Pair Programming With Generative AI

Spinellis, Diomidis

DOI
10.1109/MS.2024.3363848

Publication date
2024

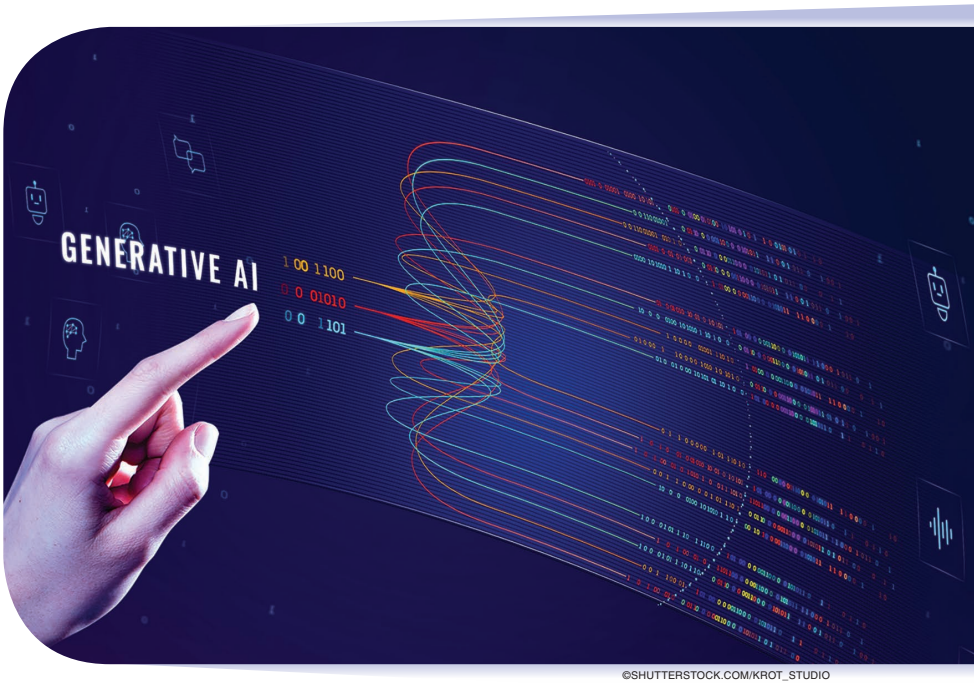
Document Version
Final published version

Published in
IEEE Software

Editor: Diomidis Spinellis
dds@aueb.gr

Pair Programming With Generative AI

Diomidis Spinellis



©SHUTTERSTOCK.COM/KROT_STUDIO

LARGE-LANGUAGE MODELS ARE transforming the way we work as software developers. Three key avenues are IDE assistants, such as GitHub Copilot, which can generate code for us; powerful cloud-based APIs (application programming interfaces) that we can use for summarizing, inferring, expanding, or transforming text; and interactive chatbots, such as ChatGPT, which can offer coding help and suggestions. A comprehensive overview of these technologies and their applications appeared in a recently published *IEEE Software* column.[1] Here, based on my experience from about 10,000 generative AI prompts covering both simple and advanced programming tasks, I provide an overview of how we can most effectively use generative AI to increase our productivity. While AI cannot fully assume the role of a pair programming[2] colleague, with appropriate prompting and care, it can provide extraordinary help on many fronts.

The Good

The power of generative AI in aiding development tasks comes from its vast training corpora and speed. It can thus suggest code in areas we don't know well or generate code faster than we can type.

The most useful application of generative AI in programming is in navigating APIs. After decades of effort, large-scale software reuse has now become a reality through open source software libraries and package managers that can easily install them.[3] Now, a remaining problem is to know which API to use and how to employ it. Here, AI, prompted with the action we want to perform, will swiftly generate the appropriate code. For example, consider the prompt: "Write Python command-line argument parsing code for two required string arguments: input-file and output-file."

