simonwillison.net/2026/Feb/23/agentic-engineering-patterns/
[5] https://john.regehr.org/writing/zero_dof_programming.html

difficulties and thus engage in this work themselves. They would also give students level-appropriate metrics for what it means to "stand behind" code that they produced.

Early-stage students can thus focus on realistic problems—such as consumer applications or data analyses—for which they can have interesting conversations about requirements and constraints, building on their knowledge of the world, and engage in rich peer-reviews and critiques. They can grapple with problems that are beyond what they themselves can program, but without sacrificing the ability to run and experimentally evaluate the consequences of their decisions. This is all in stark contrast with how most introductory computing courses are currently taught.

This raises a common question about teaching agentic programming to novices: where will students learn to *write* code? Computing students still need to be able to do it: it is a medium used in many subsequent courses; working in code helps us learn its strengths and weaknesses; a trained computer scientist needs to understand the computational models that form the discipline; and so on. But how much code writing students need, and *when*, is not clear.

The radical proposal would almost entirely eliminate code writing from introductory courses: students focus on designs and specifications, and we let agents write all the code. Some may ask how students can possibly audit agents' output without having first written and traced code. Let's be realistic: tracing doesn't scale, even at the lower volume of agentic output for late-semester introductory course assignments; auditing programs of a scale that do something interesting is unlikely to happen. Perhaps students need to learn how to have conversations with agents about the "-ilities" as a form of auditing; they may be able to do this meaningfully just working at the level of the specifications. We don't know whether this can work, but it is worth experimenting to find out.

This proposal is not only radical but also disruptive, because it would require inverting several courses in the standard university sequence. Returning to the flaky compiler metaphor: most current introductory courses do not show the *output* of a compiler. Rather, they use high-level programming languages to accomplish tasks and meet learning objectives, and later courses break through the abstraction of the language implementation to show what happens at increasingly lower levels. Effectively, the radical proposal would do the same here, moving much of the programming content in today's early courses later, to when it becomes necessary for students to understand the details. Flakiness means that this approach is messier than it was when assembly moved back in the curriculum; we would need experiments to understand how to properly design this.

The less radical but still significant modification is to interweave design and specification with manual code reading and writing. In this version, we initially emphasize how to *read* code so students can learn how it reflects specifications and constraints. Most introductory computing courses instead emphasize writing code from early on, even though this runs counter to most other forms of literacy! (A handful of computing education projects [1, 4, 6, 11, 12] have shown benefits to learning to read code before writing it, though usually on small programs and often with younger learners.) From there, we can have students write code to practice encoding specifications and constraints for themselves. Embracing specifications and tests as an intermediate representation buys curricular flexibility: some problems would be scoped to students' ability to write code by hand, while others would intentionally be beyond students' code-writing skills, but emphasize all the other learning objectives we have discussed.

Undertaking either redesign calls for significant research. It may seem impossible to teach introductory computing this way, but that could also just be an artifact of our own expert blind spots [7]. Instead we have to see the world through the eyes of a novice to determine what can and can't work for them, without the distorting lens of our own expertise and experience. At Brown, we ran a course in Spring 2026 for just this purpose with students who had already had a semester of programming. We are assembling our findings, but our ideas about specifications as viable intermediate representations arise from that experience.

5 A Call for Action

Computing education must respond to the quality and assurance aspects of agentic programming. Crucially, these needs are not limited to future professional software engineers. A social studies student, say, who takes only one computing course, then goes off to create programs that may be used in decision-making settings, may never take that advanced "software testing" course, so their single exposure should make sure they can both identify what can go wrong and know how to guard against it. This has *always* been true, but until recently, the scope of systems an undertrained student could produce was often limited. Now, even small prompts can produce whole systems, whose fancy interfaces can easily mask various subtle underlying weaknesses. As researchers have long observed with spreadsheets and other end-user platforms [3, 5, 8], significant advances in tooling make even casual programmers much more powerful—and hence, unwittingly, much more dangerous.

