Editor in Chief: **Ipek Ozkaya**
Carnegie Mellon Software Engineering Institute
ipek.ozkaya@computer.org

Can Architecture Knowledge Guide Software Development With Generative AI?

Ipek Ozkaya

WE ARE AT a point where AI-based approaches to software development automation are expected to speed up progress on several fronts, including reducing development errors and making changes at scale and with reduced effort. As the initial excitement over generative AI tools has waned, there is a growing awareness of the many potential risks and pitfalls that need to be considered when using them for development, from security to unexpected failures to trust issues. In addition, preliminary empirical findings from analyzing the use of AI-based development assistants indicate that the desired ideal improvement in productivity and quality will not be a result of better tools, even if they are helpful; significant improvement will be the result of understanding and redesigning task flows[1] and ensuring expert judgment in the use of these tools.[2]

Developers and researchers have focused much attention on the use of generative AI tools to improve implementation activities. However, there has been little attention on how design and architecture tasks can be effectively accomplished with generative AI-based software development tools. Some things never change! Hasn't it always been the case that deliberate design and its value are assumed to emerge from code, which is an afterthought when things don't work, or that essential complexity is assumed to be a reality of long-lived systems?

Analyzing developer use data to date demonstrates that expertise will be essential to assess not only the correctness of the tool recommendations but also the fitness for the purpose of code developed with AI-augmented tools.[2] As local changes are made with the help of AI-assistants, there are likely implications of the changes to the overall structure and behavior of the systems. In other words, we need to think about how we will design and build systems in the future given that AI-augmented tools will be playing a more significant role. We need to make sure that these systems are safe, reliable, and fit for purpose while ensuring that the process of designing, developing, and deploying them takes into account solving issues at different levels of abstraction.

There are several questions to explore when incorporating design and architectural concerns into the use of generative AI-based system development explorations, especially with generative AI tools.

- What are the specific design and architectural concerns that need to be addressed when using generative AI?

Digital Object Identifier 10.1109/MS.2023.3306641
Date of current version: 13 October 2023

0740-7459/23©2023IEEE

- Can generative AI tools be used to improve the design and architecture of systems?
- Can these tools provide new features and capabilities that can be used to support the architectural design process?
- Could generative AI tools be used to generate design patterns and tactics, which could then be used to guide the code generation or evolution of tools?
- Can these tools accelerate the generation of alternative designs and their comparison?
- Can they be used to provide feedback on designs, such as identifying potential risks and issues?

Overall, the use of AI-augmented and generative AI tools has the potential to revolutionize the way in which systems are designed and architected. At least this is how we may want to envision the future of software development. However, a key underlying assumption in answering all of these questions is that sufficient architecture knowledge is available to reliably develop underlying large language models (LLMs) and they can be encoded and decoded to guide aspects of system development with tools. These questions all point to the need to focus on architecture knowledge management and to perhaps consider how or if it can even be shared through generative AI tools.

Architecture Knowledge

Software architecture knowledge can be considered as the union of the following elements:[3]

- *Architecture design*: The overall structure of the software system, including its components, their relationships, and the data that they exchange, makes up architecture design.
- *Design decisions*: These are the choices that were made during the architecture design process, such as the choice of programming language, the choice of data model, and the choice of architectural patterns.
- *Assumptions*: Architects and developers make choices in attributes that were assumed to be true during the architecture design process, such as the size of the system, the performance requirements, and the security requirements.
- *Context*: The environment in which the software system will be used, such as the hardware platform, the operating system, and the network infrastructure, influences design decisions, assumptions, and tradeoffs.
- *Other factors*: There are other factors that were considered during the architecture design process, such as cost of development, the cost of maintenance, and time to market.

All of these elements together determine why a particular software solution is the way it is.[4] While it is possible to capture the design through the code and other implementation artifacts, capturing design decisions and assumptions requires knowing the context and the tradeoffs. These tradeoffs also often involve deciding between many competing and ever so evolving technologies and how they may be compared to each other.[4,5,6]

The software engineering community has focused on architecture knowledge management as a key aspect of system design since the early 2000s.[3] There has also been considerable progress in abstracting and

CONTACT US

AUTHORS
For detailed information on submitting articles, visit the "Write for Us" section at www.computer.org/software

LETTERS TO THE EDITOR
Send letters to software@computer.org

ON THE WEB
www.computer.org/software

SUBSCRIBE
www.computer.org/subscribe

SUBSCRIPTION CHANGE OF ADDRESS
address.change@ieee.org
(please specify *IEEE Software*.)

