The Coming Commoditization of Computational Thinking

Douglas C. Schmidt is the Dean of Computing, Data Sciences & Physics
at William & Mary, Williamsburg, Virginia, USA

Abstract

Computational thinking—the ability to frame and solve problems with computational tools—has long been synonymous with programming. Traditionally, it required mastering languages like Python, Java, or C++ and navigating syntax, debugging, and algorithm design. That era is rapidly changing. Large language models (LLMs) and generative AI have lowered these barriers, making high-level problem solving accessible to anyone who can describe a problem in plain language. This shift signals the commoditization of computational thinking, turning a specialized skill into a broadly available utility.

The impact is most visible in software development. AI coding assistants such as GitHub Copilot, Claude Code, and OpenAI Codex already generate large portions of code, freeing developers to focus on intent, design, and architecture while delegating routine implementation. Programming is becoming less about handcrafting every line and more about guiding an interactive dialogue with AI. Yet this paradigm introduces challenges: AI outputs are probabilistic, not guaranteed correct. Without rigorous testing and oversight, organizations risk brittle systems and technical debt. Best practices in prompt engineering, version control, and human–AI collaboration—echoing principles of software engineering—will be vital for building reliable AI-augmented systems.

The consequences extend beyond developers. Knowledge workers in finance, law, education, and other fields are already using LLMs to generate analyses, draft reports, and automate tasks once reserved for specialists. As computational thinking democratizes, the very definition of "programmer" broadens to include anyone who can problem-solve effectively in partnership with AI. This expansion creates opportunities for innovation but heightens the need for human creativity, critical thinking, and ethical responsibility.

Education stands at a crossroads. Just as calculators reshaped math instruction, LLMs will transform computing curricula. Rather than banning AI tools, educators must teach responsible use: crafting prompts, validating outputs, and recognizing errors. Foundational knowledge of algorithms, data structures, and software principles remains essential for detecting when AI-generated solutions fall short. Equally important, ethics and academic integrity must take center stage as AI permeates classrooms.

The commoditization of computational thinking represents both disruption and opportunity. The computing community should neither resist nor surrender to this shift but reimagine professional practice and education for an AI-augmented era. Human creativity and judgment will determine whether this new utility amplifies achievement or diminishes it.

From Code to Conversation: How LLMs Are Transforming Computational Thinking

Computational thinking—the ability to formulate and solve problems with computing tools—is undergoing a significant shift. Advances in generative AI, especially large language models (LLMs) [2], are democratizing this skill and making high-level problem solving broadly accessible. Today, nearly anyone with an Internet connection can leverage LLMs to tackle complex tasks using plain English instructions, often without writing a single line of traditional code, thereby commoditizing capabilities that once required specialized programming expertise.

This emerging paradigm lowers longstanding barriers. Jeannette Wing's seminal vision of "computational thinking" [9] emphasized logical problem decomposition and algorithmic idea design.

Historically, putting that into practice meant mastering programming languages like Python, Java, or C++ and wrestling with arcane syntax and debugging, which limited who could transform ideas into software. Now, however, natural-language interfaces enable users to describe problems plainly and let an AI generate a solution [4].

Consequently, knowledge work is at a historic inflection point. Tasks like report writing, dataset analysis, and code drafting—once exclusive to skilled professionals—are increasingly automatable by AI, which creates unease about the future of many roles. In practice, a financial analyst armed with an LLM can accomplish in hours what used to take days. Likewise, a programmer paired with an AI assistant can produce software faster and often with comparable quality.

Even the very definitions of "programmer" or "developer" are evolving. If solving a problem through an interactive dialogue with an LLM counts as programming, then the ranks of those capable of "programming," broadly defined, are expanding rapidly. *Prompt engineering* – the skill of crafting precise, effective instructions for AI – has become a recognized competency in this landscape [8]. The developer's role is already shifting toward guiding or editing AI-generated work: the human outlines the problem, the AI drafts a solution (be it code, analysis, or design), and the human refines and verifies it.

Creative and architectural decisions remain human-led, but much of the routine implementation and information gathering associated with developing software-reliant systems can now be offloaded to LLMs. The net effect is that computational thinking is becoming a ubiquitous utility service. This democratization brings great opportunity but also raises urgent questions for computing professionals and educators alike.

Beyond Hand-Coding: LLMs Redefine Programming

Perhaps nowhere is the impact of LLMs more evident than in software development. Generative AI models can produce substantial chunks of source code from minimal prompts, fundamentally augmenting the programming process. Developers using AI coding assistants like GitHub Copilot, OpenAI Codex, or Claude Code are effectively working alongside a tireless AI coding partner. The results are striking: according to GitHub, nearly 46% of code written by developers who use Copilot is now AI-generated on average (and over 60% in certain languages like Java) [7].

The value of LLMs goes beyond programming speed by enabling developers to work at a higher level of abstraction, focusing more on intent and design rather than slogging through boilerplate [10]. In effect, LLMs are redefining what it means to "write a program." Instead of painstakingly coding every detail, many tasks can be accomplished by describing the problem to an LLM. The act of programming starts to look more like giving instructions in plain language, then iteratively refining the AI's output. Programming is thus becoming more of a *conversation* with the computer, with humans and AI working in tandem to produce solutions [6].

As AI-generated code moves from toy examples to production-scale software, developers must apply the same rigor that traditional software engineering demands. Companies using LLMs in their software pipeline report that naive prompt-hacking leads to brittle code, unforeseen failures, and technical debt [5]. In response, emerging best practices for *robust prompt engineering* echo long-established practices from software engineering [1]. These include systematic testing and validation of AI outputs, version control and documentation of prompts, iterative refinement based on bug reports, and integration of AI-produced modules with conventional code. These practices mirror traditional engineering disciplines and are critical for reliability.

