Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Perspective:

Software development in the age of intelligence: embracing large language models with the right approach

Xin PENG

School of Computer Science, Fudan University, Shanghai 200438, China

E-mail: pengxin@fudan.edu.cn

Received Aug. 8, 2023; Revision accepted Sept. 25, 2023; Crosschecked Oct. 26, 2023

https://doi.org/10.1631/FITEE.2300537

The emergence of large language models (LLMs), represented by ChatGPT, has had a profound impact on various fields, including software engineering, and has also aroused widespread concerns. To see a right way through the fog, we have recently been discussing and contemplating a theme of "software development in the age of LLMs," or rather "the capability of LLMs in software development," based on various technical literature, shared experiences, and our own preliminary explorations. Additionally, I have participated in several online interviews and discussions on the theme, which have triggered further insights and reflections. Based on the aforementioned thinking and discussions, this article has been composed to disseminate information and foster an open discussion within the academic community. LLMs still largely remain a black box, and the technology is still rapidly iterating and evolving. Moreover, the existing cases reported by practitioners and our own practical experiences with LLM-based software development are relatively limited. Therefore, many of the insights and reflections in this article may not be accurate, and they may be constantly refreshed as technology and practice continue to develop.

ORCID: Xin PENG, https://orcid.org/0000-0003-3376-2581

1 Impact of large language models

The wave of enthusiasm surrounding ChatGPT (based on GPT-3.5) has yet to subside when suddenly GPT-4 made its debut, bringing about a comprehensive impact across all domains (OpenAI, 2023; Zhao et al., 2023). In the field of software engineering, both the academic and industrial communities have been astounded by the all-encompassing capabilities of LLMs in software development. Numerous technical experts have enthusiastically harnessed the power of LLMs to tackle a myriad of development tasks. These tasks encompass requirement analysis, software design, code implementation, software testing, code refactoring, defect detection, and repair, spanning each stage of the software development process illustrated in Fig. 1 (Dou et al., 2023; Du et al., 2023; Hou et al., 2023; Liu et al., 2023; Wang et al., 2023; Yuan et al., 2023a, 2023b; Zheng et al., 2023). Some have even attempted end-to-end software application development using LLMs (Wu et al., 2023). The results have been nothing short of astonishing, to the extent that many have exclaimed, "programming is about to be terminated" or "programmers will be laid off in large numbers." Consequently, there have been numerous assertions of a "new era in software engineering;" some call it Software Engineering 3.0, while others refer to it as Software 2.0, signifying that software development is poised to enter a fully intelligent new era.

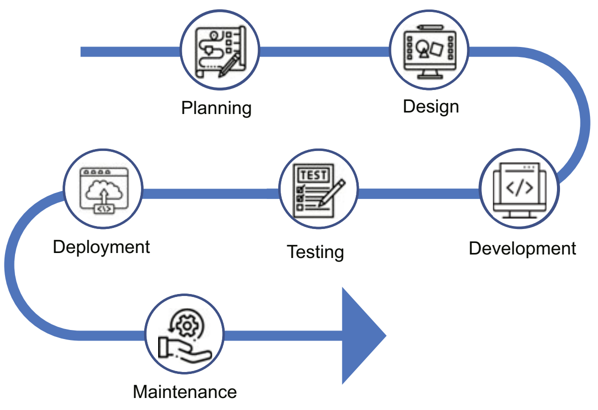

Fig. 1 Overview of the software development life-cycle

The powerful capabilities of LLMs in software development are undoubtedly undeniable, and their disruptive impact on the field can be foreseen. In other words, it is certain that LLMs will drive software development into a new era of intelligence. However, what remains uncertain is what this new era will look like. What kind of changes can be expected in the foreseeable future regarding software development approaches? Which software development roles will disappear, and which new roles will emerge? Both the academic and industrial communities are perplexed and anxious because the road ahead is unclear. For instance, Matt Welsh, former professor of computer science at Harvard University and director of engineering at Google, predicted that generative artificial intelligence (AI) will lead to the end of programming within three years, leaving only two roles for humans in software development: product manager and code reviewer (Welsh, 2023). If his prediction comes true, many research directions and related job positions in the current field of software engineering, such as software architecture and software maintenance, would become unnecessary.

