On Automated Assistants for Software Development: The Role of LLMs

Mira Leung
University of British Columbia
Vancouver, BC
miralng@cs.ubc.ca

Gail Murphy
University of British Columbia
Vancouver, BC
murphy@cs.ubc.ca

***Abstract*—Software developers handle many complex tasks that include gathering and applying domain knowledge, coordinating subtasks, designing interfaces, turning ideas into elegant code, and more. They must switch contexts between these tasks, incurring more cognitive costs. Recent advances in large language models (LLMs) open up new possibilities for moving beyond the support provided by automated assistants (AAs) available today. In this paper, we explore if a human memory model can provide a framework for the systematic investigation of AAs for software development based on LLMs and other new technologies.**

***Index Terms*—automation, machine learning, artificial intelligence, large language models, software development productivity**

I. Introduction

The ability to automate software engineering (SE) tasks has increased substantially with recent advances in generating code with large language models (LLMs). Emerging research on LLMs in SE has explored areas such as developer-LLM interaction interfaces [1], [2], test case generation [3], and quality and security concerns of LLM-generated code [4]–[6].

In this paper, we step back and ask if a model of human memory can help guide the development of future AAs for SE based on LLMs or other emerging technologies. The model we explore is Adaptive Control of Thought Rational (ACT-R) [7], which has distinct modules for declarative, production, and working memory. Among other cognitive models, ACT-R's hybrid architecture best balances production rules with information structuring, and aims to match human cognition [8].

We use the model to describe modular automation in the context of a common SE task, namely fixing a race condition bug (Section II). We then introduce a framework to describe three tiers of AAs for SE based on the model (Section III), explain how it applies to the above bug (Section IV) and raise open questions for future work (Section V). We weave earlier efforts applying cognitive models to SE tasks and AA support throughout. Our paper makes three contributions:

- We describe how the ACT-R memory model applies to cognitive processing needed in software development.
- We present a novel, ACT-R-based framework to describe how AAs might augment a developer's cognitive capacity.
- We describe a structured roadmap for future work in automated assistants for SE, mainly those based on LLMs.

This work was partly funded via NSERC by a post-graduate scholarship and a grant (RGPIN-2022-03139). The first author is also affiliated with Google.

II. ACT-R and SE

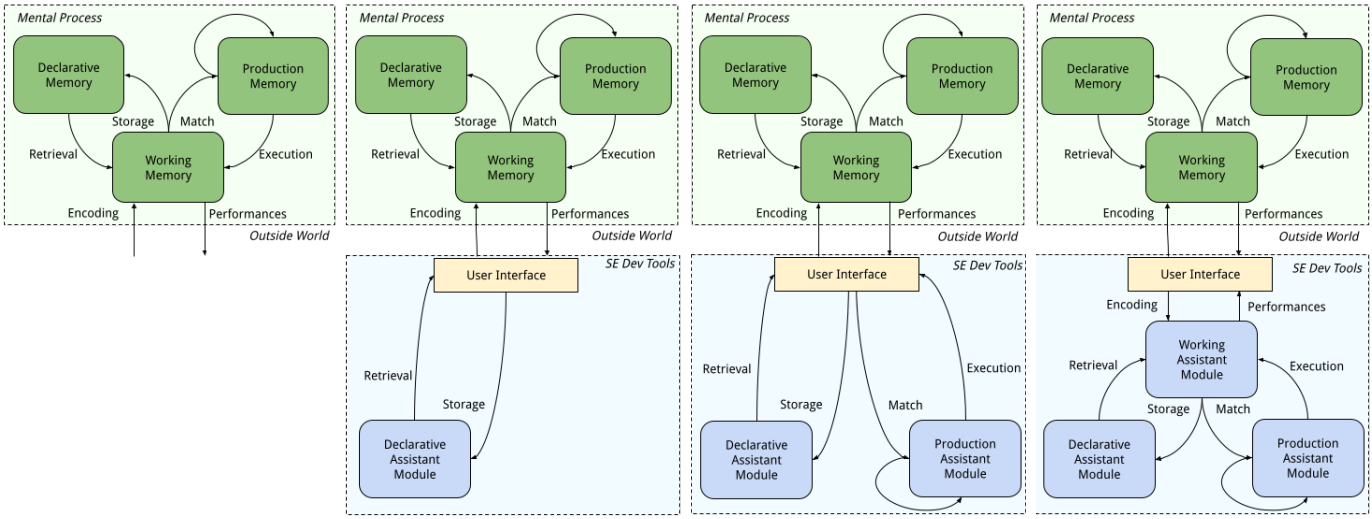
Researchers have long been interested in the cognitive processes at play in software development. For example, Robillard discussed key concepts such as declarative memory, chunking and planning, and their relation to software development [9]. Parnin applied cognitive concepts to understand how developers work on long-term tasks, despite frequent interruptions and task switches [10]. Others considered the cognitive processes used for specific tasks such as program comprehension [11]. More recently, researchers sought to link aspects of cognition theory with brain activity through fMRI studies [12].