Probabilistic, Not Perfect: The Limits of Current LLMs

For all their impressive capabilities, however, today's LLMs have well-known limitations that demand human oversight [2]. Unlike traditional programs, LLMs don't follow a deterministic series of logical steps on a dataset. Instead, they generate responses by making a sequence of probabilistic choices guided by patterns in their training data. This fundamental difference means that LLMs cannot guarantee correctness or truthfulness.

Moreover, these failures are not easily fixed by simple means like prompt rewording or model tuning. It might appear that clever "prompt engineering" (e.g., instructing the AI to double-check its work or think step-by-step) could eliminate mistakes. Careful prompting can reduce errors [8] but are no silver bullet. Because LLM outputs are generated from statistical correlations rather than grounded in formal rules, there will always be a non-zero chance of a flaw.

In practical terms, these limitations with LLMs mean that professionals must cultivate a healthy skepticism and a habit of verification. If an AI assistant writes a piece of code, it should be tested and code-reviewed like any other code since subtle errors may be hidden by the AI's polished delivery. These powerful probabilistic tools are best used as assistants, not oracles. The role of human experts is not eliminated but instead becomes even more pivotal to interpret and validate AI's contributions.

New Skills, Old Foundations: Education in the LLM Era

What does the advent of LLMs mean for computer science education? In an era where students can effortlessly generate a working program via natural-language prompts, educators face a dilemma: how to teach foundational computing skills when the "dirty work" of coding or crunching numbers is increasingly handled by AI. Some might argue that traditional programming courses are becoming obsolete. After all, why spend weeks learning Java syntax or debugging pointers in C++, when an AI copilot can produce a solution in seconds? The answer, however, is not to abandon the foundations, but to adapt our curriculum to the new reality.

First, we must integrate AI tools *into* education rather than try to ban them. Just as calculators found their way into math classes (accompanied by lessons on when and how to use them), generative AI should find its way into programming courses with appropriate guidance. Forward-thinking instructors are designing assignments that encourage responsible AI use. Rather than deeming it "cheating," they guide students to use these tools appropriately. Computing educators echo this sentiment [3], urging we teach students how to collaborate with AI and critically evaluate AI-generated results.

Students must become fluent in AI-assisted problem solving—learning to craft effective prompts, iterate in dialogue, and decompose tasks for machine partners. Alongside that, however, students must hone their critical reasoning and validation skills, e.g., testing AI outputs by writing unit tests for code an AI writes and learning to recognize when generated code or results diverge from expectations and requirements.

Importantly, none of this means the old foundations become irrelevant. On the contrary, a strong grasp of computing fundamentals is more crucial than ever – it's what enables one to detect when the AI is going astray. Understanding algorithms, data structures, and complexity helps developers notice if AI-generated solutions are inefficient or incorrect. Knowledge of security and correctness principles alerts an engineer when an AI's code suggestion might be vulnerable or flawed.

Beyond technical skills, ethical and societal discussions must become a bigger part of the curriculum. With LLMs generating content, questions of academic integrity naturally arise, so students and faculty need guidelines on what constitutes acceptable use of AI in coursework. By engaging students with these questions, we prevent misuse *and* produce professionals who are thoughtful about AI's broader

implications. In short, integrating LLMs into education isn't just about tools; it offers students a broader view of how technology intersects with responsibility and social outcomes.

In conclusion, the rise of LLMs marks the next chapter in the decades-long democratization of computing. Just as spreadsheets in the 1980s allowed non-programmers to harness computation, and the web in the 1990s put information at everyone's fingertips, AI language models now make problem-solving accessible to the masses. The computing community should neither resist this tide nor be swept away by it. Instead, we must reimagine our professional and educational practices for an AI-augmented future. Those who learn to combine human insight and judgment with AI's capabilities will achieve results previously out of reach. Those who cling to older paradigms may find themselves left behind. Computational thinking may be commoditized, but human creativity, critical thinking, and ethical responsibility remain essential. By cultivating these strengths in tandem with AI, we ensure the coming commoditization of computational thinking augments human achievement, not replaces it.

References

1. Chen, Z., Wang, C., Sun, W., Yang, G., Liu, X., Zhang, J. M., and Liu, Y. 2025. Promptware engineering: Software engineering for LLM prompt development. *arXiv* preprint arXiv:2503.02400. https://arxiv.org/abs/2503.02400.
2. Cerf, V. G. 2023. Large language models. *Commun. ACM* 66, 8 (Aug. 2023), 7–8.
3. Denny, P., et al. 2024. Computing education in the era of generative AI. *Commun. ACM* 67, 11 (Nov. 2024), 56–65.
4. Michaelsen, G. A., and dos Santos, R. P. 2024. Is English the new programming language? How about pseudo-code engineering? *Acta Scientiarum* 26, 1 (Jan./Feb. 2024), 157–204.
5. Menshawy, A., et al. 2024. Navigating challenges and technical debt in large language model deployment. In *Proc. EuroMLSys'24*.
6. O'Reilly, T. 2025. AI and programming: The beginning of a new era. *O'Reilly Radar* (May 8, 2025).
7. Ramel, D. 2023. GitHub Copilot AI tech upgraded, already generates 61% of Java code. *Visual Studio Magazine* (Feb. 15, 2023).
8. White, J., et al. 2023. A prompt pattern catalog to enhance prompt engineering with ChatGPT. In *Proc. 30th Pattern Languages of Programming Conf. (PLoP'23)*.
9. Wing, J. M. 2006. Computational thinking. *Commun. ACM* 49, 3 (Mar. 2006), 33–35.
10. Ziegler, A., et al. 2024. Measuring GitHub Copilot's impact on productivity. *Commun. ACM* (Feb. 2024).