PERSONAL HISTORY

A CODER CONSIDERS THE WANING DAYS OF THE CRAFT

Coding has always felt to me like an endlessly deep and rich domain. Now I find myself wanting to write a eulogy for it.

By James Somers

November 13, 2023

Artificial intelligence still can't beat a human when it comes to programming. But it's only a matter of time. Illustration by Dev Valladares

Save this story

I have always taken it for granted that, just as my parents made sure that I could read and write, I would make sure that my kids could program computers. It is among the newer arts but also among the most essential, and ever more so by the day, encompassing everything from filmmaking to physics. Fluency with code would round out my children's literacy—and keep them employable. But as I write this my wife is pregnant with our first child, due in about three weeks. I code professionally, but, by the time that child can type, coding as a valuable skill might have faded from the world.

I first began to believe this on a Friday morning this past summer, while working on a small hobby project. A few months back, my friend Ben and I had resolved to create a *Times*-style crossword puzzle entirely by computer. In 2018, we'd made a Saturday puzzle with the help of software and were surprised by how little we contributed—just applying our taste here and there. Now we would attempt to build a crossword-making program that didn't require a human touch.

When we've taken on projects like this in the past, they've had both a hardware component and a software component, with Ben's strengths running toward the former. We once made a neon sign that would glow when the subway was approaching the stop near our apartments. Ben bent the glass and wired up the transformer's circuit board. I wrote code to process the transit data. Ben has some professional coding experience of his own, but it was brief, shallow, and now about twenty years out of date; the serious coding was left to me. For the new crossword project, though, Ben had introduced a third party. He'd signed up for a ChatGPT Plus subscription and was using GPT-4 as a coding assistant.

Something strange started happening. Ben and I would talk about a bit of software we wanted for the project. Then, a shockingly short time later, Ben would deliver it himself. At one point, we wanted a command that would print a hundred random lines from a dictionary file. I thought about the problem for a few minutes, and, when thinking failed, tried Googling. I made some false starts using what I could gather, and while I did my thing—programming—Ben told GPT-4 what he wanted and got code that ran perfectly.

Fine: commands like those are notoriously fussy, and everybody looks them up anyway. It's not real programming. A few days later, Ben talked about how it would be nice to have an iPhone app to rate words from the dictionary. But he had no idea what a pain it is to make an iPhone app. I'd tried a few times and never got beyond something that half worked. I found Apple's programming environment forbidding. You had to learn not just a new language but a new program for editing and running code; you had to learn a zoo of "U.I. components" and all the complicated ways of stitching them together; and, finally, you had to figure out how to package the app. The mountain of new things to learn never seemed worth it. The next morning, I woke up to an app in my in-box that did exactly what Ben had said he wanted. It worked perfectly, and even had a cute design. Ben said that he'd made it in a few hours. GPT-4 had done most of the heavy lifting.

By now, most people have had experiences with A.I. Not everyone has been impressed. Ben recently said, "I didn't start really respecting it until I started having it write code for me." I suspect that non-programmers who are skeptical by nature, and who have seen ChatGPT turn out wooden prose or bogus facts, are still underestimating what's happening.

Bodies of knowledge and skills that have traditionally taken lifetimes to master are being swallowed at a gulp. Coding has always felt to me like an endlessly deep and rich domain. Now I find myself wanting to write a eulogy for it. I keep thinking of Lee Sedol. Sedol was one of the world's best Go players, and a national hero in South Korea, but is now best known for losing, in 2016, to a computer program called AlphaGo. Sedol had walked into the competition believing that he would easily defeat the A.I. By the end of the days-long match, he was proud of having eked out a single game. As it became clear that he was going to lose, Sedol said, in a press conference, "I want to apologize for being so powerless." He retired three years later. Sedol seemed weighed down by a question that has started to feel familiar, and urgent: What will become of this thing I've given so much of my life to?

My first enchantment with computers came when I was about six years old, in Montreal in the early nineties, playing Mortal Kombat with my oldest brother. He told me about some "fatalities"—gruesome, witty ways of killing your opponent. Neither of us knew how to inflict them. He dialled up an FTP server (where files were stored) in an MS-DOS terminal and typed obscure commands. Soon, he had printed out a page of codes—instructions for every fatality in the game. We went back to the basement and exploded each other's heads.

