varnelis.net

What did Vibe Coding just do to the Commons?

Kazys Varnelis
December 15, 2025
Updated December 27, 2025

I write a lot about art and architecture, landscape, and the impact of technology on culture, but I haven't written about coding since the 1980s, when I sold my first article to *Creative Computing* magazine. Back then, I was a high school kid, spending hours working in both BASIC and 6502 assembler on the VIC-20. I loved assembler, also dubbed "machine code." It was a thrill getting so deep into a machine that you knew what was being shuffled from the microprocessor to the graphics chip or serial port to communicate with the world.

That feeling of getting inside the machine, making it do what you wanted—the hacker mindset—was also what the personal computer had promised. When the VIC-20 was released in 1981, William Shatner asked in the ads, "Why buy just a video game?" The personal computer was a complete break from the first mainframe era of the 1950s and 1960s, when computing meant submitting jobs to a priesthood and waiting hours for results, and from the second mainframe era of the 1970s, when access was restricted to universities and corporations. For a high school kid, getting paid for articles—even ones that were never published—was an incredible feeling. But the joy of working with early computers produced a whole subculture. I was in a user group in Berkshire County, Massachusetts, and we would trade programs we had written, copying them onto cassettes we brought to the meetings. Joseph Vanhoenacker, who ran the group, was the director of Berkshire Mental Health, a lovely man perfectly willing to put up with a fifteen-year-old who wouldn't stop talking about the possibilities computers created; like everyone else there, he shared the sense that everything would soon be different.

By 1990, everything was different, but our control over computers had quietly collapsed. Computers became genuinely useful. Everyone in college was writing their essays on computers; businesses used spreadsheets; you could balance your checkbook with Quicken, but yet another form of

disenfranchisement was underway. One fall, I came back to university, and the department secretary was gone. The faculty, who had come of age when using a typewriter was not considered appropriate for anyone hoping to be taken seriously as an academic, had somehow learned to type, and her services were no longer needed. The first great wave of computer-driven white-collar job extinctions was starting, and women without college degrees lost a path to the middle class. But more than that, people stopped writing their own software and bought it shrink-wrapped from stores instead. The complexity had scaled beyond what any hobbyist could manage. The machine was still technically programmable, but the barrier had risen out of reach. The first culprit was the Macintosh, released in 1984 and marketed as "a computer for the rest of us"—but "the rest of us" meant users, not programmers. The graphical interface hid the machine's workings beneath icons and windows that felt intuitive, even magical. There was no prompt when you turned on the computer, no command line at all.

Arthur C. Clarke famously wrote that any sufficiently advanced technology is indistinguishable from magic. Starting with the Mac (or more technically, the Lisa), Steve Jobs took that as a design brief. It succeeded by making you forget there was code underneath. The IBM PC and its clones kept a command line visible, but they were headed in the same direction. Soon, the machine had become a beige box you operated, not a system you controlled. For my part, I had no patience for abstract math and even less patience for Pascal, the highly formal programming language taught in computer science programs. When I got a Mac in 1990, I briefly tried programming it, but the process was so unfamiliar and complicated that I gave up. I've returned to programming every now and then—for example, I wrote some Python code to drive my installation Perkūnas—but I never embraced coding the way I had in high school. I've always felt that as a loss. I've enjoyed my career, but this was a path I didn't take, a whole branch of life gone. Moreover, for me, being a coder wasn't just being a nerd; it was wrapped up in the punk-rock ethos of hacking. No polished interface can substitute for that.

Last October, at a workshop at Camp in Aulus-les-Bains led by Matthew Olden, I got a glimpse of that feeling again. Matthew—who, along with the other instructors, Kathy Hinde and Carl Stone, I now count as a friend—is a musician and programmer who has spent 25 years developing his own generative music software and releasing it online. In 2004, his band won an award for best left-field electronic act; the judges didn't realize the tracks were generated by algorithms. At camp, the class he taught was on "vibe coding": describing what you want in a prompt and letting an AI write the code for you. My interest was in programming Arduinos, which I had never done before, and within a day or two, using Claude Sonnet 3.5 and ChatGPT 4o, I was able to recreate the code for Perkūnas on the Arduino-like ESP32. Not bad, I thought.

