CCC and CRA-I Joint Whitepaper

April 2025

The Future of Programming in the Age of Large Language Models

Authors

Arjun Guha (Northeastern University) and Ben Zorn (Microsoft)

Contributors

Carolyn Jane Anderson (Wellesley College), Molly Q Feldman (Oberlin College), Sumit Gulwani (Microsoft), Jing Liu (University of Michigan), Erik Meijer (Independent Researcher), Nadia Polikarpova (University of California San Diego), Ufuk Topcu (University of Texas at Austin), Eran Yahav (TECHNION & Tabnine), and Lingming Zhang (University of Illinois Urbana-Champaign)

Reviewed by:

Gabrielle Allen (University of Wyoming)

Introduction

In a few short years, large language models (LLMs) have become more and more capable of performing a wide variety of tasks, including tasks that require expertise in software development. As a result, LLMs are now in widespread use by computing professionals; educators in computing are starting to teach students how to use LLMs; and researchers in academia and industry continue to advance their capabilities. The widespread adoption of these models has shifted both how we program software and how we teach the next generation of developers, and has called into question what the future of programming will look like.

In order to examine the research questions arising in relation to a future of programming with LLMs, the Computing Community Consortium (CCC) convened a series of roundtables with experts in LLM development, software development, and computing education. CCC catalyzes the computing research community by organizing workshops, creating white papers, and fostering discussions that describe policy and funding recommendations for future research directions. These virtual roundtable discussions ran for one hour each, with groups of between two and six external experts, not including the organizers. This roundtable series featured three separate roundtable discussions, with attendees separated into three groups: software developers, educators, and industry leaders focused on deploying these models. Each roundtable presented different sets of questions targeted towards each of these groups of

individuals and their areas of expertise. This whitepaper shares the challenges posited by the roundtable participants, as well as recommendations endorsed collectively by the group.

Large Language Models in Software Engineering

LLMs are a class of neural networks that are trained on large corpora of text. When the training corpus includes source code (which includes natural language comments), developer documentation, Q&A about software engineering tasks, and so on, an LLM can learn to bidirectionally translate between natural language and programming languages, generate documentation from code, translate between programming languages, generate test cases, help debug code, and more.

Professional software engineers have rapidly adopted LLMs. The first generation of LLM-powered tools were autocomplete tools that provided rapid, inline suggestions to developers as they write code (GitHub, 2021, Weiss and Yahav, 2013). These are now in widespread use, and several studies indicate that most professional developers find them helpful (Dunay et al., 2024, Vaithilingam et al., 2022, Ziegler et al., 2022). LLM-powered suggestions are flexible, and many developers quickly learn to use them in different ways, ranging from helping them write code faster ("saving keystrokes") to helping them rapidly understand unfamiliar tasks and explore unfamiliar APIs (Barke et al., 2023, Nam et al., 2024). Looking ahead, we anticipate that most software engineers will be expected to know how to leverage LLMs to accelerate their work.

Understanding and adapting code are essential skills. However, LLMs do not eliminate the need to read and write code. A well-known limitation of LLMs is that it is up to developers themselves to check that an LLM's response meets their expectations for code correctness, security, readability, and so on. In the most general case, the need to have developers vet LLM responses serves as an implicit "speed limit" on how fast an LLM can write code. Today, LLM-powered inline suggestions typically do not produce more than a few dozen lines at a time, which allows an experienced developer to vet LLM-generated code relatively quickly. However, a system that generates thousands of lines of code at once and requires hours of testing and code review would be unlikely to gain traction (Yahav, 2022). For the foreseeable future, we expect that software engineers will still need to be able to read and adapt LLM-generated code. In fact, being able to rapidly read and understand code is more important than ever.

LLMs can help software engineers understand code better. Nevertheless, there is significant research and development on systems and interfaces to facilitate vetting LLM-generated code. One approach is to highlight regions of LLM-generated code that are more likely to require

review than others (Vasconcelos et al., 2024). Another approach is to use LLMs that explain their reasoning, which many conversational models do, or engage in discussion with an LLM about generated code (Ross et al., 2023). Finally, an interface that shows developers how the generated code manipulates program values (Ferdowsi et al., 2024) can help them understand LLM outputs faster. The latter work points out two problems: a developer may fail to detect a problem in LLM-generated code, or may discard LLM-generated code that is in fact correct. Interfaces for vetting aim to address both problems. Moreover, an LLM can also explain human-written code, and help a developer more quickly understand an unfamiliar codebase. We believe there is much more research and development needed on how software engineers interact with a variety of LLM-powered tools to fully realize the gains that are possible.