2 Calm reflection

ChatGPT (and other LLM-based tools) has been widely accepted as an effective intelligent assistant in various software development tasks (Hou et al., 2023). This reminds me of the research I conducted 15 to 20 years ago on software reuse and software product lines. While achieving local and partial reuse through code copy–paste and application program interface (API) calls is relatively straightforward, the real challenge lies in systematic reuse-based software development that takes into account the requirements and design (as promised by software product lines). Similarly, using ChatGPT for intelligent assistance in local software development tasks, such as code snippet generation and technical queries, is not difficult. The real challenge lies in harnessing it for end-to-end, systemic generative development.

In this regard, some industry experts have explored this area and shared their experiences on various platforms. From these explorations, we can observe that for common small-scale software applications like Tetris or Snake, ChatGPT can gradually generate complete executable programs through natural language interactions when properly guided, producing code with reasonably high quality. This leads to the question of whether this end-to-end generative software development capability can be extended to larger, more complex enterprise software projects. It reminds me of Brooks' discourse on the fundamental difficulties (essence) and accidental difficulties (accident) in software development in his classic work "No Silver Bullet Essence and Accidents of Software Engineering" (Brooks, 1987). The essence of difficulties lies in conceptualizing complex abstract software entities, while the programming language representation of these entities is only an accidental difficulty. Most of the progress in software engineering over the past few decades has been in dealing with accidental difficulties, while fundamental difficulties (primarily centered around requirements and design) have seen little advancement.

End-to-end generative software development clearly requires overcoming the two major challenges of requirement analysis and software design. How capable is ChatGPT in these areas? Based on the current shared experiences, it appears that ChatGPT does possess some analysis and design capabilities, including:

1. automatically generating detailed requirements based on high-level requirements and organizing them into items,
2. generating standardized representations of requirements, such as unified modeling language (UML) sequence diagrams,
3. automatically generating design structures

consisting of multiple classes,

4. incrementally generating application code based on developer prompts and understanding the existing code structure,

5. automatically refactoring code based on general design principles to enhance comprehensibility and scalability, and

6. generating database schemas and corresponding SQL statements based on requirements.

Can ChatGPT really perform analysis and design in software development? Based on the practical experiences shared by various experts, it is not accurate to say that it cannot. However, the attempted software applications are relatively small in scale (e.g., two or three hundred lines of code) with simple requirements (e.g., Tetris games and library management) or even readily available code and documents for similar applications online. In addition, to enable ChatGPT to generate complete code through dialogic interactions, developers need to possess strong communication and guidance skills, consistently expressing requirements clearly, decomposing development tasks into a series of well-defined steps, and guiding ChatGPT accordingly. For example, in a case shared by an industrial expert (https://mp.weixin.qq.com/s/sUMt9oyASUU0eO9jlCurEw), the implementation process of a simple Tetris game was broken down into 10 tasks: create a game framework, replace the console with a graphical user interface (GUI), display an L-shaped block on the GUI, move the block and perform collision detection, rotate the block, let the block fall and create a new block, check for game over, add a line elimination feature, include all types of blocks, and add scoring capability. This task breakdown reflects a very rational process of incremental and iterative development, where subsequent tasks depend on the completion of previous tasks, and each task's result can be confirmed (e.g., the program is runnable). Although this task breakdown was also inspired by ChatGPT's output, developers still need to grasp the overall iterative process, and design considerations are incorporated. Furthermore, developers need to continuously check ChatGPT's output, promptly identifying errors and executing corrections. Particularly, in the requirement analysis process, ChatGPT's requirement refinement relies mainly on common sense or general features of certain software, which may overlook some critical requirements, and innovative or personalized requirements need to be supplemented by developers.

Let us discuss some possible limitations of the current application of LLMs in the field of software engineering, considering their training approach and the essence of software development. These limitations are not specific to any particular stage; instead, they span various stages in the software development lifecycle, including planning, design, development, testing, deployment, and maintenance.

1. Scale and complexity of software development may limit the capabilities of LLMs from both human and machine perspectives.