Common to many of the models explored about cognitive processes in software development are the ideas that developers rely on knowledge, of both a given project and on domain background, and rely on strategies or procedures to use this information to complete tasks. These ideas are captured and structured by Anderson's ACT-R model that describes how the brain structures knowledge [7].

We use this model in a novel way to explore how a developer's work might be enhanced through automation. Much of the automation work in SE has focused on replacing a single kind of task, such as generating tests or refactoring code [3], [13]. When looking at these tasks from the perspective of the ACT-R memory model, we see an approach where assistants instead automate *pieces* of these tasks, then work together towards full automation. By focusing on individual parts, it may be possible to build assistants that can be tailored to specific projects, yet generalized to support automation of more kinds of tasks, working in concert with the developer.

In the ACT-R model (Fig. 1a), *declarative memory* holds facts, or declarative knowledge, which can be temporal. An example of a long-lived fact is the binary search algorithm, whereas a short-lived one might consist of the value of program variables in a given loop iteration. *Production memory* applies transformations to declarative knowledge. Finally, *working memory* performs higher-level functions such as coordinating the declarative and production memory modules, storing interim results, and activating subsequent operations [14].

To illustrate how this model maps to a software development task, consider a race condition bug in a Java project. A present-day developer may fix this bug with the simplified journey below, with relevant memory modules shown in parentheses.



(a) ACT-R model. (b) Framework tier 1. (c) Framework tier 2 (d) Framework tier 3.

Fig. 1: ACT-R as applied to human cognition (1a) and our developer-tool framework (1b-1d).

T.A (Declarative/Working): Refresh knowledge on race conditions, and understand the logic at potential error sites.
T.B (Production/Working): Rank potential error sites in descending order of likelihood.
T.C (Declarative/Working): Understand runtime behavior of the top-ranked site from *T.B*. Assuming this is the culprit, look up example fixes.
T.D (Production/Working): Design the fix.
T.E (Production/Working): Implement the approach from *T.D*.
T.F (Production/Working): Write preventative tests.

A developer tackling this bug must process different kinds of information as they perform these tasks, decide next steps, and apply transformations. ACT-R suggests that human cognition power is proportional to the amount of knowledge stored and how effectively it is deployed, which leads us to ask: Can AAs increase a developer's cognitive power?

III. Developer-Tool Interaction Tiers in SE

We introduce a framework, depicted in Fig. 1, that describes three tiers in which modules, such as AAs, could help extend the cognition of a software developer based on the ACT-R model. We illustrate this framework by exploring requirements for these modules and anchoring them in present-day tools. The first tier of the framework introduces a module that extends the developer's declarative memory, the second tier another module that supplements their production memory, and in the third tier, a corresponding module that augments their working memory. We describe each tier in turn.

A. Tier 1: Declarative Memory Extension

In this tier (Fig. 1b), declarative memory is extended with an external module that makes it easy to store and retrieve declarative knowledge through an integrated development environment (IDE) or a similar set of tools. For this module to be most helpful to the developer, it should gather and present all task-relevant declarative knowledge at once, which reduces context-switching for the developer while expanding the capacity of their declarative memory.

Using the race condition example from Section II, a present-day developer's initial declarative knowledge might consist of the following facts: there are two or more threads accessing one or more values, and multithreaded code areas are potential causes. The developer might act upon this starting point to inform how they gather bits of knowledge for subtasks *T.A* and *T.C* via internet searches and code lookup tools. Afterwards, they might keep these knowledge bits at hand across many web browser and code editor tabs for easy access, and to support synthesizing varied facts in their declarative memory.

B. Tier 2: Production Memory Extension

Tier 2 (Fig. 1c) would augment Tier 1's capabilities by adding an externalized production module, again accessed through tooling like an IDE, to alleviate the developer's production memory by taking on transformation sub-tasks such as architecture and code design, implementation, and writing tests. A production module can most benefit a developer by reducing the manual labor needed to transform declarative knowledge, as long as it is provided with a description of the expected output. This module can go beyond the principles in recommenders and refactoring tools [13], [15] to handle a wider variety of software development tasks.