Certain software engineering tasks are very amenable to LLM automation. For example, an LLM can write developer documentation. Documentation that is slightly wrong or needlessly wordy, is likely better than no documentation at all. Software testing, which takes up a significant portion of software development time, is often amenable to LLM-based automation (Deng et al., 2023). There are established metrics for test suite quality that can be applied to LLM generated tests (DeMillo et al., 1978). Moreover, it is possible to build systems that use an LLM in a loop to automate the process of discarding failing tests and building a diverse set of tests (Schafer et al., 2024). Moving beyond test generation, LLM-powered systems can also accelerate the debugging process (Bajpai et al., 2024). LLM-powered test generation is starting to be deployed at scale (Alshahwan et al., 2024), and we expect the technology to become more widely available.

Data science tasks are being automated by LLMs. A data analysis or data cleaning task may require writing code, but typical errors may be obvious from spot-checking the output. The developer can also test LLM-generated code using test datasets that they are intimately familiar with, before applying it to novel data. LLMs are already having a significant impact on these kinds of data science tasks. For example, significant effort is going into the problem of generating SQL queries from natural language, with progress driven by benchmarks that cover dozens of domains (Li et al., 2023). LLMs are also proficient with widely used Python data processing and graph plotting libraries (Lai et al., 2023). This makes it possible to build tools such as ChatGPT Data Analysis (Agarwal et al., 2023), which uses an LLM to generate code, executes it in a sandboxed environment, automatically displays charts and tables, and supports several file formats. The user can inspect LLM-generated code if needed, and the system automatically prompts the LLM to revise its code when certain errors occur. LLMs are likely to have significant impact on programming tasks where the code is secondary and code output is amenable to spot-checking.

There is also significant interest in using LLMs to solve open-ended software development tasks. Towards this goal, there is work on LLM-powered systems that try to solve real-world bug reports and have to navigate repositories with several hundred thousand lines of code (Jimenez et al., 2023, Xia et al., 2024). The systems that try to tackle these tasks use LLMs in several parts of complex pipelines that may involve automatically interacting with the terminal, examining the results of test execution, or browsing documentation (GitHub Next, 2024, Wang et al., 2024, Xia et al., 2024). Although the stated goal of some of this work is to develop "AI software engineers", there is a significant gap between what these systems and human engineers can do. For example, new software engineers are onboarded to understand existing processes and unwritten expectations; they ask questions when requirements are unclear; and they communicate about their work at different levels of abstraction for different audiences. Although these technologies are at a very early stage, we expect these "software engineering agents" to grow more capable, especially as LLMs become cheaper to operate.

LLMs are rapidly becoming cheaper to operate and can now run locally. Some of the more sophisticated applications discussed above are costly because they require several rounds of interaction with LLMs. Whereas the frontier of model development appears to involve developing larger models, there is also a significant amount of effort being put into making smaller models more capable. Research groups are rapidly releasing smaller models that are as capable as slightly older models that were orders of magnitude larger (Ben Allal et al., 2024, Gunasekar et al., 2023). There is also significant work being put into making LLM inference more efficient (Dao et al., 2022, Kwon et al., 2023), which makes both large and small models more cost effective. Both PC and smartphone operating systems are now being released with support for Small Language Models (SLMs) that run on-device, which helps allays concerns about data sharing. We can expect new use-cases to emerge as the cost of LLM inference continues to fall and the capabilities of SLMs improve.

Large Language Models in Computing Education

Some courses are being reimagined to leverage LLMs. Instructors have many perspectives on when and how to use LLMs in the computing curriculum (Lau and Guo, 2023). Many instructors need to carefully consider how student use of LLMs fits with the goals of their course, especially if unrestricted use of an LLM may circumvent student learning. On the other hand, they also present an opportunity to rethink course goals, including the goals of a CS1 course (Vadaparty et al., 2024). A course that allows students to use LLMs can be more aligned with software engineering practice, may allow students to build more sophisticated software, and may allow the course to cover content that would otherwise not be possible. Moreover, today's state-of-the-art LLMs can perform certain teaching tasks, such as giving feedback on buggy code or grading simple programs, nearly as well as expert teaching assistants (Phung, et al.