When I got back to the US, I had a pressing need for a WordPress plug-in for my website so I could search for and export selected posts to text files. I used vibe coding to put one together. It brought down a staging version of my site a couple of times while I was debugging, but overall, it worked and has continued to work flawlessly for the last year. But more advanced projects were beyond vibe coding's reach. I wasted much of November trying to get an ESP32 to handle another project for a biennale, but it turned out to be impossible because the hardware doesn't support host-mode audio over USB. Both ChatGPT 4o and Claude Sonnet 3.5 deceived me, falsely claiming to have looked up and read online documentation, getting into death loops, and offering the same solutions over and over. I gave up on vibe coding and focused on working collaboratively with AIs on artistic work, a project that became Fables of Acceleration.

About two weeks ago, I noticed increased chatter on social media about how well Claude Code, an AI coding tool from Anthropic, works with the new Opus 4.5 model. I tried an experiment: create a version of Spectre, a desktop tank combat game I remembered from the early 1990s, written in JavaScript so it could run in a browser. The result was primitive, but after half an hour of coding and tweaking, it clearly worked.

The most common critique of AI coding is that it merely regurgitates existing content. This critique is itself regurgitated so reflexively, in such identical phrasing, that one wonders if the critics have considered the irony. Still, I thought it best to challenge the AI to reimagine that game as a 3D wireframe car shooter set on the Los Angeles freeway system. It worked immediately. I spent a day doing other things—writing, answering email, cleaning the studio, training Ajman the cat to do new tricks—checking in occasionally to offer feedback, never once touching the code myself. By the end of the day, I had a browser-based game I called Sig Alert, after the California Highway Patrol's term for a traffic incident blocking lanes for thirty minutes or more. The game is a throwback to my experience living in LA between 1995 and 2005. Other drivers, gripped by road rage, are shooting at you; you shoot back. But there are civilians too, and if you hit too many of them, the police begin chasing you. *Falling Down* as a video game. If you kill more than ten civilians, the game announces "make mine animal style!"—a reference to In-N-Out Burger's secret menu—at which point everyone starts shooting at you, and you gain points from shooting everyone. I had Claude generate an 8-bit chiptune soundtrack inspired by Throbbing Gristle, Chris & Cosey, and Pink Floyd's "On the Run," while Google Nano Banana Pro produced a splash screen featuring my 1983 Saab. The game is deliberately rough since roughness is part of the aesthetic. Jen thinks a game about shooting others on the freeway is immoral. She's not wrong.

A throwback cover and splash screen featuring my 1983 Saab 900S in the manner of a 1990s game cover for Sig Alert. That's not me; I would never have worn a bandana around my head.