First, the generative development process heavily relies on developers' step-by-step guidance of the LLM, which requires developers have a complete and in-depth understanding of the entire software design and implementation process. Only then can they break down tasks in a reasonable manner and guide the LLM to generate code progressively. When a software project reaches tens of thousands, hundreds of thousands, or even millions of lines of code, the human brain may no longer be able to fully grasp the entire code generation process. Additionally, for large-scale complex software systems, task breakdown and implementation are not strictly sequential; instead, they involve multiple parallel threads with continuous forking and joining. In traditional software development, developers need to decompose large-scale complex systems into multiple layers, with different teams and individuals responsible for different levels and sections of work. The development team needs to communicate, coordinate, and adjust design plans as needed. This process is challenging to replace with LLMs. Furthermore, LLMs themselves have limitations in their ability to fully comprehend the global development process of complex systems. For instance, a comparative study by our team (https://mp.weixin.qq.com/s/GMMjF9sDv0c31AoRTXSIYA) found that even in the small-scale software generation process, ChatGPT might "forget" previously generated code (e.g., inconsistent method names or method return values). This suggests that LLMs may have difficulty in generating code for large-scale software implementations. Similarly, while LLMs can excel in generating unit tests for individual functions, their performance in generating system-level tests for complex systems is less than ideal. Finally, the existence of manual code review underscores the ongoing significance of software design and code quality. Even the most optimistic estimates acknowledge that code generated by LLMs still requires human review. As a result, for large-scale software systems, principles such as modularity, information hiding, separation of concerns, and code comprehensibility remain essential. Otherwise, developers responsible for reviewing the code may struggle to understand the code generated by LLMs, and become bottlenecks in the software development process.

2. LLMs may lack abstract thinking capabilities and may exhibit limitations in precision.

Software design, especially high-level architecture design, often involves complex abstract thinking. Some design abstractions have clear layer-by-layer refinement, such as gradually refining interface designs into concrete implementations. However, there are also design abstractions that involve global system abstractions, often including complex trade-off decisions, such as selecting software architecture styles and patterns. These complex design abstractions may not be the strong suit of LLMs. The training approach of LLMs makes them proficient in generating relevant content based on flattened context and prompt information. They can achieve creative and associative effects through fine-grained (e.g., word-level) flexible combinations, but they may lack the corresponding comprehension and application abilities for extensive abstract design decisions. Additionally, the probabilistic nature of LLMs conflicts with the precision sought in software development. Therefore, while LLMs can often achieve 80%–90% accuracy in local coding tasks, they require developers to promptly identify issues for alert, correction, or even direct modification and supplementation.

3. In software requirements and design, there exists a significant amount of tacit knowledge that is difficult to capture.

The success of LLMs largely comes from learning existing Internet text corpora (including code) and professional books and materials. In contrast, much of the knowledge related to software development requirements and design does not have explicit written records. It might exist in the minds of developers (including architects), on whiteboards, or within discussions from project meetings. Even if the development team provides detailed requirements and design documents, it is generally understood that not all the important information about requirements and design can be found in the documents.

Furthermore, there are often numerous comparisons and debates concerning requirements and design decisions, and these decision-making processes are typically not explicitly recorded. While one could argue that LLMs can learn requirements and design knowledge given sufficient data, these tacit knowledge aspects are challenging to capture, and meeting the prerequisite of sufficient data might be difficult.

Even if we try to record such information through photographs or meeting minutes, its high level of abstraction or ambiguity (e.g., the meaning of a certain element in a design and its impact on code implementation) might make it hard for LLMs to learn and apply this knowledge.

4. There are doubts about the long-term maintenance and support capabilities of LLMs for complex software systems.

Enterprise software systems generally have long lifecycles, during which they may undergo modifications for various reasons such as changing requirements, evolving usage environments, fixing bug, improving performance, and customizing for different clients. These software maintenance and evolution processes entail many software engineering challenges and issues.

To achieve intelligent development support for software systems, LLMs cannot be a one-time solution (i.e., only responsible for initial code generation). Instead, they need to be able to handle various tasks related to functionality extensions and code modifications throughout the extended software maintenance and evolution processes. This requires the LLMs understand the various requirements and design solutions already implemented in the code, as well as the precise correspondence between requirement/design elements and code elements. Additionally, the models need to be aware of and manage the interaction relationships between code modifications and different parts of the system (e.g., direct or indirect impacts caused by modifications). Furthermore, the code generated by LLMs may contain many repetitive segments, and the long-term maintenance, especially consistency modifications, of these code clones could become burdensome.

3 Embracing large language models with the right approach

The remarkable performance of LLMs in some programming tasks has brought excitement to many, and it has also fostered optimistic prospects for the disruption of software engineering and the realization of comprehensive generative software development. However, I believe that when discussing the software development capabilities of LLMs, we must first differentiate the types of software being developed. For small-scale software applications or even applications suitable for end-user programming tasks, it is entirely possible to use LLMs for end-to-end code generation. However, for large-scale and complex software systems, achieving end-to-end code generation based solely on the given requirements is not yet feasible.