2024). We should expect that many students will enter college with some prior experience using LLMs. As a result, courses cannot ignore them but will have to explicitly address how students should use LLMs to support learning, and how to leverage non-chat modalities that are common in software development.

Students need to be taught how to use LLMs. It is clear that teaching students to successfully work with an LLM requires emphasizing a different set of skills than a traditional CS1 course. In a traditional course, students spend a significant amount of time learning how to write code, which includes learning syntax, learning to debug errors, learning good style, and so on. A student with an LLM may be able to spend less time on these code writing topics, but now needs to learn other skills, such how to read and evaluate code they did not write; how to decompose large problems into smaller problems that an LLM can reliably solve; and how to test code for correctness. These are fundamental programming skills that all computing professionals need, but many CS1 courses have historically not had the time to cover them in depth, and these concepts are instead introduced later in most curricula.

It is clear that programming with an LLM requires technical skills. LLMs offer a natural language interface to computer programming, but without training, many students struggle to prompt them to write code (Nguyen et al., 2024). Writing prompts requires technical, software development skills (Guy et al. 2024), but the natural language interface to LLMs can leave novice programmers with an illusion of competence, which requires careful scaffolding to overcome (Prather et al., 2024).

For many computer science students, being able to write code without LLM assistance remains important for a career in software engineering. There are enough specialized domains where LLMs are not known to be effective, and employers that do not allow their developers to use LLMs today. On the other hand, many students learn to code as preparation for careers that are not in software engineering. Many of these students learn to code in classes for non-majors, summer programs, or are self-taught. These students typically do not write production software that is deployed to customers. Their coding may be limited to writing scripts to analyze data or making modest changes to existing code. Many institutions have developed interdisciplinary curricula to serve these students (Barr et al., 2024), and LLMs have the potential to significantly reduce the friction that comes from trying to integrate teaching computing with other fields. There is more research needed to better understand the impact of LLMs on computing education, and how curricula should adapt so that the technology can be beneficial to all students.

The capabilities of LLMs continue to expand. In every educational context, there are challenges with using LLMs effectively. A fundamental problem for educators is to teach students an

effective mental model of LLM technology, so that they know when the LLM can and cannot help them. This mental model is difficult to develop because the technology is changing very rapidly. Moreover, researchers fundamentally have an incomplete picture of the capabilities of LLMs: new capabilities and limitations are sometimes discovered in models months after they are developed. The rapid pace of LLM development underscores the need for investing in curriculum development to keep students current.

Students and educators need to have access to LLMs. Without affordable access to LLMs, neither students nor educators can learn to use them effectively. The most capable LLMs today require subscriptions that may be unaffordable for many students. Fortunately, both free cloud-hosted and locally-hosted LLMs continue to improve significantly. Local hosting offers stability: an educator can choose when to upgrade to a better LLM, or even deliberately use an older LLM to avoid continuously adjusting their curriculum. However, local hosting requires cutting-edge hardware, which can be more expensive than commercially hosted APIs. Educators and university administrators need to understand the tradeoffs between the different ways to access LLMs.

Conclusion

LLMs have already had a significant impact on how software is developed in practice, and students need to be prepared for careers that require expertise in LLM technology. The faculty who train students for careers in software may benefit from professional development programs to help keep up with these changes in software engineering.

How software engineers interact with LLMs is also rapidly evolving. Researchers are developing increasingly capable multimodal LLMs that can, for example, reason about graphs or screenshots. Researchers are also developing new interaction modes that go beyond the traditional chat interface and text autocomplete.

These rapid advances can make it hard to design long-lasting curricula. Although commercial providers rapidly update their platforms to the latest models, education will benefit from the availability of good LLMs that remain unchanged for a few semesters. A platform that guarantees stability would allow faculty to iteratively develop course material.

However, faculty and students should also be aware of the frontier of LLM-powered technology. Unfortunately, the most capable models and LLM-powered tools require subscriptions that may not be universally affordable. Institutions and technology companies should work to ensure that all computing students can learn to leverage these technologies. One way to keep up with the frontier of progress in LLMs is to facilitate deeper communication, exchanges, and joint appointments between academia and industry.