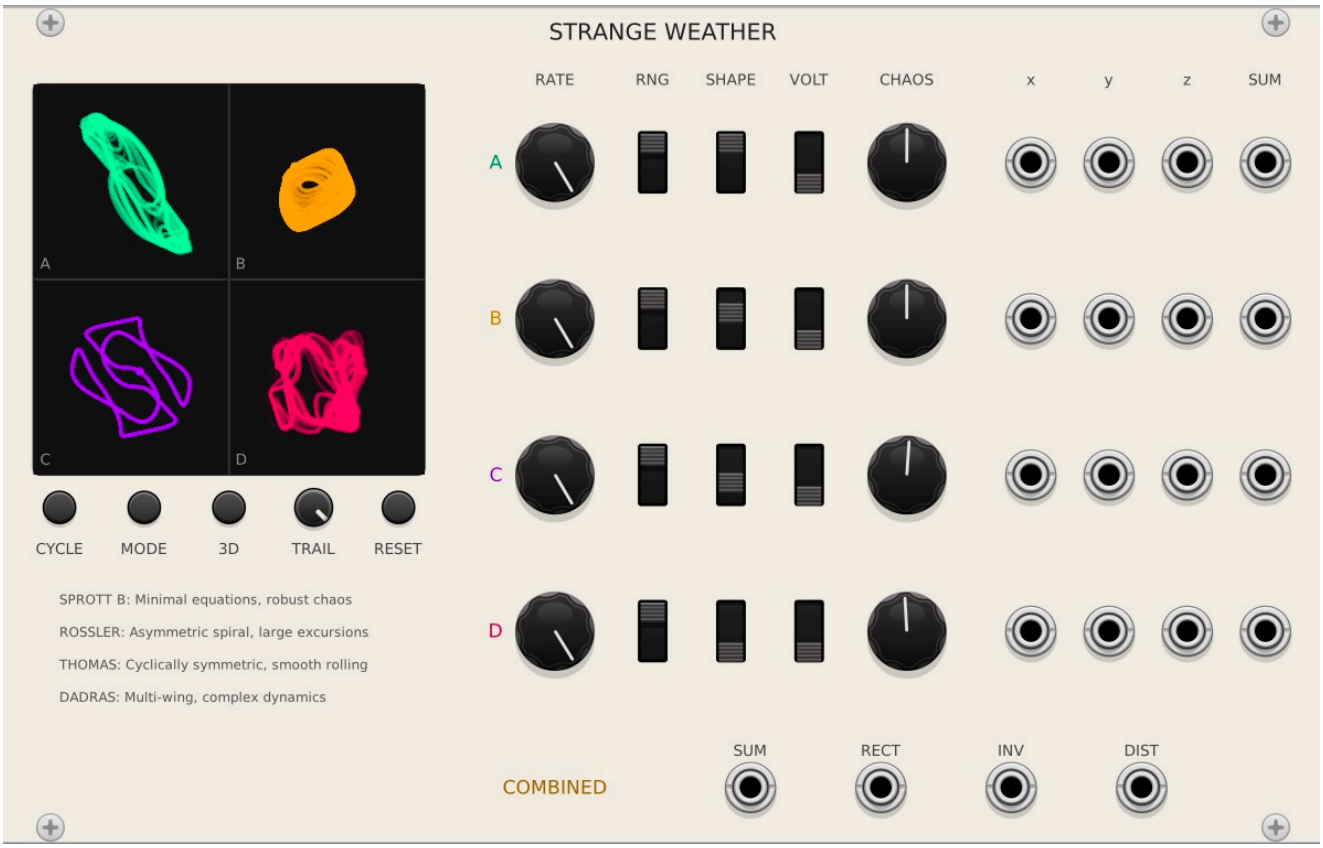
But my kid, now in a game program at NYU, is going to be the game developer, not me. Sig Alert was just proof that vibe coding could work better now. I have a few art projects underway, and when those find a home, I'll be glad to show them online. But I was also curious about just how far this could go. I decided to take up some old software that had fallen by the wayside, so I went to GitHub—the platform where most open-source software now lives, a combination of code repository and social network for programmers that has become the de facto infrastructure of collaborative development. I started with JPEGDeux, a simple Mac slideshow program that was itself a revival of JPEGView, a beloved piece of postcardware first released in 1991 when Macs ran on Motorola 68000 processors. When that was no longer viable, the JPEGDeux fork allowed the program to run on OS X, first on PowerPC and later on Intel Macs. Now, with a fourth chipset, Apple Silicon, Jpegview was finally orphaned. I had asked both Claude and ChatGPT to rebuild it last year, and while they had some success, there were fundamental issues we never got past, notably, images did not scale to the full size of a window. The result felt like it was badly written by AI, because it was. This time, I used Claude Code Opus 4.5 directly on a GitHub fork (a copy of the code I cloned into my own GitHub repository), and within a few

tries, had it running as well as it ever had. I added the ability to display videos, a file picker, and other enhancements in an afternoon. You can download the latest release here.

But this was still small beer. About ten days ago, I caught a cold, and it brought me down for about a week. When I'm sick, my brain is off as far as high-level processes like writing go, and even reading is no fun, but vibe coding was just my speed and surely better than doom scrolling. I thought about what the single most useful application would be for my own workflow. I often get poorly scanned PDFs of publications, and the downloadable, public-domain books on Google Books usually leave a lot to be desired. In the past, I used a program called ScanTailor for processing, but the workflow was clunky. It couldn't take a PDF or export one after it was finished; it worked on a directory of images, and that's all it could output. Each run required substantial tweaking, and if the white balance was off, I'd need to go into Lightroom to fix the pages. Cleaning up a book often took more than an hour. Moreover, it's hard to find a version that runs on Apple Silicon, and since updates are by volunteers, they are sporadic at best. Even the complexity of running it was daunting. When a new version was released, I often had to go through a complex series of steps to build it from source code. Frequently, that failed, and I didn't know why.

Over the past week, I forked ScanTailor and substantially modernized it. I added PDF import and export—features I had long wished for—and updated it to run on Silicon Macs, taking advantage of new frameworks that exploit these chips' capabilities. I redesigned the interface and added algorithms to determine whether each page should be black-and-white, grayscale, or color, while keeping file sizes as small as possible. Now you give it a PDF and get a PDF back, often with no tweaking. What used to take me an hour takes minutes. I decided to get an Apple Developer Account so I could distribute releases as .dmg files, so anybody with a Mac could download and install the program. I'd be delighted if you could try it and share your feedback.