MEMBERSHIP CHANGE OF ADDRESS
member.services@ieee.org

MISSING OR DAMAGED COPIES
contactcenter@ieee.org

REPRINT PERMISSION
IEEE utilizes Rightslink for permissions requests. For more information, visit www.ieee.org/publications/rights/rights-link.html

Digital Object Identifier 10.1109/MS.2023.3311928

encapsulating architectural knowledge with the goal of generalizing and sharing it. There is a vast body of literature that documents repeatable solutions to repeatable problems in the form of architecture patterns, design patterns, and tactics.[6] Yet, the software engineering discipline has always been challenged by the static nature of architecture knowledge capture in the face of rapid technology change. There were attempts at developing knowledge repositories to structure knowledge about emerging technologies;[5] however, since such repositories have not been seamlessly integrated with the design and implementation flow of developers, they have not made it into the toolchain of software engineers effectively.

Architectural knowledge and the ability to make meaningful tradeoffs are expert skills and imply the experience of seeing similar examples over different situations. In addition, architecture knowledge management implies the ability to see through the design decisions as implementation constructs.

Martin Fowler illustrates the design knowledge and the expertise required to guide the development process using a generative AI tool quite vividly in an example of a self-testing code demonstration during a conversation with Xu Hao, Thoughtworks's head of technology in China.[7] As shown in "Self-Testing Code," the initial prompt that Hao uses to kick off the code generation process with ChatGPT is elaborate with design and technology stack information. The first prompt alone involves the following components, which can almost be perceived as an architecture and design-driven prompt template. It is easy to observe a number of generalizable steps in the prompt as follows:

[type of system]
[list of technologies in the tech stack]
[framework used for components] [architectural pattern used for the system]
[elaboration on the architecture pattern suggested with alternatives]
[known implementation strategies]
[recommended patterns for tests required]
[requirements for a portion of the design]
[required output format (for example, code, explanation)]

This initial prompt to provide all of this information takes about 400 words. Hao requests and receives a plan from ChatGPT, the generative AI tool of his choice, in this exercise. The plan includes step-by-step instructions and provides recommended components to include in the design and implementation. The recommended plan also includes references to design elements and constructs, such as "create encapsulated view model interface" or the use of architecture pattern vocabulary like "layers."

Design Prompts to Guide Software Development

The importance of well-structured prompts, most often referred to as *prompt engineering*, has already been established in the short time when generative AI technologies have become a part of software engineers' toolkits. *Prompt engineering* is the process of designing and crafting prompts that are used to control generative AI tools. Prompt patterns

SELF-TESTING CODE

The following is an example of a self-testing code[7] (Source: Reused with permission from Martin Fowler, "An Example of LLM Prompting for Programming," at https://martinfowler.com/articles/2023-chatgpt-xu-hao.html):

The current system is an online whiteboard system. Tech stack: typescript, react, redux, konvajs and react-konva. And vitest, react testing library for model, view model and related hooks, cypress component tests for view.

All codes should be written in the tech stack mentioned above. Requirements should be implemented as react components in the MVVM architecture pattern.

There are 2 types of view model in the system.

1. Shared view model. View model that represents states shared among local and remote users.
2. Local view model. View model that represents states only applicable to local user

Here are the common implementation strategy:

1. Shared view model is implemented as Redux store slice. Tested in vitest.
2. Local view model is implemented as React component props or states(by useState hook), unless for global local view model, which is also implemented as Redux store slice. Tested in vitest.
3. Hooks are used as the major view helpers to retrieve data from shared view model. For most the case, it will use 'createSelector' and 'useSelector' for memorization. Tested in vitest and react testing library.
4. Don't dispatch action directly to change the states of shared view model, use an encapsulated view model interface instead. In the interface, each redux action is mapped to a method. Tested in vitest.
5. View is consist of konva shapes, and implemented as react component via react-konva. Tested in cypress component tests

Here are certain patterns should be followed when implement and test the component

1. When write test, use describe instead of test.
2. Data-driven tests are preferred.
3. When test the view component, fake view model via the view model interface

Awareness Layer
Requirement:
Display other users' awareness info(cursor, name and online information) on the whiteboard.

AC1: Don't display local user
AC2: When remote user changes cursor location, display the change in animation.