Today, a developer working on the race condition bug would have to rely on their mental production memory to apply the transformations in *T.B*, *T.D*, *T.E*, and *T.F*, as bug localization tools have not yet been widely adopted by industry developers.

C. Tier 3: Working Memory Extension

This tier (Fig. 1d) would add an external working memory module to perform higher-level functions such as coordinating between the declarative and production modules from prior

tiers, deciding what knowledge or transformations are needed, and keeping state of task status. These capabilities extend the developer's cognitive capacity by alleviating the mental overhead involved in coordination activities.

Taking *T.A* as an example, the developer's working memory manages information gathering by evaluating if sufficient knowledge has been gathered, and whether to continue the search. In *T.B*, the working memory gives the list of snippets to production memory with the transformation to apply.

Developers today rely primarily on their own working memory to manage and context-switch between declarative and production memory. Empirical studies of developers' work in depth [16] help inform the kinds of tools that are likely needed to support a developer's working memory.

IV. A Future World with Automated Assistants

We use the race condition example from Section II to revisit each tier, to illustrate what each of their AAs can be, and to consider how they can elevate a developer's cognitive power.

A. Tier 1 Example

In an environment with a declarative memory assistant, instead of sequentially executing several internet and codebase searches for *T.A*, a developer might pose a single question, such as "*Provide context on race conditions and potential error sites in my codebase*." In response, the assistant would search the internet to provide a refresher on race conditions in Java, search the codebase for multithreaded areas, and return all this information together. For *T.C*, the developer might ask the declarative assistant to help explain the result, to which it might invoke a debugger and report back on stateful behavior such as the value of variables in the ith iteration of a loop.

By automating information-finding activities and reducing the number of round trips between the developer's declarative and working memory, this declarative assistant can help a developer complete a task more efficiently.

B. Tier 2 Example

A second assistant would be added to Tier 1 to augment the developer's production memory abilities. A developer working on subtask *T.B* would retrieve a list of all multithreaded code areas from the declarative assistant, then ask the production assistant to transform them into a ranked list of probabilities where the bug is most likely to occur. For *T.D*, the developer would ask the assistant to transform facts from the declarative assistant into a design for the selected approach.

Production assistants introduce the potential for parallelized workflows. For instance, the assistant could apply several mitigation options in *T.B*, instead of just implementing only one, which shares similar concepts with speculative execution [17]. This parallelization means the developer can examine the concrete code of multiple solutions and their runtime behavior to choose the best one. Presently, implementing code requires significant developer time, which gives rise to the current workflow of picking only one to implement based on design analysis. However, if the chosen solution needs to be replaced due to unanticipated side effects, the developer will have wasted their time on implementing the initial option. Tasks that involve reducing execution time or machine memory usage would especially benefit from this approach, since the chosen solution's interaction with existing behavior can worsen performance in ways that were not anticipated during the pre-implementation design stage.

C. Tier 3 Example

Imagine an IDE with a fully-featured working memory assistant added to the production and declarative assistants from Tier 2; an example interaction with the developer on the race condition but is shown in Fig. 2. The developer would start by asking the working memory assistant to fix the race condition (Step 1 in Fig. 2), to which the assistant would interact with the declarative and production assistants to obtain the most likely error sites (Steps 2 and 3, which map to *T.A* and *T.B*). The working memory assistant would then confirm the sites to fix with the developer, who picks one (Step 4 and 5) based on environmental factors such as the code's intended use. Next, the assistant asks the developer to pick the most appropriate fix from the ones suggested by the declarative assistant (Steps 6-8, equal to *T.C*), determines a task list and the corresponding transformations to apply (Step 9), then asks the production assistant to execute them (Step 10, analogous to *T.D* to *T.F*), finishing the task.

The working memory assistant should consult the developer for decisions that only a human can make. Examples include (a) where a fix *should* be applied, (b) what *kind* of fix is correct, and (c) what *end-user needs* to consider. For the race condition bug, the developer might answer with (a) a race condition bug-fix is not applicable to stateless code, (b) a generic Java fix may not work for an Android app, and (c) runtime latency is not an acceptable trade-off. These environmental factors may fluctuate in task-specific and project-specific ways, so the assistant should know when to seek human intervention.

V. The Role of LLMs in SE - A Roadmap

The concepts of the assistants in each tier are tantalizing, Can recent advances in LLMs help build these assistants? In this section, we focus on unique opportunities for LLMs based on mapping the ACT-R model to software development. We recognize the challenges identified elsewhere with LLMs, such as hallucination or security against prompt injection attacks [18], but do not focus on them in this short paper.