Clearly, the AI did not do this autonomously; I directed it, reviewed its work, and caught its errors. Errors are relatively frequent, but not a roadblock. I'm not a C++ programmer, but I have a sense for code from my early days and would likely have introduced just as many bugs myself—maybe more. Most critically, what would take a good programmer weeks takes an AI mere hours. Even better, as you learn how it works, you can run multiple instances at once.



Strange Weather, my first module for VCVRack

As I got better at using Claude Code and became more familiar with GitHub, I started other projects. I made Strange Weather, a module for the VCVRack music synthesis platform that generates modulation voltages from four different strange attractors. Last night, I began working on an iPhone app that turns ambient sounds into generative audio pieces, much like the late, lamented RJDJ did. And I have a portfolio/slideshow program for iPad in the works.

For the first time since the 1980s, I feel like I can do whatever I want—imagination is my only limit. Vibe coding is a bit like being a wizard, casting spells that make things happen. It's also a bit like being a hacker, tinkering with a system you don't fully understand. I have buried the lede in this story, but **vibe coding is the single biggest transformation since ChatGPT 3.5; it is one of the biggest since the dawn of computing.** Let me be clear: someone with a good sense of how tech works but very little modern coding knowledge can, within a few days, write pretty much any program they want, save for a AAA game. An age in which every mildly tech-savvy person has their own personal suite of programs is upon us.

There's an irony here. I argued earlier that the Mac took Clarke's dictum about technology being indistinguishable from magic as a design brief, and that this was a kind of disenfranchisement: the code was hidden, the user

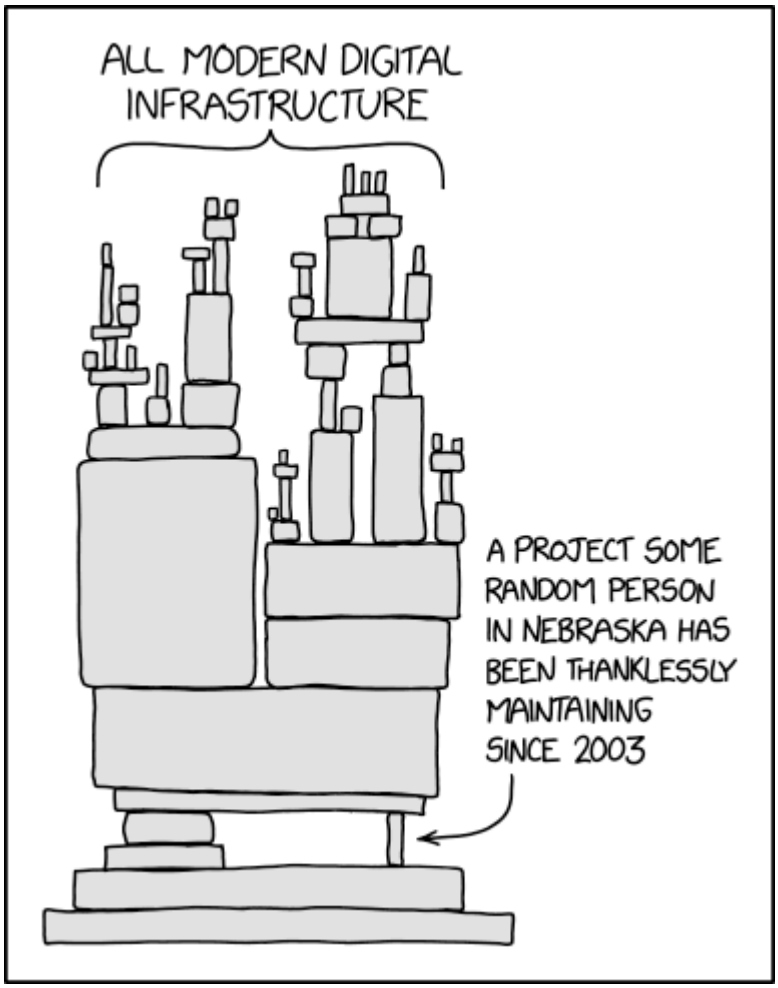
reduced to operator. Now the magic has flipped. Instead of consuming software I can't see inside, I am producing software, even if I don't know how it's created or what is in the code.