I thought that my brother was a hacker. Like many programmers, I dreamed of breaking into and controlling remote systems. The point wasn't to cause mayhem—it was to find hidden places and learn hidden things. "My crime is that of curiosity," goes "The Hacker's Manifesto," written in 1986 by Loyd Blankenship. My favorite scene from the 1995 movie "Hackers" is when Dade Murphy, a newcomer, proves himself at an underground club. Someone starts pulling a rainbow of computer books out of a backpack, and Dade recognizes each one from the cover: the green book on international Unix environments; the red one on N.S.A.-trusted networks; the one with the pink-shirted guy on I.B.M. PCs. Dade puts his expertise to use when he turns on the sprinkler system at school, and helps right the ballast of an oil tanker—all by tap-tapping away at a keyboard. The lesson was that knowledge is power.

But how do you actually learn to hack? My family had settled in New Jersey by the time I was in fifth grade, and when I was in high school I went to the Borders bookstore in the Short Hills mall and bought "Beginning Visual C++," by Ivor Horton. It ran to twelve hundred pages—my first grimoire. Like many tutorials, it was easy at first and then, suddenly, it wasn't. Medieval students called the moment at which casual learners fail the *pons asinorum*, or "bridge of asses." The term was inspired by Proposition 5 of Euclid's Elements I, the first truly difficult idea in the book. Those who crossed the bridge would go on to master geometry; those who didn't would remain dabblers. Section 4.3 of "Beginning Visual C++," on "Dynamic Memory Allocation," was my bridge of asses. I did not cross.

But neither did I drop the subject. I remember the moment things began to turn. I was on a long-haul flight, and I'd brought along a boxy black laptop and a CD-ROM with the Borland C++ compiler. A compiler translates code you write into code that the machine can run; I had been struggling for days to get this one to work. By convention, every coder's first program does nothing but generate the words "Hello, world." When I tried to run my version, I just got angry error messages. Whenever I fixed one problem, another cropped up. I had read the "Harry Potter" books and felt as if I were in possession of a broom but had not yet learned the incantation to make it fly. Knowing what might be possible if I did, I kept at it with single-minded devotion. What I learned was that programming is not really about knowledge or skill but simply about patience, or maybe obsession. Programmers are people who can endure an endless parade of tedious obstacles. Imagine explaining to a simpleton how to assemble furniture over the phone, with no pictures, in a language you barely speak. Imagine, too, that the only response you ever get is that you've suggested an absurdity and the whole thing has gone awry. All the sweeter, then, when you manage to get something assembled. I have a distinct memory of lying on my stomach in the airplane aisle, and then hitting Enter one last time. I sat up. The computer, for once, had done what I'd told it to do. The words "Hello, world" appeared above my cursor, now in the computer's own voice. It seemed as if an intelligence had woken up and introduced itself to me.

Most of us never became the kind of hackers depicted in "Hackers." To "hack," in the parlance of a programmer, is just to tinker—to express ingenuity through code. I never formally studied programming; I just kept messing around, making computers do helpful or delightful little things. In my freshman year of college, I knew that I'd be on the road during the third round of the 2006 Masters Tournament, when Tiger Woods was moving up the field, and I wanted to know what was happening in real time. So I made a program that scraped the leaderboard on pgatour.com and sent me a text message anytime he birdied or bogeyed. Later, after reading "Ulysses" in an English class, I wrote a program that pulled random sentences from the book, counted their syllables, and assembled haikus—a more primitive regurgitation of language than you'd get from a chatbot these days, but nonetheless capable, I thought, of real poetry:

I'll flay him alive
Uncertainly he waited
Heavy of the past

I began taking coding seriously. I offered to do programming for a friend's startup. The world of computing, I came to learn, is vast but organized almost geologically, as if deposited in layers. From the Web browser down to the transistor, each sub-area or system is built atop some other, older sub-area or system, the layers dense but legible. The more one digs, the more one develops what the race-car driver Jackie Stewart called "mechanical sympathy," a sense for the machine's strengths and limits, of what one could make it do.

At my friend's company, I felt my mechanical sympathy developing. In my sophomore year, I was watching "Jeopardy!" with a friend when he suggested that I make a playable version of the show. I thought about it for a few hours before deciding, with much disappointment, that it was beyond me. But when the idea came up again, in my junior year, I could see a way through it. I now had a better sense of what one could do with the machine. I spent the next fourteen hours building the game. Within weeks, playing "Jimbo Jeopardy!" had become a regular activity among my friends. The experience was profound. I could understand why people poured their lives into craft: there is nothing quite like watching someone enjoy a thing you've made.