Embracing LLMs is undoubtedly a correct and even necessary direction for software enterprises to improve their quality and efficiency. In fact, if information security risks are not a concern, directly using LLMs like ChatGPT in the enterprise development process can significantly enhance development productivity and quality. However, to achieve comprehensive and systematic software intelligent development, there is still a lot of foundational work to do, and there are several key issues that need to be explored. Here are some related considerations and suggestions, corresponding to the four limitations outlined in Section 2.

1. Solidly building a digitized and knowledge-driven foundation for software development.

Software development provides digital solutions for various industries, yet the level of digitization within software development itself is often quite low. For instance, common software assets such as public components are not effectively organized or reused, and the phenomenon of reinventing the wheel (repeatedly implementing the same functionality) is widespread. The cause and effect of each code modification are often unclear, and development tasks that trigger code changes (e.g., feature implementation or bug fixes) and the impact of code modifications (e.g., introducing code issues or changes in metrics) lack systematic records. Additionally, the description of requirements and design knowledge embedded within the software lacks explicit documentation and clear mapping to the code, leading developers to frequently rethink the same problems. In such cases, expecting LLMs to directly provide transformative and comprehensive intelligent development experiences might be unrealistic. The correct approach may be to solidly build a digitized and knowledge-driven foundation for software development, and then to combine it with LLMs to achieve more systematic intelligent development support.

For instance, this digitized and knowledge-driven foundation could involve creating a library of domain-specific common components and establishing a descriptive system, implementing a comprehensive tracking and management system for software code clones and the software supply chain, establishing tracking relationships among development tasks (e.g., feature implementations and bug fixes), developers, and code submissions and modifications, and establishing and maintaining high-level design descriptions and their mapping to code units. These digitized and knowledge-driven efforts can themselves enhance the quality and efficiency of software development. LLMs can provide technical and domain-specific knowledge to support the digitization and knowledge-driven aspects of software development, while also serving as a powerful means to integrate information and knowledge within these digitized and knowledge-driven platforms.

2. Placing greater emphasis on fundamental software engineering capabilities such as requirements, design, and validation.

I agree with the viewpoint expressed by Bertrand Meyer in his blog post at *Communications of the ACM* (Meyer, 2023). He suggested that LLMs like ChatGPT will not bring about the end of programming but instead revive some good old mainstays of software engineering, such as requirement analysis, precise specification, and software validation (including dynamic testing and static analysis). These traditional software engineering techniques have the potential to rejuvenate in the era of LLMs, but it may require careful consideration that how to integrate them organically with data-driven LLM technologies. As mentioned earlier, there is also a need to strengthen the digitized and knowledge-driven infrastructure in areas such as requirement analysis, design, and testing.

3. Exploring an intelligent interactive engine that effectively integrates LLMs, developers, and various tool capabilities.

The current situation in the field of software development is similar to the challenges addressed by Gartner's recent promotion of hyperautomation. There are many automation tools (such as debugging and testing tools, compilation and building tools, and code static analysis tools) that have formed rich repositories of software artifacts and processes (such as general component libraries, open-source code repositories, software development online Q&A systems, defect tracking systems, and version management systems). However, the problem is how to seamlessly integrate these automation capabilities and resources into an intelligent experience in the entire process with the support of AI technology.

For software development, another important issue is how to organically integrate the capabilities of humans (developers) and machines (LLMs and various tools) to achieve efficient human–machine collaboration. As mentioned before, the role of LLMs in generative software development is closely related to the interaction and guidance abilities of humans. Additionally, human experience plays a significant role in high-level decision-making and code review processes.

Therefore, a systematic intelligent development process should not rely solely on the interaction abilities between developers and LLMs. Instead, it should pursue the establishment of an intelligent interaction engine capable of unifying and effectively integrating the capabilities of LLMs, developers, and various tools. For example, relying on the next-generation integrated development environment (IDE), a unified developer portal can use an intelligent interaction engine to understand the current development tasks and progress. Based on this, it can flexibly schedule and use the generative capabilities of LLMs (such as refining requirements or generating code segments), existing digitized information and knowledge (such as code dependencies, general components, and software design decisions), various tool capabilities (such as running code static analysis or automated testing, querying relevant defect reports, and submitting new defect reports), and developers' subjective judgments (such as making requirements and design decisions and reviewing the results generated by LLMs).

