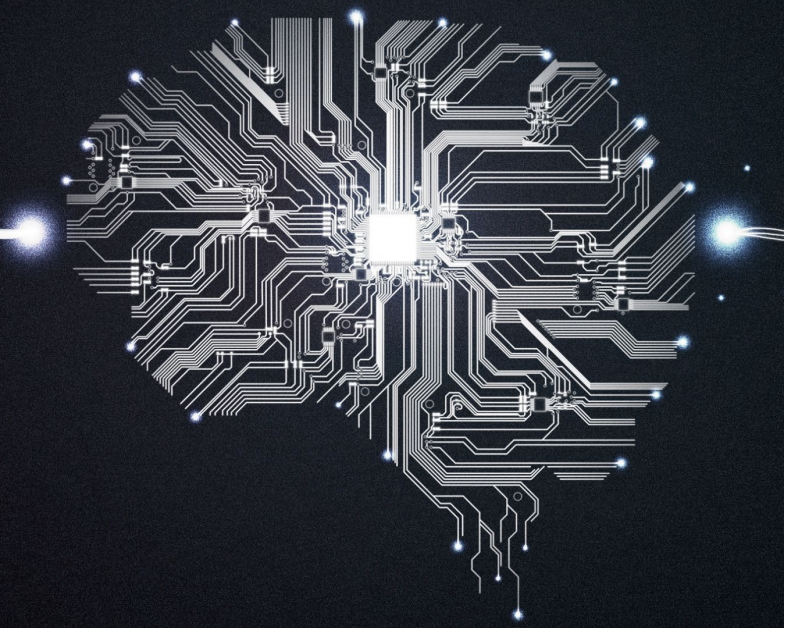
Responsible Software Development in the Era of Generative AI

From the Perspective of the Knowledge Gap Between AI and Humans

NTT DATA Corporation
Toyosu Center Building,
3-3-3 Toyosu, Koto-ku, Tokyo 135-6033, Japan
Tel: +81-3-5546-8051 Fax: +81-3-5546-2405
https://www.nttdata.com/global/en/

Contents



1 Preface

The advent of generative AI marks a significant milestone in software development. As artificial intelligence continues to evolve, it not only enhances human capabilities but also reshapes the paradigms of creativity and problem-solving. This white paper examines the impact of generative AI on software development, with particular emphasis on the knowledge gap between humans and AI. It discusses technical elements expected to advance in the future and considers their implications for the development process. Lastly, it addresses the risks of knowledge debt introduced by generative AI and outlines strategies to mitigate them.

2 State of Generative AI in Software Development

Software developers are increasingly using generative AI systems to streamline the development process and improve productivity. GitHub Copilot is an AI-powered code completion tool that suggests code snippets based on the context provided by developers.[1] By using this tool, developers can reduce the effort involved in repetitive tasks or internet searches. AI agents have the ability to autonomously perform tasks to achieve given goals and have gained popularity in software development. Devin AI and Cline are general-purpose software development agents that autonomously perform tasks from analysis to implementation based on human instructions.[2,3] Many development tasks are automated by AI agents, allowing developers to focus on verifying the validity of the results. As a result, development teams can expect improvements in the quality, cost, and delivery of the software development process. Furthermore, by delegating repetitive tasks and search operations to AI, development teams can achieve a more satisfying developer experience

Two trends can be observed from the recent evolution of AI systems. The first point is that the role of AI is shifting from supporting software development tasks to substituting for human work. GitHub Copilot, released in 2021, is a code completion tool that utilizes generative AI and is a forerunner of development efficiency tools. It has supported developers' implementation tasks by providing completion features for the source code they are writing. Subsequently, with the evolution of AI models, more powerful AI coding tools such as Cursor and Aider were introduced in 2023.[4,5] These tools directly edit source code based on the requests provided by developers. The source code rewriting feature was also implemented in Copilot Edits in 2024, making it a popular approach as a modern AI coding tool. Developers can now complete many coding tasks by simply accepting the results of AI work. This differs from the conventional approach of supporting developers, as AI actively performs coding tasks. In addition to coding, the use of AI for test code generation is an area where AI utilization is advancing. It has been possible to generate test code variations using GitHub Copilot. Qodo Cover, an open-source agent that provides very powerful test code generation using AI, was announced in 2024.[6] Specifically designed for white-box testing, it analyzes the source code to identify potential paths and generate test cases that achieve high coverage. It receives the source code to be tested as input and generates test cases with high code coverage. The generated test cases are automatically executed, and their validity is confirmed. As a result, it is guaranteed that the generated test code contributes to improving coverage. In this process, developers do not even need to review the existence of hallucinations, which are errors included in the AI's generated results. This tool is designed to minimize the effort of human verification and correction. Previous AI tools have been designed with the assumption that human review and correction of the generated results are necessary. In contrast, user interfaces that assume the direct acceptance of the results of powerful AI models have become common.