In the midst of all this, I had gone full "Paper Chase" and begun ignoring my grades. I worked voraciously, just not on my coursework. One night, I took over a half-dozen machines in a basement computer lab to run a program in parallel. I laid printouts full of numbers across the floor, thinking through a pathfinding algorithm. The cost was that I experienced for real that recurring nightmare in which you show up for a final exam knowing nothing of the material. (Mine was in Real Analysis, in the math department.) In 2009, during the most severe financial crisis in decades, I graduated with a 2.9 G.P.A.

And yet I got my first full-time job easily. I had work experience as a programmer; nobody asked about my grades. For the young coder, these were boom times. Companies were getting into bidding wars over top programmers. Solicitations for experienced programmers were so aggressive that they complained about "recruiter spam." The popularity of university computer-science programs was starting to explode. (My degree was in economics.) Coding "boot camps" sprang up that could credibly claim to turn beginners into high-salaried programmers in less than a year. At one of my first job interviews, in my early twenties, the C.E.O. asked how much I thought I deserved to get paid. I dared to name a number that faintly embarrassed me. He drew up a contract on the spot, offering ten per cent more. The skills of a "software engineer" were vaunted. At one company where I worked, someone got in trouble for using HipChat, a predecessor to Slack, to ask one of my colleagues a question. "Never HipChat an engineer directly," he was told. We were too important for that.

This was an era of near-zero interest rates and extraordinary tech-sector growth. Certain norms were established. Companies like Google taught the industry that coders were to have free espresso and catered hot food, world-class health care and parental leave, on-site gyms and bike rooms, a casual dress code, and "twenty-per-cent time," meaning that they could devote one day a week to working on whatever they pleased. Their skills were considered so crucial and delicate that a kind of superstition developed around the work. For instance, it was considered foolish to estimate how long a coding task might take, since at any moment the programmer might turn over a rock and discover a tangle of bugs. Deadlines were anathema. If the pressure to deliver ever got too intense, a coder needed only to speak the word "burnout" to buy a few months.

From the beginning, I had the sense that there was something wrongheaded in all this. Was what we did really so precious? How long could the boom last? In my teens, I had done a little Web design, and, at the time, that work had been in demand and highly esteemed. You could earn thousands of dollars for a project that took a weekend. But along came tools like Squarespace, which allowed pizzeria owners and freelance artists to make their own Web sites just by clicking around. For professional coders, a tranche of high-paying, relatively low-effort work disappeared.

The response from the programmer community to these developments was just, Yeah, you have to keep levelling up your skills. Learn difficult, obscure things. Software engineers, as a species, love automation. Inevitably, the best of them build tools that make other kinds of work obsolete. This very instinct explained why we were so well taken care of: code had immense leverage. One piece of software could affect the work of millions of people. Naturally, this sometimes displaced programmers themselves. We were to think of these advances as a tide coming in, nipping at our bare feet. So long as we kept learning we would stay dry. Sound advice—until there's a tsunami.

When we were first allowed to use A.I. chatbots at work, for programming assistance, I studiously avoided them. I expected that my colleagues would, too. But soon I started seeing the telltale colors of an A.I. chat session—the zebra pattern of call-and-response—on programmers' screens as I walked to my desk. A common refrain was that these tools made you more productive; in some cases, they helped you solve problems ten times faster.

I wasn't sure I wanted that. I enjoy the act of programming and I like to feel useful. The tools I'm familiar with, like the text editor I use to format and to browse code, serve both ends. They enhance my practice of the craft—and, though they allow me to deliver work faster, I still feel that I deserve the credit. But A.I., as it was being described, seemed different. It provided a *lot* of help. I worried that it would rob me of both the joy of working on puzzles and the satisfaction of being the one who solved them. I could be infinitely productive, and all I'd have to show for it would be the products themselves.