Overall, therefore, we are advocating for a significant restructuring of our early curricula. Agentic programming is here to stay. If computing education looks askance at it, then students—especially given the drumbeat coming from various media and influential people—are simply going to treat computing curricula as irrelevant. We owe it to students, and to society as a whole, to adapt our curricula so that students can embrace agentic tools while also learning what they need to design, develop, debug, and defend the code that they bring into the world. Returning to our core metaphor, the compiler part of the flaky compiler *enables* us to aim for much more ambitious goals than ever before; and the flaky parts *necessitate* a broad, multi-modal description of our systems to keep the agent on track.

A common refrain from introductory programming instructors is, "I don't teach programming—I teach problem solving!" For them, this should be a moment of liberation! If we are freed from the drudgery of painfully producing code in some low-level syntax, we should now be able to focus almost entirely on the "problem-solving" parts, but now with richer problems. We firmly believe that decades of work in computer science and software engineering and design offer a rich well of ideas and methods that can (compiler) and must (flaky) finally be put to work in introductory courses.

Acknowledgments. We are grateful for feedback from Ben Shapiro, Bill Pugh, Dan Wallach, Harrison Goldstein, Iris Bahar, Ji Yun Son, Joe Politz, Matthew Wang, Robert Goldstone, Srikumar Subramanian, Will Crichton, and the anonymous reviewers. We especially thank the 19 students in our Spring 2026 course for joining our lively experiment. 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] Marina Umaschi Bers. 2019. Coding as another language: a pedagogical approach for teaching computer science in early childhood. *Journal of Computers in Education* 6, 4 (Dec. 2019), 499–528. doi:10.1007/s40692-019-00147-3

[2] Frederick P. Brooks. 1987. No Silver Bullet Essence and Accidents of Software Engineering. *Computer* 20, 4 (April 1987), 10–19. doi:10.1109/MC.1987.1663532

[3] G. B. Davis. 1989. *CAUTION: user-developed systems can be dangerous to your organization.* John Wiley & Sons, Inc., USA, 209–228.

[4] Felienne Hermans and Marlies Aldewereld. 2017. Programming is Writing is Programming. In *Companion Proceedings of the 1st International Conference on the Art, Science, and Engineering of Programming* (Brussels, Belgium) *(Programming '17)*. Association for Computing Machinery, New York, NY, USA, Article 33, 8 pages. doi:10.1145/3079368.3079413

[5] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The state of the art in end-user software engineering. *ACM Comput. Surv.* 43, 3, Article 21 (April 2011), 44 pages. doi:10.1145/1922649.1922658

[6] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 101–112. doi:10.1145/1404520.1404531

[7] 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.

[8] Raymond R. Panko and Ralph H. Sprague, Jr. 1998. Hitting the wall: errors in developing and code inspecting a 'simple' spreadsheet model. *Decision Support Systems* 22, 4 (apr 1998), 337–353.

[9] David Lorge Parnas. 1997. Software Engineering: An Unconsummated Marriage (Extended Abstract). In *Software Engineering — ESEC/FSE'97*, Mehdi Jazayeri and Helmut Schauer (Eds.). Lecture Notes in Computer Science, Vol. 1301. Springer, Berlin, Heidelberg. doi:10.1007/3-540-63531-9_1

[10] Remington Rand. 1957. *FLOW-MATIC Programming System.* Technical Report. Remington Rand, Univac Division. https://archive.computerhistory.org/resources/text/Remington_Rand/Univac.Flowmatic.1957.102646140.pdf

[11] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. 2006. An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (Hobart, Australia) *(ACE '06)*. Australian Computer Society, Inc., AUS, 243–252.

[12] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J. Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253. arXiv:https://doi.org/10.1080/08993408.2019.1565235 doi:10.1080/08993408.2019.1565235