A. Declarative Assistant

LLMs can effectively summarize knowledge [19], making them a good foundation for declarative assistants. Instead of developers having to find and assemble disparate bits of information like API documentation or code snippets, LLMs can leapfrog that by providing answers to high-level questions.

Currently, LLMs are focused not only on extracting and assembling bits of knowledge, but also on synthesis. For the purposes of a declarative assistant, it might be helpful for LLM technology to produce bits of knowledge without

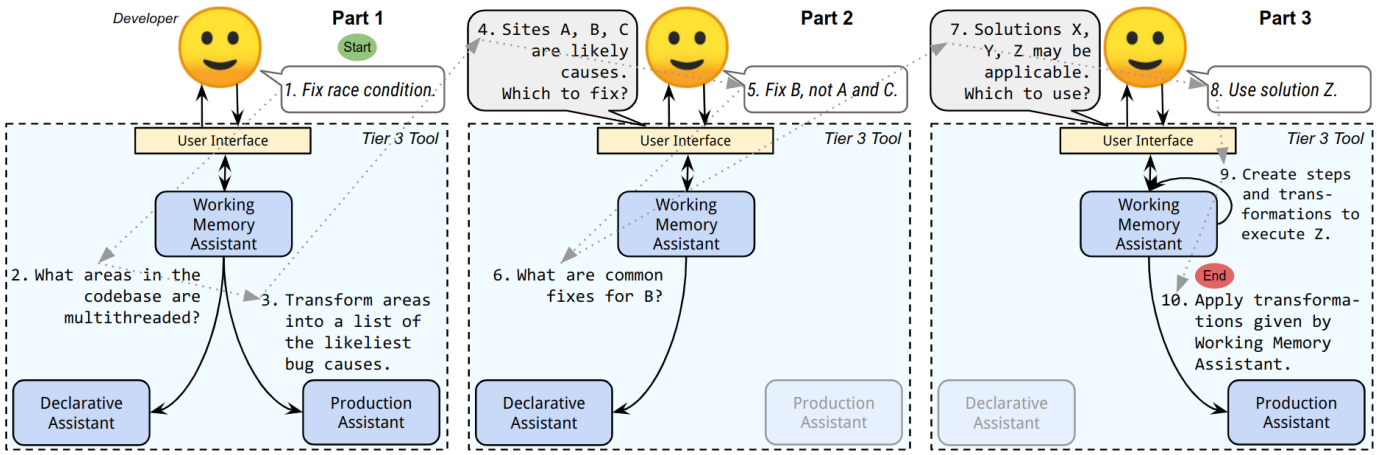


Fig. 2: Simplified Tier 3 developer-tool workflow for the race condition bug. Steps are shown with numbers and grey arrows.

synthesizing them together, or synthesizing smaller bits at a time. A focus on smaller nuggets of synthesized facts may be easier for a developer or tool to verify. Nascent efforts are emerging in this area, such as recent end-user tools that begin to provide interactive citations in LLM-generated content [20]. However, verifying bits of knowledge still requires a broader investigation into what sources are considered reliable, and what text metrics or human-computer interaction models can ensure and inspire human confidence in factual accuracy.

Open questions remain on how LLMs can help build such a declarative assistant. For one, how can an LLM be customized to a developer's project environment, like language or library versions, so that returned answers are relevant? As another example, what is the most appropriate way to present information so the developer can process it easily? A helpful format could involve connecting nodes of knowledge pieces as a network graph, a mental structure that is currently hard to decipher from the content generated by present-day LLMs.

B. Production Assistant

Production assistants focus on applying transformations to declarative knowledge, which can be built upon LLMs since they can perform transformations like natural language translations or code generation [6], [21]. While developer tasks also involve transformations, bridging the large gap between an initial problem statement (e.g., fix a race condition bug) and its final solution requires us to address many open questions.

Could LLMs be helpful for a production memory assistant if specialized for particular tasks, like separating design production from code generation? Or should there be access to the internal steps of an LLM? How could correctness be managed in this LLM, such that its transformations always produce valid and correct results, or emits errors otherwise? How could we ensure the assistant can detect, and return errors, when it is given insufficient input or bad transformations?

C. Working Memory Assistant

Are LLMs appropriate as working memory assistants? Early experiments such as Auto-GPT signal possible directions for future research, but recent evaluations show there exists room for growth [22]. Working memory necessitates comprehension of a task and its domain, strategizing ways to solve the task, and adapting to environmental factors such as problem constraints. LLMs currently lack these and other higher level cognitive functions typically attributed to Artificial General Intelligence (AGI) [23]. Thus, AGI may be a more appropriate building block for working memory assistants.