The actual work product of most programmers is rarely exciting. In fact, it tends to be almost comically humdrum. A few months ago, I came home from the office and told my wife about what a great day I'd had wrestling a particularly fun problem. I was working on a program that generated a table, and someone had wanted to add a header that spanned more than one column—something that the custom layout engine we'd written didn't support. The work was urgent: these tables were being used in important documents, wanted by important people. So I sequestered myself in a room for the better part of the afternoon. There were lots of lovely sub-problems: How should I allow users of the layout engine to convey that they want a column-spanning header? What should *their* code look like? And there were fiddly details that, if ignored, would cause bugs. For instance, what if one of the columns that the header was supposed to span got dropped because it

didn't have any data? I knew it was a good day because I had to pull out pen and pad—I was drawing out possible scenarios, checking and double-checking my logic.
But taking a bird's-eye view of what happened that day? A table got a new header. It's hard to imagine anything more mundane. For me, the pleasure was entirely in the process, not the product. And what would become of the process if it required nothing more than a three-minute ChatGPT session? Yes, our jobs as programmers involve many things besides literally writing code, such as coaching junior hires and designing systems at a high level. But coding has always been the root of it. Throughout my career, I have been interviewed and selected precisely for my ability to solve fiddly little programming puzzles. Suddenly, this ability was less important.

I had gathered as much from Ben, who kept telling me about the spectacular successes he'd been having with GPT-4. It turned out that it was not only good at the fiddly stuff but also had the qualities of a senior engineer: from a deep well of knowledge, it could suggest ways of approaching a problem. For one project, Ben had wired a small speaker and a red L.E.D. light bulb into the frame of a portrait of King Charles, the light standing in for the gem in his crown; the idea was that when you entered a message on an accompanying Web site the speaker would play a tune and the light would flash out the message in Morse code. (This was a gift for an eccentric British expat.) Programming the device to fetch new messages eluded Ben; it seemed to require specialized knowledge not just of the microcontroller he was using but of Firebase, the back-end server technology that stored the messages. Ben asked me for advice, and I mumbled a few possibilities; in truth, I wasn't sure that what he wanted would be possible. Then he asked GPT-4. It told Ben that Firebase had a capability that would make the project much simpler. Here it was—and here was some code to use that would be compatible with the microcontroller.

Afraid to use GPT-4 myself—and feeling somewhat unclean about the prospect of paying OpenAI twenty dollars a month for it—I nonetheless started probing its capabilities, via Ben. We'd sit down to work on our crossword project, and I'd say, "Why don't you try prompting it this way?" He'd offer me the keyboard. "No, you drive," I'd say. Together, we developed a sense of what the A.I. could do. Ben, who had more experience with it than I did, seemed able to get more out of it in a stroke. As he later put it, his own neural network had begun to align with GPT-4's. I would have said that he had achieved mechanical sympathy. Once, in a feat I found particularly astonishing, he had the A.I. build him a Snake game, like the one on old Nokia phones. But then, after a brief exchange with GPT-4, he got it to modify the game so that when you lost it would show you how far you strayed from the most efficient route. It took the bot about ten seconds to achieve this. It was a task that, frankly, I was not sure I could do myself.
In chess, which for decades now has been dominated by A.I., a player's only hope is pairing up with a bot. Such half-human, half-A.I. teams, known as centaurs, might still be able to beat the best humans and the best A.I. engines working alone. Programming has not yet gone the way of chess. But the centaurs have arrived. GPT-4 on its own is, for the moment, a worse programmer than I am. Ben is much worse. But Ben plus GPT-4 is a dangerous thing.

It wasn't long before I caved. I was making a little search tool at work and wanted to highlight the parts of the user's query that matched the results. But I was splitting up the query by words in a way that made things much more complicated. I found myself short on patience. I started thinking about GPT-4. Perhaps instead of spending an afternoon programming I could spend some time "prompting," or having a conversation with an A.I.

In a 1978 essay titled "On the Foolishness of 'Natural Language Programming,' " the computer scientist Edsger W. Dijkstra argued that if you were to instruct computers not in a specialized language like C++ or Python but in your native tongue you'd be rejecting the very precision that made computers useful. Formal programming languages, he wrote, are "an amazingly effective tool for ruling out all sorts of nonsense that, when we use our native tongues, are almost impossible to avoid." Dijkstra's argument became a truism in programming circles. When the essay made the rounds on Reddit in 2014, a top commenter wrote, "I'm not sure which of the following is scariest. Just how trivially obvious this idea is" or the fact that "many still do not know it."

When I first used GPT-4, I could see what Dijkstra was talking about. You can't just say to the A.I., "Solve my problem." That day may come, but for now it is more like an instrument you must learn to play. You have to specify what you want carefully, as though talking to a beginner. In the search-highlighting problem, I found myself asking GPT-4 to do too much at once, watching it fail, and then starting over. Each time, my prompts became less ambitious. By the end of the conversation, I wasn't talking about search or highlighting; I had broken the problem into specific, abstract, unambiguous sub-problems that, together, would give me what I wanted.