But I have anxieties about sharing this work, even with you, Internet friends. There is a lot of hatred of AI out there. And since I don't know the code, I don't see how it will break things. What would the original contributors think? I doubt the current maintainers of ScanTailor would ever want to merge my changes back into their version, nor would I advise them to do so. AIs, for now, often produce tangled "spaghetti" code, though I suspect this will improve dramatically over the next couple of years. But this brings us to the problem at the heart of this essay: the culture of open source and the transformation it will face in the very near future.

There is a vast landscape of open-source software on GitHub, millions of repositories, and it is literally what the internet runs on. The browser you're reading this in—Chrome, Firefox, Safari, Edge—is built on open-source code. So is Android. So are the servers that delivered this page to you, the databases that store your email, and the encryption that protects your passwords. cURL, a tool for transferring data that most people have never heard of, is embedded in billions of devices: cars, televisions, phones, and game consoles. A tiny utility called Log4j was running on millions of systems when a critical vulnerability emerged in 2021; the maintainers, who were volunteers, were blamed for the crisis.

This leads us to the "commons." The term comes from an old debate in economics. In 1968, Garrett Hardin argued that shared resources—such as common grazing lands, fisheries, and forests—were doomed to destruction. Each farmer benefits from adding one more cow to the pasture, but if everyone does, the pasture is destroyed. The 'tragedy of the commons' became an argument for privatization: only ownership creates the incentive to preserve. Elinor Ostrom spent her career proving Hardin wrong. Studying Swiss alpine meadows, she showed that commons could be sustainably managed without privatization, but only with careful governance: clear boundaries, shared rules, monitoring, and sanctions for violations. She won a Nobel Prize for this work in 2009.

Open source was supposed to be a new kind of commons, escaping the tragedy entirely. In contrast to earlier forms of Commons, my use doesn't diminish anyone else's. But this model didn't acknowledge that the scarce resource isn't the code—it's the effort of project maintainers. The tragedy of the digital commons is not overuse but abandonment: projects that rot when no one tends them, vulnerabilities that fester, dependencies that break. The xkcd comic about all modern digital infrastructure resting on "a project some random person in Nebraska has been thanklessly maintaining since 2003" is barely a joke.



xkcd, "Dependency," licensed under a Creative Commons Attribution-NonCommercial 2.5 License.

Most of those millions of repositories are dormant. Many never got anywhere in the first place. Still, there are those in which maintainers burn out, find other jobs, have children, or lose interest, leaving behind code that is freely licensed, fully documented (or at least commented), and with its complete history of changes preserved in version control. Anyone is legally permitted to copy it, modify it, or redistribute it. And yet, until recently, this permission was largely theoretical. For most people, the freedom to change code that you cannot understand is not practical. It is like being granted access to a library in a language you do not speak.

Most projects die when their maintainer walks away. Others grow large enough to develop their own kind of inaccessibility—and then die anyway. This is, perhaps, an even greater tragedy. Again, personal experience is the best way for me to describe this. In 2005, I evaluated three competing content management systems (CMS) for my website—WordPress, Joomla, and Drupal. All had been released in the previous few years, and initially, they were competitive in terms of market share. Joomla was the most popular system in Europe and, for a time, the most popular CMS globally, but it was too complicated and clunky for me. WordPress was primarily a blogging platform at the time, and I wanted to run a full website, so I settled

on Drupal. I used Drupal for 13 years and grew to increasingly hate it and, sadly, dislike the community that ran it. It comprised some seven hundred thousand lines of code, organized into subsystems so intricate that no single developer could understand the whole. To become even minimally proficient—not to master it, to work competently within it—requires years. Every time Drupal had a major update—usually every two years—my site would completely break, and the more features I tried to add, the longer it would take to repair. My site's layout had to be coded in PHP; the design was increasingly complex, but the leaders of the Drupal community insisted it was better for everyone. Nor could you sit still. As new updates rolled out, older versions were abandoned by the community and, lacking new security updates, became vulnerable to exploits. Learning how to update a Drupal site wasn't easy. The community wasn't welcoming to people who didn't contribute, and contributing was hard if you weren't already part of it. After Drupal released its eighth update, I was done with it. My friends who had developed sites with it were also glad to be rid of it. Joomla, I am told, had a similar trajectory. Today, WordPress powers over 40% of all websites on the Internet; Drupal and Joomla, which were genuine competitors when I made my choice, have collapsed to a combined 3% of the CMS market and are still falling.