Generative AI also works well in translating code from one programming language to another. This need arises when modernizing code from a legacy code base (say C to Rust) or when wanting to employ a useful code fragment written in a language different from that of our code base (for example, Python to JavaScript). In both cases, pasting the original code and prompting for its translation into the required language typically work wonders.

Related to this application is also AI's ability to transform data in ways that make them suitable for inserting into a program. For example, it can swiftly convert a comma-separated list of key-value pairs into code for

Digital Object Identifier 10.1109/MS.2024.3363848
Date of current version: 11 April 2024

0740-7459/24©2024IEEE

initializing a JavaScript Map. AI can even often generate data or transform between data representations. In one particularly notable example, it quickly created for me a dictionary mapping document metadata ISO (International Organization for Standardization) language codes (for example, "en") into the corresponding Python's *nltk* library language names ("english"), simply by prompting "Give me as a Python map the ISO code of each of the following languages mapped to its name: arabic chinese french,…" In another case, ChatGPT generated for me a map from LaTeX composite characters to the corresponding Unicode characters.

AI works well when generating code that is mostly boilerplate, such as configuration files for vanilla GitHub continuous integration, Docker build, or Sphinx documentation. These are tasks most of us perform only rarely and are therefore unlikely to be at our fingertips. Yet the corresponding code is straightforward and consequently well within the capabilities of generative AI.

AI can also help in troubleshooting. Prompted with code and a compiler error message, it will often give us the required fix. On occasion, it can also deal with bugs, though this mostly works with (in retrospect) obvious ones, such as forgotten brackets, misused APIs, or misplaced tokens. For this, the prompt is, again, the code and a description of the undesired behavior.

I have even seen AI suggest optimizations for improving the code performance ("Make the following code faster"), though this happens mainly when our understanding of the underlying costs is shaky to begin with. In the case where I witnessed it, the code involved misuse of Python's NumPy library.

AI can be a programming language assistant and tutor. Most modern languages are extraordinarily complex, and for many, it is almost impossible to know all their features and corresponding syntax. Prompting for the required functionality ("Pass Java method XYZ as a functor") will reliably generate the appropriate incantation. Conversely, AI can aid the comprehension of code idioms. Prompted with a short snippet, it can explain its code. In the same realm, AI can help us understand the rationale and functionality of specific constructs ("What are the advantages of Python's raise from exception"), sparing us the need to read a language's documentation end to end.

The Bad

For the uninitiated, generative AI can appear deceptively brilliant. Yet its answers can easily be stupidly wrong (ChatGPT has apologized to me for errors 162 times), while its use also carries other risks.

A common case is the so-called hallucinations, where the AI comes up with nonexistent facts. I've often been suggested nonexistent API function names, parameters, and program options. Typically, we can easily catch these as compilation, runtime, or program invocation errors. Because the code and the names look disconcertingly plausible, the way to reliably verify the error is a quick search in the corresponding reference documentation.

Another issue is erroneous code. If we are lucky, the code will fail to compile or run. In one case, I persistently got suggestions to use a Python's `cryptography.x509` library function rather than the `asn1crypto.x509` function with the same name. This function then generated data that triggered errors deep within the pyHanko PDF signature library in which they were used. With some luck, such errors may be fixed with revised AI prompts. In my case, these didn't work. (A sign that we have reached AI's limits is when a subsequent answer repeats what was already established to be wrong.) This happened to me; debugging the problem required a deep dive into pyHanko's source code and careful reading of the two cryptography libraries' reference documentation.

Outdated results are a further risk. Collecting data and training a large-language model on them currently require many months. It takes even longer for facts associated with new developments to appear in training data, such as suitably licensed open source software or Q&A forum discussions. Due to this time lag, AI may present us with out-of-date advice. For example, it may recommend deprecated APIs or fail to recommend the most modern approach to a challenge.