How could an AGI assistant know how to adapt problem-solving strategies or make conceptual leaps while working on task? Unlike well-defined domains like chess, software development tasks can be highly varied depending on factors like program behavior, library dependencies, and external world constraints, to name a few. Can project management and decision-making skills be developed in an AGI assistant, such that it can and adapt its instructions to mitigate any deficiencies in the output of the declarative and production assistants? If a generated solution is insufficient, how might this AGI assistant know when, and where, in the lifecycle of a task to solicit human intervention? Further, how would it learn from its experiences to improve continuously for future tasks? Challenges like these remain to advance AGI assistants, particularly for building the strategizing and cognitive abilities needed to meaningfully assist software developers.

VI. Conclusion

Applying an ACT-R model to the work of a software developer shows promise for teasing apart the different kinds of cognitive support needed to augment a developer's capabilities. Viewing a developer's work through this model suggests ways in which LLMs can best augment human capabilities through modular automation of parts or all of software development tasks. The path to build these assistants raises exciting challenges for the future research needed to achieve this vision.

VII. Acknowledgment

We gratefully acknowledge the reviews and helpful comments by reviewers on an earlier version of this paper.

REFERENCES

[1] S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, "The programmers assistant: Conversational interaction with a large language model for software development," in *Proceedings of the 28th International Conference on Intelligent User Interfaces*, 2023, pp. 491–514.

[2] A. M. McNutt, C. Wang, R. A. Deline, and S. M. Drucker, "On the design of ai-powered code assistants for notebooks," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–16.

[3] Z. Khaliq, S. U. Farooq, and D. A. Khan, "Transformers for gui testing: A plausible solution to automated test case generation and flaky tests," *Computer*, vol. 55, no. 3, pp. 64–73, 2022.

[4] S. Houde, V. Radhakrishna, P. Reddy, J. Darwade, H. Hu, K. Krishna, M. Agarwal, K. Talamadupula, and J. Weisz, "User and technical perspectives of controllable code generation," in *Annual Conference on Neural Information Processing Systems*, 2022.

[5] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" *arXiv preprint arXiv:2211.03622*, 2022.

[6] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt," *arXiv preprint arXiv:2304.10778*, 2023.

[7] J. R. Anderson, "Production systems and the act-r theory," *Rules of the mind*, pp. 17–44, 1993.

[8] F. E. Ritter, F. Tehranchi, and J. D. Oury, "Act-r: A cognitive architecture for modeling cognition," *Wiley Interdisciplinary Reviews: Cognitive Science*, vol. 10, no. 3, p. e1488, 2019.

[9] P. N. Robillard, "The role of knowledge in software development," *Communications of the ACM*, vol. 42, no. 1, pp. 87–92, 1999.

[10] C. Parnin, "A cognitive neuroscience perspective on memory for programming tasks."

[11] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[12] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann, "Measuring neural efficiency of program comprehension," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 140–150.

[13] J. Hannemann, G. C. Murphy, and G. Kiczales, "Role-based refactoring of crosscutting concerns," in *Proceedings of the 4th international conference on Aspect-oriented software development*, 2005, pp. 135–146.

[14] M. C. Lovett, L. M. Reder, and C. Lebiere, "Modeling working memory in a unified architecture: An act-r perspective." 1999.

[15] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 3, pp. 1–35, 2011.

[16] S. Chattopadhyay, N. Nelson, Y. R. Gonzalez, A. A. Leon, R. Pandita, and A. Sarma, "Latent patterns in activities: A field study of how developers manage context," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 373–383.

[17] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis of integrated development environment recommendations," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 669–682, 2012.

[18] F. Perez and I. Ribeiro, "Ignore previous prompt: Attack techniques for language models," *arXiv preprint arXiv:2211.09527*, 2022.

[19] Z. Luo, Q. Xie, and S. Ananiadou, "Chatgpt as a factual inconsistency evaluator for abstractive text summarization," *arXiv preprint arXiv:2303.15621*, 2023.

[20] S. Hsiao, "Whats ahead for bard: More global, more visual, more integrated," May 2023. [Online]. Available: https://blog.google/technology/ai/google-bard-updates-io-2023

[21] W. Jiao, W. Wang, J.-t. Huang, X. Wang, and Z. Tu, "Is chatgpt a good translator? a preliminary study," *arXiv preprint arXiv:2301.08745*, 2023.

[22] H. Yang, S. Yue, and Y. He, "Auto-gpt for online decision making: Benchmarks and additional opinions," *arXiv preprint arXiv:2306.02224*, 2023.

[23] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.