The second point is that AI is expanding the range of tasks it can handle at once. GitHub Copilot complements incomplete source code and could only handle a single file. In contrast, the functionality of Copilot Edits, provided since 2024, automatically edits multiple files to realize developers' requests. Developers can reduce their work time by not having to open and edit files one by one to implement a feature. Furthermore, in the software engineering community, research on source code generation and translation at the repository level is actively conducted. For example, researchers are studying technologies that automatically identify files to edit to implement a feature as well as technologies that analyze dependencies and consistency between files during code generation.[7] In the near future, technologies that simultaneously edit all the source code in a repository are expected to be put into practical use.

3 Knowledge Gap between Human Developers and AI Systems

Generative AI is changing software development. However, it does not have as much knowledge as human developers. Because of this gap, AI cannot deliver the same work quality as humans. We focus on this difference in available knowledge and how it affects software development.

In software development, human developers use a wide range of knowledge and combine it as needed to complete tasks. If AI systems have less knowledge than humans, they cannot produce the same outcomes. To clarify this issue, we examine the internal structures of AI systems and the project-specific information they require. Then, we discuss the technical challenges that must be addressed.

To understand the essential structure of knowledge utilization, we focus on two aspects of AI systems: the internal and external aspects. Specifically, we distinguish between the AI system itself (internal) and project-specific information as viewed from the AI system (external). We then organize the interaction between AI systems and project-specific information. Figure 1 illustrates this structure. The left side of the figure represents the AI system, and the right side represents project-specific information. This figure will be referred to throughout this paper.

The internal design of an AI system is intended not only to incorporate vast amounts of general knowledge gleaned from training data, but also to enable thinking and planning for task execution. These capabilities heavily rely on the performance of the underlying language model. With the evolution of AI models such as the OpenAI GPT series, the general knowledge that AI possesses continues to grow. It is no exaggeration to say that these models surpass the knowledge of individual humans by encompassing nearly all publicly available information. Furthermore, with the advent of powerful reasoning models like the OpenAI o1 series, AI systems are able to engage in even more advanced thinking and planning. As a result, they can autonomously devise procedures and carry out tasks to achieve given goals in certain scenarios. By incorporating agent-like functionalities into their internal structure, AI systems can interact with development environments and conduct iterative trial and error. In such cases, AI systems have become capable of handling a large portion of developers' work.

The external world as viewed from an AI system, namely project-specific information, is multi-layered. Developers involved in the project use this specific information on a daily basis. Ideally, to delegate developers' work to AI, the same amount of information should be provided to AI as to developers. To replicate developers' work, AI systems need to combine diverse project-specific information to perform tasks. Otherwise, the results of AI systems' work will be incomplete, and developers will need additional work to correct the AI's output. In short, bridging the knowledge gap between AI and developers is essential to maximize the effectiveness of AI utilization.