More troubling is erroneous code that runs but produces wrong results. Confusingly, sometimes the incorrect code may even be accompanied by correct output supposedly created from it. One cause of erroneous code is that currently, generative AI is woefully inadequate for anything that requires reasoning steps—that is, code that will implement some algorithmic thinking and not just a well-known algorithm. In one case, I persistently got completely bogus suggestions for a recursive SQL query, a syntax I admittedly have trouble internalizing. In the end, I had to craft the query from scratch on my own, which showed me the limits of generative AI, even for some simple tasks. Others have also documented AI generating code with security vulnerabilities, which are also often difficult to catch without being deeply familiar with the corresponding risks.

ABOUT THE AUTHOR

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business, Athens 104 34, Greece, and a professor of software analytics in the Department of Software Technology at the Delft University of Technology, 2600 AA Delft, The Netherlands. He is a Senior Member of IEEE. Contact him at dds@aueb.gr.

Finally, it is worth keeping in mind that when interacting with AI, our code or prompts leave our organization and get processed by the service's cloud infrastructure. In some domains, the potential leaks of personal data or intellectual property may be organizational, regulatory, or legal concerns.

And Avoiding the Ugly

Half a century ago, an MIT professor, Joseph Weizenbaum, was shocked to observe that ELIZA, a very primitive conversation program he had developed, drew people to become emotionally attached to it to the extent that some psychologists seriously suggested it could be used to automate psychotherapy.[4] The reason for this is that we humans tend to anthropomorphize the world around us as a shortcut for understanding it. This sets a dangerous trap when dealing with generative AI because we can easily forget that it will output only plausible-looking rather than correct or appropriate answers. Consequently, it is important for us developers to employ guardrails that reduce the risks of our AI interactions.

Our first lines of defense are our knowledge and ability to reason about the generated code. Lacking these skills and therefore treating the generated code as a black box is irresponsible and ethically wrong. We should read the provided code and understand what it is doing, or for elements that are not clear or obvious, be able to reach that understanding after reading the corresponding language or API reference documentation. Consequently, even when programming with generative AI, we need a solid knowledge of all applicable concepts appearing in the code, such as control and data structures, operators, functions, types, parameter passing methods, classes, interfaces, parameterized types, dynamic dispatch, exceptions, anonymous functions, closures, concurrency, and protocols. This knowledge is required both for comprehending generated code and for dealing with cases, such as those I discussed in the previous section, where the AI reaches a dead end.

Our next two lines of defense involve setting up our projects to follow established code hygiene practices: static analysis and automated tests. These are always important—but even more so when AI-generated code ends up in our software.

Start with static analysis tools. Does the proposed code pass through a linter? This is crucial in languages with dynamic typing, such as Python and JavaScript, where the interpreter cannot employ type checking to detect errors that are likely to cause runtime failures.

Continue with tests. These can be unit tests that exercise the provided code, demonstrating its functionality according to our expectations, or integration tests that demonstrate an emerging property we want to maintain, such as a given throughput. Even when we use generative AI to create the test code, the different prompts used for generating the main code from the prompts used for the tests and (hopefully) our verification that the tests actually check the code's intended properties and edge cases help ensure that plausible but incorrect code will not slip through.

Having described the benefits and risks and our defenses for generative AI in programming, it is easy now to also address the elephant in the room: the effect that AI can have on developers' jobs. Without even going into the realms of software requirements, architecture, product, and process management, which call for an, unattainable for AI, deep understanding of organizations, context, and intents, it is clear that even for coding tasks, generative AI is a powerful productivity booster for knowledgeable programmers rather than their replacement.

References

1. C. Ebert and P. Louridas, "Generative AI for software practitioners," *IEEE Softw.*, vol. 40, no. 4, pp. 30–38, Jul./Aug. 2023, doi: 10.1109/MS.2023.3265877.
2. K. Beck, and C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2003.
3. D. Spinellis, "Cracking software reuse," *IEEE Softw.*, vol. 24, no. 1, pp. 12–13, Jan./Feb. 2007, doi: 10.1109/MS.2007.9.
4. J. Weizenbaum, *Computer Power and Human Reason*. New Orleans, LA, USA: Pelican Books, 1984, pp. 5–7.