Having found the A.I.'s level, I felt almost instantly that my working life had been transformed. Everywhere I looked I could see GPT-4-size holes; I understood, finally, why the screens around the office were always filled with chat sessions—and how Ben had become so productive. I opened myself up to trying it more often.

I returned to the crossword project. Our puzzle generator printed its output in an ugly text format, with lines like `"s""c""a""r""*""k""u""n""i""s""*" "a""r""e""a"`. I wanted to turn output like that into a pretty Web page that allowed me to explore the words in the grid, showing scoring information at a glance. But I knew the task would be tricky: each letter had to be tagged with the words it belonged to, both the across and the down. This was a detailed problem, one that could easily consume the better part of an evening. With the baby on the way, I was short on free evenings. So I began a conversation with GPT-4. Some back-and-forth was required; at one point, I had to read a few lines of code myself to understand what it was doing. But I did little of the kind of thinking I once believed to be constitutive of coding. I didn't think about numbers, patterns, or loops; I didn't use my mind to simulate the activity of the computer. As another coder, Geoffrey Litt, wrote after a similar experience, "I never engaged my detailed programmer brain." So what *did* I do?

Perhaps what pushed Lee Sedol to retire from the game of Go was the sense that the game had been forever cheapened. When I got into programming, it was because computers felt like a form of magic. The machine gave you powers but required you to study its arcane secrets—to learn a spell language. This took a particular cast of mind. I felt selected. I devoted myself to tedium, to careful thinking, and to the accumulation of obscure knowledge. Then, one day, it became possible to achieve many of the same ends without the thinking and without the knowledge. Looked at in a certain light, this can make quite a lot of one's working life seem like a waste of time.

But whenever I think about Sedol I think about chess. After machines conquered that game, some thirty years ago, the fear was that there would be no reason to play it anymore. Yet chess has never been more popular—A.I. has enlivened the game. A friend of mine picked it up recently. At all hours, he has access to an A.I. coach that can feed him chess problems just at the edge of his ability and can tell him, after he's lost a game, exactly where he went wrong. Meanwhile, at the highest levels, grandmasters study moves the computer proposes as if reading tablets from the gods. Learning chess has never been easier; studying its deepest secrets has never been more exciting.

Computing is not yet overcome. GPT-4 is impressive, but a layperson can't wield it the way a programmer can. I still feel secure in my profession. In fact, I feel somewhat more secure than before. As software gets easier to make, it'll proliferate; programmers will be tasked with its design, its configuration, and its maintenance. And though I've always found the fiddly parts of programming the most calming, and the most essential, I'm not especially good at them. I've failed many classic coding interview tests of the kind you find at Big Tech companies. The thing I'm relatively good at is knowing what's worth building, what users like, how to communicate both technically and humanely. A friend of mine has called this A.I. moment "the revenge of the so-so programmer." As coding per se begins to matter less, maybe softer skills will shine.

That still leaves open the matter of what to teach my unborn child. I suspect that, as my child comes of age, we will think of "the programmer" the way we now look back on "the computer," when that phrase referred to a person who did calculations by hand. Programming by typing C++ or Python yourself might eventually seem as ridiculous as issuing instructions in binary onto a punch card. Dijkstra would be appalled, but getting computers to do precisely what you want might become a matter of asking politely.

So maybe the thing to teach isn't a skill but a spirit. I sometimes think of what I might have been doing had I been born in a different time. The coders of the agrarian days probably futzed with waterwheels and crop varietals; in the Newtonian era, they might have been obsessed with glass, and dyes, and timekeeping. I was reading an oral history of neural networks recently, and it struck me how many of the people interviewed—people born in and around the nineteen-thirties—had played with radios when they were little. Maybe the next cohort will spend their late nights in the guts of the A.I.s their parents once regarded as black boxes. I shouldn't worry that the era of coding is winding down. Hacking is forever. ♦

Published in the print edition of the November 20, 2023, issue, with the headline "Begin End."

More Science and Technology

- Can we stop runaway A.I.?
- Saving the climate will depend on blue-collar workers. Can we train enough of them before time runs out?
- There are ways of controlling A.I.—but first we need to stop mythologizing it.
- A security camera for the entire planet.
- What's the point of reading writing by humans?
- A heat shield for the most important ice on Earth.
- The climate solutions we can't live without.

Sign up for our daily newsletter to receive the best stories from *The New Yorker*.

James Somers is a writer and a programmer based in New York.