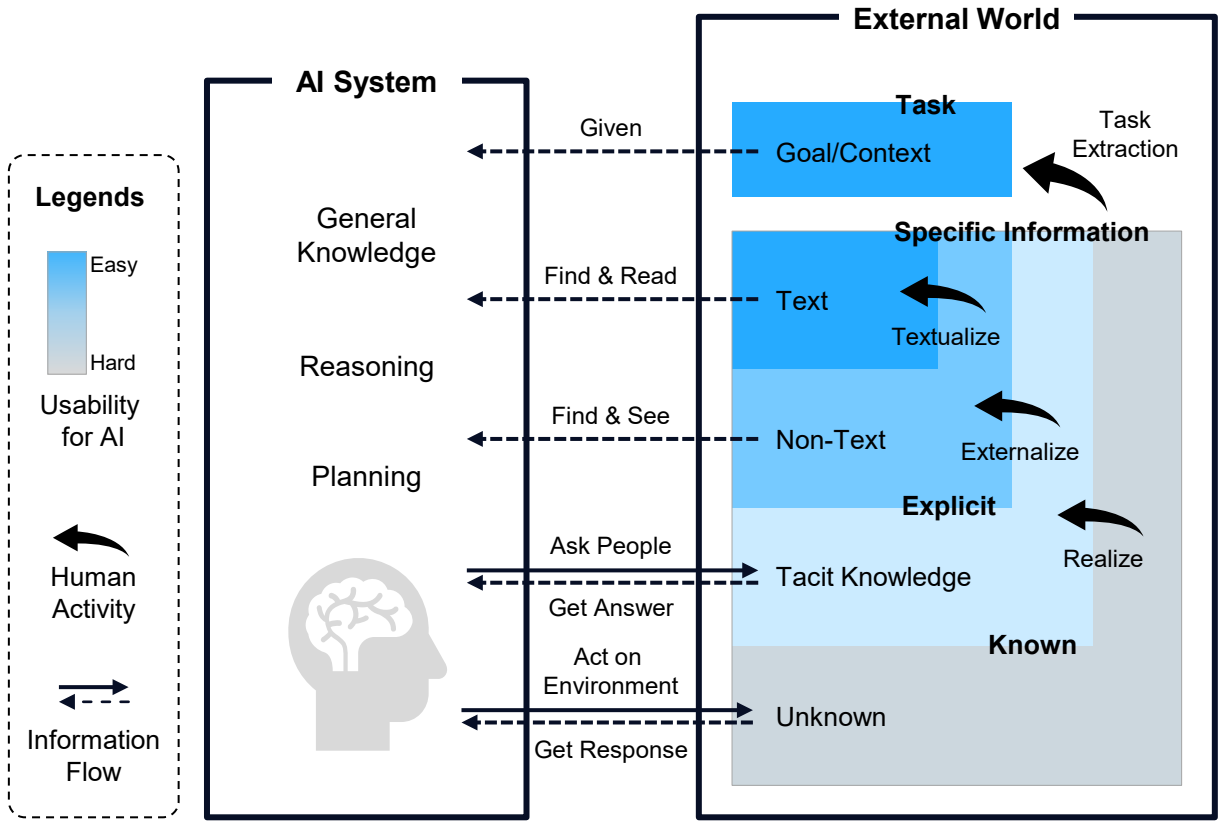


Figure 1: Interaction between AI Systems and External World

Project-specific information consists of the following elements:

- **Text Data:** Large Language Models (LLMs) can process various types of text data—such as source code, documents, execution logs, and development history—without needing to treat them differently. This makes it possible for them to handle a wide range of tasks efficiently.
- **Non-Text Data:** Non-text data managed in binary formats, such as UML diagrams, flowcharts, or files in MS Excel or PDF. This category includes coding conventions, test procedures, and tickets on management systems if they are not stored as flat text. Non-text data tends to be more difficult to process by computers than text data. It is managed as project-specific "formal knowledge" along with text data.
- **Tacit Knowledge:** Implicit knowledge and experiential knowledge held by project members. This knowledge is essential for the success of development projects, including development rules, design patterns, and past experiences. It also includes the characteristics of each development member, past agreements with teams or clients, and deadlines. Alongside explicit knowledge, this knowledge is treated as "known" information for project members.
- **Unknown Information:** Project-specific information that no project member is aware of or information that can only be determined by verifying it.. This category includes potential defects, implementations with unclear specifications, incompatible information for new platforms, and the scope of change impacts. It also includes the accuracy of task results before validation and the user experience of the software prior to release.

AI systems for software development perform tasks by acquiring project-specific information at each layer when given goals and context. They directly reflect the results of their work as part of project-specific information. The acquisition methods of each information and common challenges are organized as follows:

Acquisition of Text Data

AI systems acquire text data necessary to perform tasks. AI reads the files provided as the context of the task or retrieves text from the file system or database. The approach known as Retrieval Augmented Generation (RAG), which combines generation and retrieval, is included in this category. The challenges include the difficulty of discovering the necessary information for task completion when the amount of text data is vast. Additionally, if the quality of the text is low, there is a higher possibility that the generated results will be inaccurate. There are cases where AI systems do not have interfaces to access text data.

Acquisition of Non-Text Data

AI systems acquire non-text data necessary to perform tasks, similar to text data. If non-text data contains useful text data, preprocessing is performed within the AI system to extract text. Multi-modal AI is used to read the content of diagrams and image data. Many of the challenges are similar to those of text data. That is, it is difficult to search for appropriate data for tasks. The reading accuracy of multi-modal AI tends to decrease compared to processing text data. As a result, the generated results are more likely to contain errors or omissions.

Acquisition of Tacit Knowledge

AI systems interact with project members through communication channels to acquire the tacit knowledge of project members. AI systems ask appropriate members questions, and members answer questions to provide information to AI systems. User-in-the-loop, which is often introduced in the context of AI agents, is a mechanism for feeding back human knowledge to AI and is included in this category. Improving the generated results based on feedback from humans leads to an improvement in the quality of the final generated results.

The challenges include the cost of operating communication channels with developers. AI systems also need to accurately determine which member to ask what. If this decision is inaccurate, it may place extra burden on developers, resulting in increased human costs, such as extra time, effort, and lowered productivity.

Acquisition of Unknown Information

AI systems directly act on the development environment or software execution environment to reveal unknown information and obtain the results. For example, AI systems build the generated source code, conduct tests, and determine the correctness of the source code based on the test results. If the test fails, AI systems receive the result as feedback and use it to improve the next generation.

The challenges include the difficulty of finding useful information in vast execution logs. There are also cases where AI systems cannot easily access the development or execution environment. When building source code in a dedicated environment, there are costs associated with setting up and managing the environment. Running a program with generated results that have not been verified for quality poses security risks. For example, the generated source code may include communication with external entities, posing a risk of leaking confidential information. To prevent such risks, it is necessary to prepare an environment with carefully designed network settings.

In general, data located in the inner layer of project-specific information is easier for AI to handle. Therefore, to utilize project-specific information, it is desirable to convert data from the outer to the inner layer. This activity has been carried out since before generative AI was used in software development because it is beneficial for human developers. Specific actions and their barriers are as follows:

Textualization

From Non-Text to Text

Developers convert non-text data into text data. Developers read flowcharts and describe their contents as text data. They may also use tools such as Optical Character Recognition (OCR) to convert images to text data or extract text from PDF files. It is also possible to use multi-modal AI for this activity.

The main barrier is the high cost of converting non-text data to text data. Development projects have a large amount of non-text data, and non-text data formats are diverse. Therefore, it is difficult to cover all conversion patterns with simple conversion logic.