Provide an overall solution following the guidance mentioned above. Hint, keep all awareness information in a Konva layer, and an awareness info component to render cursor, and name. Don't generate code. Describe the solution, and breaking the solution down as a task list based on the guidance mentioned above. And we will refer this task list as our master plan.

have become critical in leveraging the power of LLMs and applications built on them because they guide users to specify exactly what they want the tool to do. This is important because while generative AI tools can be powerful, they are also very unpredictable and inconsistent. Carefully crafted prompts will not completely improve the correctness of the solution generated at all times, but it is one powerful strategy to rely on when working with generative AI tools.

Crafting appropriate prompts requires expertise in the problem at hand and can be a complex and challenging task, but it is essential for getting the most out of generative AI tools. There are a number of factors that need to be considered when crafting a prompt, such as the task that the tool is being used for, the desired output format, and the level of detail that is required.[8] It is also important to test the prompt with the tool to make sure that it produces the desired results. The generic example presented by Fowler[7] already hints that architectural knowledge will become a vital skill not only for

assessing the results of code generation but most likely also for providing essential input to help iterate development tasks using generative AI tools.

If we want to use generative AI tools to assist in system development beyond method or class-level implementation tasks and toy examples, we must embrace incorporating architectural knowledge into the process.

LLMs can easily be trained on generic and public information such as architecture patterns, tactics, technologies, and their attributes that are already available as book text and other publications. These data, when accessed more easily with the assistance of generative AI tools, can enable improved design guidance in the development process, which can reduce the cognitive load and time of design exploration for developers.

Will Architecture Expertise Be Fashionable Once More?

Software architecture has been misconceptualized as not being the cool kid on the software engineering block before. Recall the miscommunications during the early years of agile software development processes.[9] These misconceptions have sometimes resulted in software architecture and design being discarded. Incorrect execution of the architect role as an ivory tower architect, disconnected from the realities of implementing and deploying systems, also contributed to these misconceptions. Architecture knowledge and design decisions not only guide the structure and behavior of systems, but they guide how software is developed, maintained, scaled, and adapted to meet evolving requirements.

If we want to use generative AI tools to assist in system development beyond method or class-level implementation tasks and toy examples, we must embrace incorporating architectural knowledge into the process. Using architectural patterns, tactics, and design constructs to direct code generation in generative AI tools and applying this knowledge toward iterative explorations could help make generative AI tools more applicable to more complex activities in the software development process. Incorporating architecture knowledge into data that LLMs are trained on and to tools that software engineers use to solve large scoped development problems ranging from technology upgrades, language translation, to software evolution is an unexplored area with challenging problems, but exciting applications if successful.

References

1. I. Ozkaya, "The next frontier in software development: AI-augmented software development processes," *IEEE Softw.*, vol. 40, no. 4, pp. 4–9, Jul./Aug. 2023, doi: 10.1109/MS.2023.3278056.
2. C. Bird et al., "Taking flight with copilot," *Commun. ACM*, vol. 66, no. 6, pp. 56–62, Jun. 2023, doi: 10.1145/3589996.
3. P. Kruchten, P. Lago, and H. van Vliet, "Building up and reasoning about architectural knowledge," in *Quality of Software Architectures*, vol. 4214, C. Hofmeister, I. Crnkovic, and R. Reussner, Eds., Berlin, Heidelberg: Springer, 2006, pp. 43–58.
4. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Reading, MA, USA: Addison-Wesley, 2012.
5. I. Gorton, R. Xu, Y. Yang, H. Liu, and G. Zheng, "Experiments in curation: Towards machine-assisted construction of software architecture knowledge bases," in *Proc. IEEE Int. Conf. Softw. Architecture (ICSA)*, Gothenburg, Sweden, 2017, pp. 79–88, doi: 10.1109/ICSA.2017.27.
6. F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture, on Patterns and Pattern Languages*. Hoboken, NJ, USA: Wiley, 2007.
7. M. Fowler, "An example of LLM prompting for programming," *Martin Fowler*, Apr. 2023. Accessed: Jul. 2023. [Online]. Available: https://martinfowler.com/articles/2023-chatgpt-xu-hao.html
8. J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," Mar. 2023, *arXiv:2303.07839*.
9. P. Abrahamsson, M. A. Babar, and P. Kruchten, "Agility and architecture: Can they coexist?" *IEEE Softw.*, vol. 27, no. 2, pp. 16–22, Mar./Apr. 2010, doi: 10.1109/MS.2010.36.