References

Sandhini Agarwal, Ilge Akkaya, Valerie Balcom, Mo Bavarian, Gabriel Bernadett-Shapiro, Greg Brockman, Miles Brundage, Jeff Chan, Fotis Chantzis, Noah Deutsch, Brydon Eastman, Atty Eleti, Niko Fe lix, Simon Posada Fishman, Isa Fulford, Christian Gibson, Joshua Gross, Mike Heaton, Jacob Hilton, Xin Hu, Shawn Jain, Joy Jiao, Haozhun Jin, Logan Kilpatrick, Christina Kim, Michael Kolhede, Andrew Mayne, Paul McMillan, David Medina, Jacob Menick, Andrey Mishchenko, Ashvin Nair, Rajeev Nayak, Arvind Neelakantan, Rohan Nuttall, Joel Parish, Alex Tachard Passos, Adam Perelman, Filipe de Avila Belbute Peres, Vitchyr Pong, John Schulman, Eric Sigler, Natalie Staudacher, Nicholas Turley, Jerry Tworek, Ryan Greene, Arun Vijayvergiya, Chelsea Voss, Jiayi Weng, Matt Wiethoff, Sarah Yoo, Kevin Yu, Wojciech Zaremba, Shengjia Zhao, Will Zhuk, and Barret Zoph. 2023. ChatGPT Plugins. https://openai.com/index/chatgpt-plugins/.

Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the ACM International Conference on the Foundations of Software Engineering* (FSE). 185–196.

Yasharth Bajpai, Bhavya Chopra, Param Biyani, Cagri Aslan, Sumit Gulwani, Dustin Coleman, Chris Parnin, Arjun Radhakrishna, Gustavo Soares. 2024. Let's Fix this Together: Conversational Debugging with GitHub Copilot. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).*

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages (PACMPL)* 7, OOPSLA (2023).

Valerie Barr, Carla E. Brodley, Elsa L. Gunter, Mark Guzdial, Ran Libeskind-Hadas, and Bill Manaris. 2024. CS + X: Approaches, Challenges, and Opportunities in Developing Interdisciplinary Computing Curricula. Association for Computing Machinery, New York, NY, USA.

Loubna Ben Allal, Anton Lozhkov, and Elie Bakouch. 2024. SmolLM - Blazingly Fast and Remarkably Powerful. https://huggingface.co/blog/smollm.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Re. 2022. FlashAttention: Fast and ´ Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*.

R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, Lingming Zhang. 2023. Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *International Symposium on Software Testing and Analysis (ISSTA)*.

Omer Dunay, Daniel Cheng, Adam Tait, Parth Thakkar, Peter C. Rigby, Andy Chiu, Imad Ahmad, Arun Ganesan, Chandra Maddila, Vijayaraghavan Murali, Ali Tayyebi, and Nachiappan Nagappan. 2024. Multi-Line AI-assisted Code Authoring. In *International Conference on the Foundations of Software Engineering (FSE)*.

Kasra Ferdowsi, Ruanqianqian (Lisa) Huang, Michael B. James, Nadia Polikarpova, and Sorin Lerner. 2024. Validating AI-Generated Code with Live Programming. In *CHI Conference on Human Factors in Computing Systems (CHI)*.

GitHub. 2021. GitHub Copilot: Your AI Pair Programmer. https://github.com/features/copilot/. GitHub Next. 2024. Copilot Workspace. https://next.github.com/projects/copilot-workspace.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio Cesar Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harki rat Singh Behl, Xin Wang, Sebastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks Are All You Need. arXiv:2306.11644 [cs]

Tommy Guy, Peli de Halleux, Reshabh K Sharma, and Ben Zorn. Prompts are Programs. https://blog.sigplan.org/2024/10/22/prompts-are-programs/. 2024. ACM SIGPLAN PL Perspectives

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *International Conference on Learning Representations (ICLR)*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery, New York, NY, USA, 611–626.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. In *International Conference on Machine Learning (ICML)*.

Sam Lau and Philip Guo. 2023. From ”Ban It Till We Understand It” to ”Resistance Is Futile”: How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools Such as ChatGPT and GitHub Copilot. In *ACM Conference on International Computing Education Research (ICER)*, Vol. 1. Association for Computing Machinery, New York, NY, USA, 106–121.

Tung Phung, Victor-Alexandru Pădurean, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generative AI for Programming Education: Benchmarking ChatGPT, GPT-4, and Human Tutors. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 2 (ICER '23), Vol. 2*. Association for Computing Machinery, New York, NY, USA, 41–42.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs. In Neural Information Processing Systems Track on Datasets and Benchmarks (NeurIPS Datasets and Benchmarks).

Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In IEEE/ACM 46th International Conference on Software Engineering (ICSE). Association for Computing Machinery, 1–13. https://doi.org/10.1145/3597503. 3639187

Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How Beginning Programmers and Code LLMs (Mis)Read Each Other. In ACM Conference on Human Factors in Computing Systems (CHI). Association for Computing Machinery.

James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A. Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. 2024. The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers. In *ACM Conference on International Computing Education Research (ICER)*. Association for Computing Machinery, New York, NY, USA, 469–486.

Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. In *International Conference on Intelligent User Interfaces (IUI)*. Association for Computing Machinery, New York, NY, USA, 491–514.

Max Schafer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. Adaptive Test Generation Using a Large Language Model. IEEE Transactions on Software Engineering (TSE) 50, 1 (2024). https://doi.org/10. 1109/TSE.2023.3334955

Annapurna Vadaparty, Daniel Zingaro, David H. Smith IV, Mounika Padala, Christine Alvarado, Jamie Gorson Benario, and Leo Porter. 2024. CS1-LLM: Integrating LLMs into CS1 Instruction. In Innovation and Technology in Computer Science Education (ITiCSE). Association for Computing Machinery, New York, NY, USA, 297–303. https://doi.org/10.1145/3649217.3653584

Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In CHI Conference on Human Factors in Computing Systems. ACM, New Orleans LA USA, 1–7. https://doi.org/10.1145/ 3491101.3519665

Helena Vasconcelos, Gagan Bansal, Adam Fourney, Q. Vera Liao, and Jennifer Wortman Vaughan. 2024. Generation Probabilities Are Not Enough: Exploring the Effectiveness of Uncertainty Highlighting in AI-Powered Code Completions. ToCHI (Aug. 2024).

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents. https://doi.org/10.48550/arXiv.2407.16741 arXiv:2407.16741 [cs]

Dror Weiss and Eran Yahav. 2013. Tabnine: AI Assistant for Software Developers. https://www.tabnine.com/.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM based Software Engineering Agents. arXiv:2407.01489 [cs]

Eran Yahav. 2022. Which Should Now Be (Obviously!) Recasted as the “Fundamental Theorem of GenAI”: Every day is a good day to remind ourselves of the fundamental “theorem” of program synthesis: Synthesis is useful if the cost of specification + cost of consuming result << cost of writing the code yourself. https://x.com/yahave/status/1660729517210476544.

Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sit tampalam, and Edward Aftandilian. 2022. Productivity Assessment of Neural Code Completion. In ACM SIGPLAN International Symposium on Machine Programming (MAPS) (MAPS 2022). Association for Computing Machinery, New York, NY, USA, 21–29. https://doi.org/10.1145/3520312.3534864

Suggested Citation

Guha, Arjun & Zorn, Ben. *The Future of Programming in the Age of Large Language Models*. Washington, D.C.: Computing Research Association (CRA), April 2025. https://cra.org/wp-content/uploads/2025/04/the-future-of-programming-in-the-age-of-large-language-models_April-2025.pdf

About the Computing Community Consortium (CCC)

A programmatic committee of the Computing Research Association (CRA), CCC enables the pursuit of innovative, high-impact computing research that aligns with pressing national and global challenges. Of, by, and for the computing research community, CCC is a responsive, respected, and visionary organization that brings together thought leaders from industry, academia, and government to articulate and advance compelling research visions and communicate them to stakeholders, policymakers, the public, and the broad computing research community.

About CRA-Industry (CRA-I)

A standing committee of the Computing Research Association (CRA), CRA-Industry (CRA-I) convenes industry partners on computing research topics of mutual interest and connects them with CRA's academic and government constituents to advance shared goals and improve societal outcomes. Of, by, and for the computing research community, CRA-I recognizes the diversity of companies engaged in computing research and fosters open dialogue across sectors. Through these conversations, CRA-I identifies emerging trends, develops best practices, and produces whitepapers and reports that strengthen the computing research ecosystem and drive innovation benefiting industry and society alike.

This material is based upon work supported by the U.S. National Science Foundation (NSF) under Grant No. 2300842.

The views and conclusions expressed are those of the authors and the Computing Research Association (CRA) and do not necessarily reflect the views of NSF or the authors' affiliated institutions.