Externalization

From Tacit to Explicit Knowledge

This process is called externalization in the context of knowledge management. For example, development members document insights gained from years of experience and manage them as project-specific information. This activity not only makes it easier for AI systems to acquire project knowledge but also promotes knowledge sharing among project members.

The barrier is the vast amount of tacit knowledge. Also. tacit knowledge includes things that are difficult to verbalize, such as the intuition of skilled developers. Since human memory gradually fades, developers often start by organizing missing information before documenting it. In such cases, the cost of converting to explicit knowledge is particularly high. Furthermore, externalized information always carries the risk of becoming outdated, and development teams need to review it regularly.

Realization

From Unknown to Known

Developers explore unknown areas to understand the situation. At least one member of the development team holds that information as tacit knowledge. For example, an experienced developer analyzes source code with unclear specifications and understands its behavior. In addition, they confirm whether a new software framework meets the requirements through prototyping. By verifying the results, the development team identifies specific impacts and risks. This activity is also beneficial for risk management of the entire development project.

The main challenge is that understanding unknown areas requires deep knowledge and experience in software development. It is difficult for inexperienced developers to anticipate defects and risks in advance. Training developers with general development knowledge and project-specific experience takes a long time.

4 Towards AI-Native Software Development

When seeking to fully harness AI throughout the software development process (sometimes referred to as *AI-native software development*), there are several technical challenges to consider. Depending on how AI uses project-specific information, the ways to address challenges can be divided into three main areas: refining AI systems internally, organizing project-specific information for easier use, and integrating both internal and external AI components. The expected improvements in each area are explained below.

Enhancement of AI Systems

Enhancement of AI system performance is achieved by enabling AI systems to have richer general knowledge, more advanced reasoning capabilities, and higher autonomy. Enhanced AI systems understand the intent of tasks with fewer instructions and reach the correct results in the shortest steps. The supporting technologies include the expansion of knowledge and improvement of reasoning capabilities of AI models, and the efficiency of trial and error through advanced agentic technologies. The cost of learning and inference is important in practice. By reducing the cost of AI models, powerful AI systems can be used at the individual level. The use of domain-specific small language models (SLMs) and the merging technologies that combine SLMs helps improve the usability of AI systems. As a result, the scope of AI system applications grows wider, enabling development teams to streamline the entire software lifecycle in various projects.

Enhancement of Project-Specific Information

Enhancement of project-specific information is achieved by converting existing knowledge into a form that AI can handle more easily. The mission is to efficiently convert data from the outer layer of the knowledge layer to the inner layer. The supporting technologies for this are diverse. The evolution of basic technologies for image and language processing is important for converting images to text. Tuning technologies for recognition patterns dependent on existing formats are also important. The process of converting tacit knowledge into explicit knowledge should be explored. Tacit knowledge that can be extracted by repeatedly asking random questions to project experts is limited. Standardizing the process of accurately extracting information useful for AI should be considered. Building systems to support this would be effective. To convert unknown information into known information, we need methods to prioritize exploring areas where we can obtain the largest amount of information. For example, for source code with unclear specifications, it is necessary to prioritize the investigation of parts that are likely to change in future development.

These improvement activities may affect the existing formats that developers have been using. For instance, when converting graphical design diagrams managed in MS Excel to text representations in Mermaid format, developers need to acquire skills to read and write Mermaid. To minimize additional learning costs, training and technical support should be provided to developers.

Enhancement of Knowledge Acquisition

Enhancement of knowledge acquisition is achieved by improving the interface for AI to access project-specific information. As mentioned earlier, project-specific information is multi-layered. Ideally, AI systems should have access to all knowledge used by developers. The technology to achieve this includes advanced search technologies to acquire useful information for tasks. Improving the performance of multi-modal AI is useful for obtaining information from various non-text data. To effectively utilize tacit knowledge held by humans, a question strategy based on the expertise of project members should be designed. To acquire unknown information, AI needs access to the development or execution environment of the software. Building and operating secure sandbox environments is an important theme. When AI explores unknown areas, it is important to have strategies to efficiently obtain more useful information. For example, when verifying the validity of generated source code through testing, test cases should be strongly related to the newly generated parts of the source code.

The Model Context Protocol (MCP) announced by Anthropic in 2024 is a highly versatile approach for expanding interfaces to access project-specific information.[8] MCP is a protocol that standardizes the interaction between AI systems and external information. It reduces the effort to develop methods for acquiring and updating information for each project.