A prominent feature of this process is that the intelligent interaction engine takes the lead, combining developers' experiential judgments with the capabilities of various tools and LLMs in an organic manner, thus achieving a highly smooth, intelligent, and automated development process. Additionally, the intelligent interaction engine needs to manage and switch session states and contexts during the human–machine interaction process between developers and LLMs.

Furthermore, the intelligent interaction engine should support interactive exploration with LLMs in a "multi-threaded" manner and enable the on-demand splitting (fork) and merging (join) of these exploration threads. In this process, the intelligent interaction engine needs to manage and switch session states and contexts effectively.

4 Conclusions

Embracing LLMs is definitely a correct and even necessary direction for software enterprises to improve quality and efficiency. However, achieving systematic and comprehensive intelligent software development still requires careful consideration and there is much fundamental work to do. For enterprises, solidifying the digitization and knowledge accumulation of software development, as well as the fundamental capabilities of software engineering such as requirement analysis, design, and validation, remains crucial and is also a basic condition for achieving higher levels of intelligent development. For academic research, there is still much work to do in the direction of systematic and comprehensive intelligent software development. This also requires us have a deeper understanding of the complexity of software systems and software requirements and design, based on understanding the capabilities of LLMs.

Compliance with ethics guidelines

Xin PENG declares that he has no conflict of interest.

References

Brooks FP Jr, 1987. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10-19. https://doi.org/10.1109/MC.1987.1663532

Dou SH, Shan JJ, Jia HX, et al., 2023. Towards understanding the capability of large language models on code clone detection: a survey. https://arxiv.org/abs/2308.01191

Du XY, Liu MW, Wang KX, et al., 2023. ClassEval: a manually-crafted benchmark for evaluating LLMs on class-level code generation. https://arxiv.org/abs/2308.01861

Hou XY, Zhao YJ, Liu Y, et al., 2023. Large language models for software engineering: a systematic literature review. https://arxiv.org/abs/2308.10620

Liu JW, Xia CS, Wang YY, et al., 2023. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. http://arxiv.org/abs/2305.01210

Meyer B, 2023. AI does not help programmers. *Commun ACM*, early access.

OpenAI, 2023. GPT-4 technical report. https://arxiv.org/abs/2303.08774

Wang JJ, Huang YC, Chen CY, et al., 2023. Software testing with large language model: survey, landscape, and vision. https://arxiv.org/abs/2307.07221

Welsh M, 2023. The end of programming. *Commun ACM*, 66(1):34-35. https://doi.org/10.1145/3570220

Wu QY, Bansal G, Zhang JY, et al., 2023. AutoGen: enabling next-Gen LLM applications via multi-agent conversation. https://arxiv.org/abs/2308.08155

Yuan ZQ, Liu JW, Zi QC, et al., 2023a. Evaluating instruction-tuned large language models on code comprehension and generation. https://arxiv.org/abs/2308.01240

Yuan ZQ, Lou YL, Liu MW, et al., 2023b. No more manual tests? Evaluating and improving ChatGPT for unit test generation. https://arxiv.org/abs/2305.04207

Zhao WX, Zhou K, Li JY, et al., 2023. A survey of large language models. https://arxiv.org/abs/2303.18223

Zheng ZB, Ning KW, Chen JC, et al., 2023. Towards an understanding of large language models in software engineering tasks. https://arxiv.org/abs/2308.11396

Xin PENG is Professor and Deputy Dean of School of Computer Science at Fudan University, China. He received his PhD in Computer Science from Fudan University in 2006. He is Deputy Director of CCF (China Computer Federation) Technical Committee on Software Engineering. He is Co-Editor-in-Chief of *Journal of Software: Evolution and Process* and serves on the editorial boards of reputable journals, such as *ACM Transactions on Software Engineering and Methodology*, *Empirical Software Engineering*, and *Chinese Journal of Software*. His research interests include intelligent software development, cloud native and artificial intelligence for IT operations (AIOps), and software development and testing for smart vehicle. His works won the Best Paper Award of ICSM 2011, the ACM SIGSOFT Distinguished Paper Award of ASE 2018/2021 and ICPC 2022, the IEEE TCSE Distinguished Paper Award of ICSME 2018/2019/2020 and SANER 2023, and *IEEE Transactions on Software Engineering* Best Paper Award for 2018.