In calling the Drupal community unwelcome, I don't mean to pick on it exclusively. I found this characteristic of open source culture. It seems paradoxical that something based on free, shared labor, something ostensibly outside of the capitalist system, would not welcome newcomers, but there are always structural reasons for cultures to evolve the way they do; you just have to look deeply enough. In the early days of shared computing, systems were fragile and resources scarce; a single careless user could quickly bring down a university machine. System administrators who kept these systems running adopted a defensive posture that soon turned into the "Bastard Operator From Hell"—a satirical figure from early-nineties Usenet—and embodied the archetype of the sysadmin who treated users as "lusers," sabotaged their work, hoarded knowledge, and enforced arbitrary rules with sadistic pleasure. The satire was all too recognizable. The culture that emerged—the hazing, the gatekeeping, the suspicion of anyone who hadn't paid their dues soon became constitutive of identity. Those who survived such a hazing themselves became invested in preserving difficulty as a mark of distinction. A recent blog post by Colin M. Strickland on Perl's decline offers a case study: the language had a "significant amount of … 'BOFH' culture, which came from its old UNIX sysadmin roots" as well as "Perl IRC and mailing lists [that] were quite cliquey and full of venerated experts and in-jokes, rough on naivety, keen on robust, verbose debate, and a little suspicious of newcomers." As Ostrom concluded, successful commons need boundaries, rules, and monitoring.

The hazing ensured that anyone modifying the commons understood what they were doing. The problem is that governance became identity, and the gates became ends in themselves.

Of course, one could always fork the open source code and develop one's own version, independent of the community. But in practice, forking was not easy. You inherited the full complexity of the codebase, the technical debt, and the implicit knowledge held only by the maintainers. Successful forks were rare, usually occurring only when a community was large enough to sustain parallel development—LibreOffice splitting from OpenOffice, Illumos from OpenSolaris. For a single user who wanted one thing to work differently, forking was not realistic.

With AI, the technical details of forking are shifting. These days, anyone can make changes to code—even individuals who wouldn't have dreamed of trying before. You don't need to understand the codebase anymore or even the programming language it is written in. As long as you know what you want to change, AI can help you figure out the rest. That's how I ended up rethinking ScanTailor and JPEGDeux.

But with this new freedom comes tension. On one side are the maintainers—people who have quietly kept projects running for years, sometimes decades. They earned their place by wrestling with complexity, pushing through the hard parts, and picking up knowledge that most never see. On the other side are users, now able to fork a project, make changes, or bring abandoned code back to life without any long apprenticeship or gatekeeping. These two groups are on a collision course.

I've come up with three possible outcomes; no doubt there are others. One is fragmentation: everyone keeps their own idiosyncratic forks, and improvements rarely make it back to the main project. In the past, the hassle and cost of splitting off would eventually lead successful forks to be merged into the parent repositories. But if AI lowers those costs, that pull toward the center weakens. A thousand flowers will bloom, but they won't cross-pollinate. Another, albeit unlikely, possibility is relief for maintainers: users who used to send feature requests and bug reports now handle their own issues, reducing the burden on exhausted volunteers. And then there's the bleakest option, at least for the idea of an open source community: maintainers get bypassed entirely, ignored by the vibe coders. The gate is still there, but nobody bothers with it. The years of volunteer work, the careful tending of a codebase, the hard-won knowledge—suddenly, none of it matters to someone who walks around it.

I don't know which of these futures will win out, or if we'll see all of them, depending on the project and the community. But I do know that the scale of this shift is massive. In the 1970s and early '80s, personal computers made it possible for regular people to own and use a computer without going

through institutions. In the 1990s, open source made it possible for anyone to read, copy, and modify code. But actually doing that still required skills most people didn't have. The legal freedom was there, but the practical freedom wasn't—and that gap stuck around for almost forty years. Now, AI coding tools close that gap. Suddenly, the end user can modify software. This is not a mere boost in programmer productivity—it is a fundamental shift in who gets to participate.

If the barrier to modifying software falls due to AI, the consequences for the open source community are vast. It seems unlikely that it will survive in its current form, but what will replace it is entirely unclear to me. To return to the question that is the title of this essay: what did vibe coding just do to the digital commons?