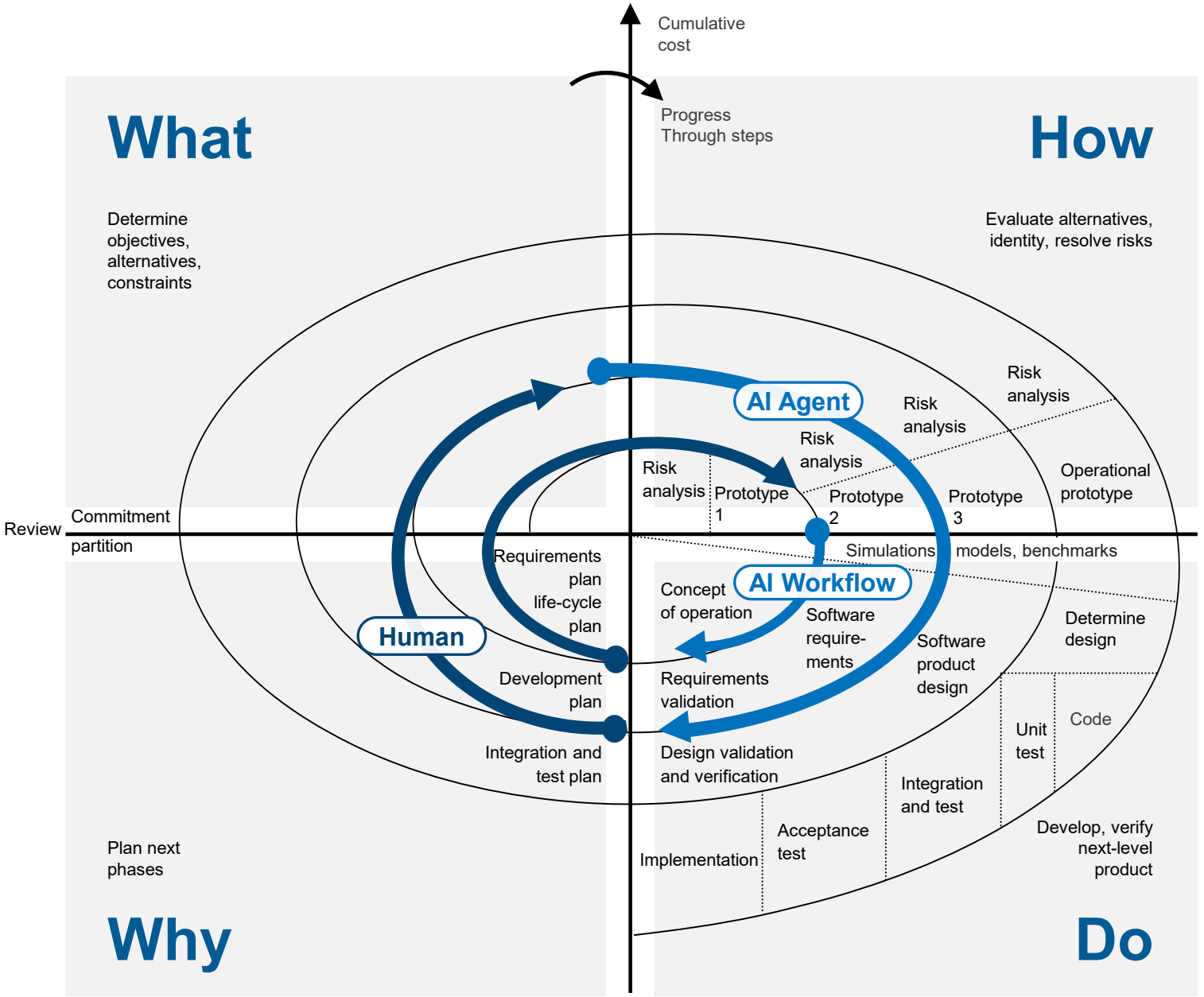


Figure 2 : Spiral Model and Roles of AI Systems

The Spiral Model in Software Development

As a result of these steady improvements, we anticipate a future in which AI systems perform a wider range of development tasks end-to-end.

Here, we consider how software development will change due to the evolution of AI systems based on the structure of the spiral model in Figure 2. The spiral model is a meta-process proposed by Barry Boehm in 1986.[9] The process moves clockwise around Figure 2, divided into four quadrants.

The lower-left area is a phase that organizes motivation for the next development scope. This phase clarifies WHY the development is necessary. The next upper-left area is a phase that considers the purpose and constraints of development, clarifying WHAT to achieve in development. The upper-right area is a phase that determines how to achieve it while considering possible risks, clarifying HOW to develop. The lower-right area is the actual development work, where waterfall or incremental development methods are adopted depending on the risks identified. In short, it has a structure of performing actual work after clarifying WHY, WHAT, and HOW in the entire development cycle. As mentioned in the original paper, this cycle structure appears at various granularities in software development. For example, in a waterfall development method, implementation phase proceeds after going through a detailed process of user requirements (WHY), system requirements/external design (WHAT), and internal design (HOW). Coding work has a finer-grained cycle structure, where implementation starts after going through stages of WHY a change is necessary, WHAT the source code should be, and HOW to change the source code.

AI systems that are used in software development can generally be divided into two types: AI workflows and AI agents. AI workflows are systems that have a fixed sequence of processes, and they typically include pre- and post-processing tasks centered on AI invocation. In contrast, AI agents adapt their processes dynamically. They are designed to use trial-and-error to achieve a specified goal.

We can visualize their roles in the spiral structure (see Figure 2 in the previous page).

- AI workflows can be seen as carrying out the actual work by following a predefined procedure (HOW). For example, a workflow that generates source code from a design document might include prompts that list each step needed such as analyzing the design document and producing high-quality code.
- AI agents autonomously devise implementation steps (HOW) based on the specified objectives (WHAT) and then perform the work. In this scenario, the primary role of developers is to define the goal (WHAT), which allows them to focus on more upstream processes than they would if they were working with AI workflows.

Looking ahead, more sophisticated AI technologies will likely accept the underlying requirements or problems (WHY) as input. They will independently figure out goals and constraints (WHAT), decide how to achieve them (HOW), and then execute the necessary tasks. As such systems become more common, the primary human responsibility will shift to creating and managing user requirements. Within the software engineering community, this transition is referred to as Software Engineering 3.0.[10] It highlights a move away from coding and other downstream tasks, and toward defining the higher-level “why” of a software development project.

5 Knowledge Debt in AI-Native Software Development

We have discussed how AI will replace developers' work in software development, and human work will shift further upstream. AI-native software development is expected to significantly improve productivity. However, new challenges that have not been a problem before may arise. One main challenge is *knowledge debt*. As humans rely more and more on AI, it becomes harder for them to maintain the knowledge necessary for long-term software evolution.

Figure 1 shows the relationship between AI systems and project-specific information. An important question here is which knowledge layer AI-generated work falls into. If an AI system generates source code and directly commits it to the codebase, does the source code become explicit knowledge of the project? The answer is no. Because no one in the project knows about the results of the AI's work, the results of the AI's work initially exist as "unknown" information. When project members understand the results of the AI's work, it becomes "known" information. Then, when developers confirm the correctness of the results of the AI's work, the information becomes explicit knowledge of the project. Only after it becomes explicit knowledge can it be utilized in subsequent tasks. Figure 3 illustrates this structure. With this in mind, it is important to incorporate a review process for turning the results of AI systems' work into explicit knowledge in software development.

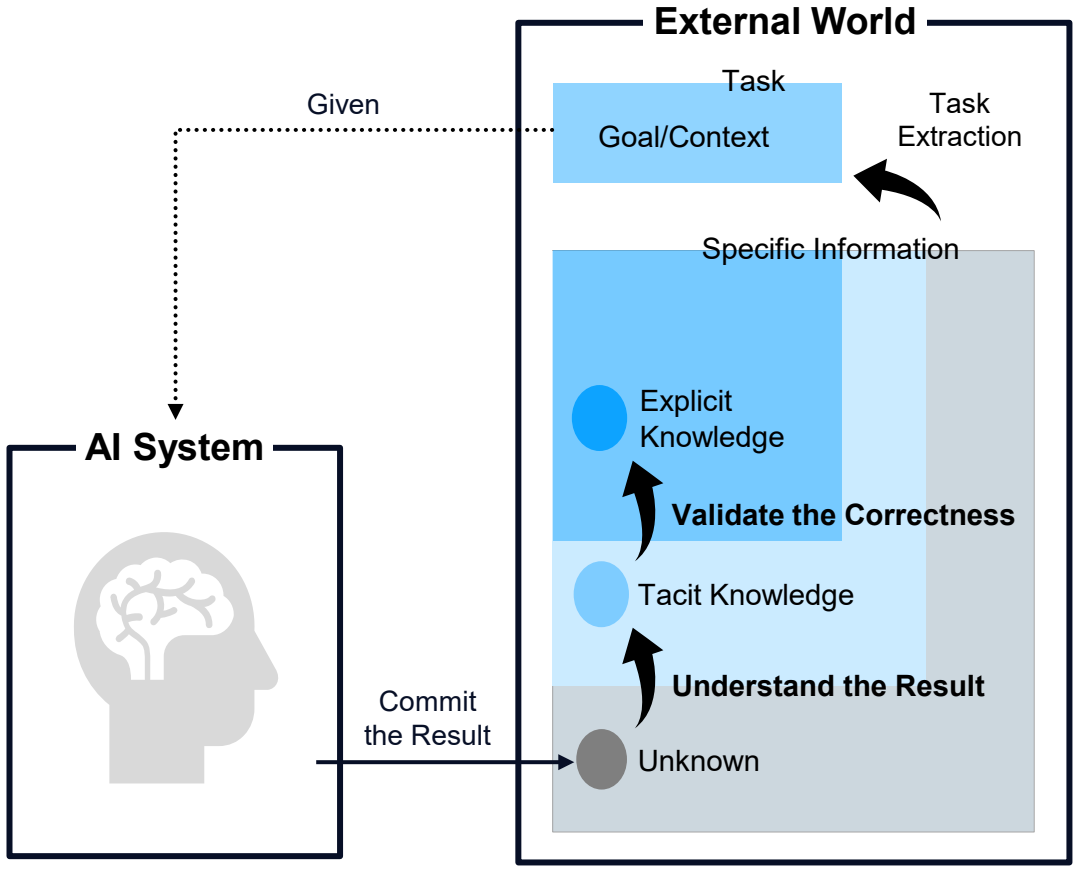


Figure 3: Effect of Review Process

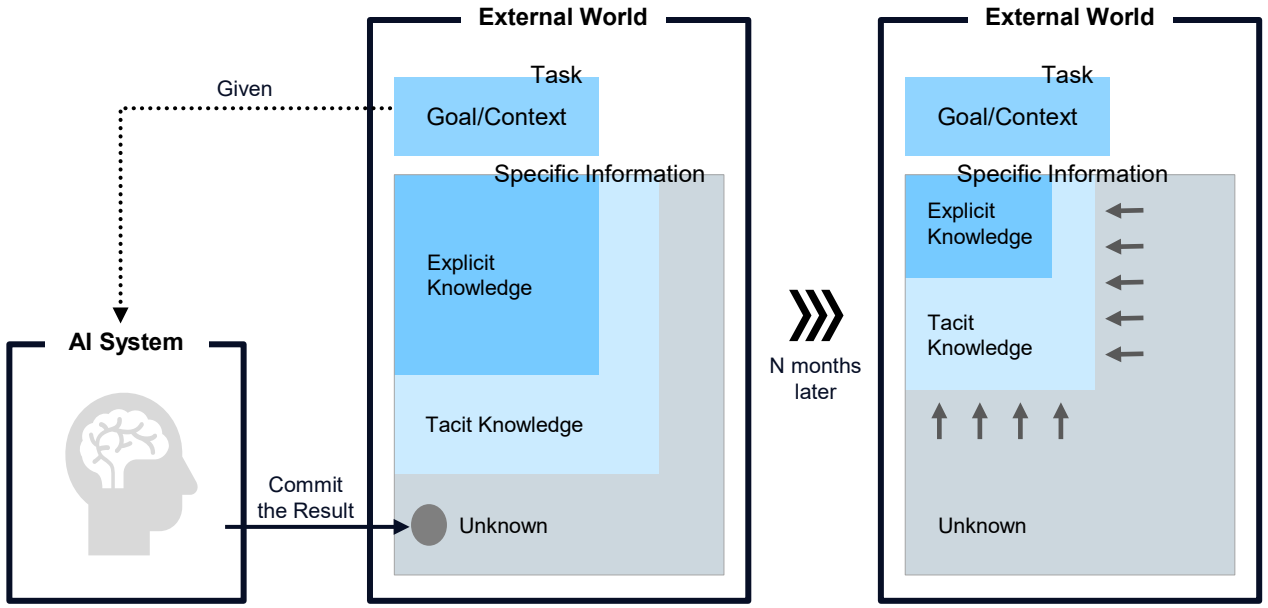


Figure 4: Accumulation of Knowledge Debt from Overreliance on AI

In a world where AI systems surpass most developers' skill levels, it becomes increasingly common to accept AI-generated outputs without manual revision. As this trend continues, a bias emerges that AI's results are always correct, leading to an overreliance on AI. Gradually, fewer developers make the effort to understand the AI's outputs, and these outputs accumulate within the project as unknown knowledge. This accumulated unknown knowledge is referred to as knowledge debt, and it represents one of the major risks associated with introducing generative AI into business processes (Figure 4). In general, the presence of source code with unknown behavior makes it difficult to ensure the quality of software products. If a problem arises that generative AI cannot solve, developers must analyze and address the issue themselves, just as they do in traditional development processes. However, in projects where excessive AI use has led to substantial knowledge debt, the cost of understanding the current state becomes higher, making problem-solving even more challenging than before. Long-term risks include decreased software quality, project delays, and increased Mean Time to Resolution (MTTR) to resolve issues that arise during software release periods.

With advances in fundamental technologies, it is expected that the accuracy of AI systems in performing tasks will continue to improve. We predict that, within a few years, AI systems will surpass the skills of most developers. This is because developers' skills typically follow a pyramid distribution: the majority are at the junior level, while only a small number reach senior or higher levels. As a result, the number of developers whose skills are overtaken by AI will increase at an accelerating rate relative to AI's capabilities. Figure 5 illustrates this structure. Once an AI system's skill level rises from the middle of the junior range to the middle of the senior range, only a minority of developers will possess higher skills than the AI system. It is natural to expect that AI will play a more central role in software development.

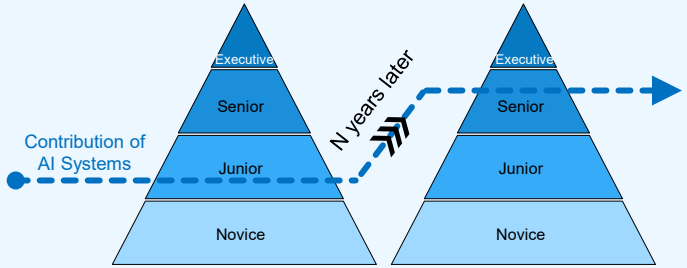


Figure 5: The Developer Hierarchy and AI's Influence

6 Responsibility Towards AI-Native Software Development

What responsibility should we have as AI transforms software development? Our significant responsibility in the accelerated software development process driven by AI is to appropriately manage the information generated by AI and achieve sustainable development. The following are main approaches:

Strengthen Review Processes:

The most straightforward way to avoid the accumulation of knowledge debt is to establish a thorough review process by human developers as a standard practice. This review should be more meticulous than those traditionally conducted during development. Developers gain a deeper understanding of current specifications, design, and implementation every time they create artifacts. However, in AI-driven development, such knowledge acquisition is often lacking. Therefore, it is important to address and compensate for this lack of knowledge in the review phase. Reviewers can enhance their understanding by running AI-generated source code and tests, experimenting with different approaches, and so on. Walkthroughs that require explanations of the thought process and intentions behind the final artifacts are also highly effective. Ideally, AI systems should keep a record of their work process for a certain period, enabling developers to ask detailed questions about the reasoning and intentions behind the artifacts when reviewing the work results.

Log Artifacts and Processes:

It is important to retain a record of the work process in addition to the artifacts generated by AI. This allows developers to refer to the AI's work history later. Through these records, they can understand when and how each component of the software was introduced. This knowledge is not only useful for developers when reviewing the AI's output, but also for making decisions in future projects. If the AI-generated source code follows a certain implementation pattern, it is not always clear whether it was intentional or simply selected from many possible options. By referring to the accumulated logs, developers can better understand the AI's intentions at the time. If the former is true, then guidelines can be established indicating that the same pattern should be used in future development.

Furthermore, these work records can serve as important input for future AI utilization. Suppose it is discovered that a particular design policy does not meet the requirements when the AI makes design decisions. By keeping logs of such considerations, the AI can avoid repeating the same mistakes in similar tasks in the future. In this way, having a mechanism to transfer knowledge from one AI to another is crucial for the long-term utilization of AI, which cannot inherently store tacit knowledge.

6 Responsibility Towards AI-Native Software Development

What responsibility should we have as AI transforms software development? Our significant responsibility in the accelerated software development process driven by AI is to appropriately manage the information generated by AI and achieve sustainable development. The following are main approaches:

Conduct Regular Review Meetings:

Holding regular meetings within the development team to understand the current software is another means. By discussing the current design and implementation of the parts of the software generated by AI, developers can increase their understanding of the results of AI‘s work. This work functions as knowledge transfer from AI to developers and can convert the information generated by AI into more usable knowledge. If developers discover improvement points for the current design and implementation through these meetings, they should manage them as a backlog. Addressing this backlog enables continuous quality improvement.

Enhance Developers’ Review Skills:

Improving the skills of reviewers is more important than ever to reduce knowledge debt. In addition to the quality assurance perspective that has been emphasized in traditional development, it is important for reviewers to strategically acquire knowledge from AI. The acquired knowledge should be managed as explicit knowledge within the team and made available for future development. At this time, it is desirable for reviewers to have the skill to document the knowledge acquired from AI appropriately. Also, there are cases where the points to check when confirming the artifacts generated by AI are different from those in traditional cases. For example, since AI's output depends on the training data, there is a possibility that AI may output incorrect results even in cases that human developers can easily avoid. There may be cases where variables unrelated to the task are refactored. When the source code of a web screen is generated, there is a possibility that the layout will be significantly distorted when rendered in a browser.

As mentioned above, there is an information gap between AI and human developers, and there are cases where we can anticipate that AI's output does not meet the requirements. For instance, if AI does not have access to information about how to create variations in tests, it is expected that AI can only generate variations that do not meet the project's criteria. Review techniques based on the nature of AI should be acquired by many developers in the future.

7 Conclusion

This paper has discussed the current state and future of software development in the AI era, focusing on the knowledge gap between AI and human developers. While AI systems for software development are advancing, reproducing the knowledge possessed by human developers is not straightforward. Only when the knowledge accessible to AI systems is the same as that of developers can AI take the lead in development tasks. On the other hand, as AI performs a wider range of tasks, there is a risk that the knowledge held by developers will gradually diminish. To avoid this risk, developers should strategically acquire and manage project-specific information more than ever before. Such control of AI utilization is the key to achieving sustainable and mature software development.

Finally, the insights in this paper are based on the experience of utilizing generative AI in software development at NTT DATA. We have utilized generative AI in numerous development projects as experts in software engineering. In projects that utilize generative AI, it is essential to optimize the balance between the benefits and risks of AI, and a systematic approach based on the expertise of software engineering is more useful than ever before. Even in the AI era, NTT DATA will continue to provide the value of responsible software development backed by highly specialized technology and experience.

References

1. GitHub Copilot. https://github.com/features/copilot.
2. Devin AI. https://devin.ai/.
3. Cline - Autonomous Coding Agent for VSCode. https://cline.bot/.
4. Cursor - The AI Code Editor. https://www.cursor.com/en.
5. Aider. https://aider.chat/.
6. Qodo Cover. https://github.com/qodo-ai/qodo-cover.
7. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph, Jan 2025, https://openreview.net/forum?id=dw9VUsSHGB
8. Model Context Protocol. https://spec.modelcontextprotocol.io/latest.
9. A Spiral Model of Software Development and Enhancement. May 1988. https://ieeexplore.ieee.org/document/59
10. Towards AI-Native Software Engineering (SE 3.0): A Vision and a Challenge Roadmap. October 2024. https://arxiv.org/abs/2410.06107

Author

Keita Takenouchi

Technology Strategist
(Software Engineering)