

A Viable Paradigm of Software Automation: Iterative End-to-End Automated Software Development

Jia Li¹, Zhi Jin^{1,2,*}, Huangzhao Zhang³, Kechi Zhang², Jiaru Qian¹, Tiankuo Zhao¹

¹School of Computer Science, Wuhan University, China

²Key Lab of High Confidence Software Technology (PKU), Ministry of Education
School of Computer Science, Peking University, China

³Independent

jia.li@whu.edu.cn, zhijin@whu.edu.cn

Abstract

Software development automation is a long-term goal in software engineering. With the development of artificial intelligence (AI), more and more researchers are exploring approaches to software automation. They view AI systems as tools or assistants in software development, still requiring significant human involvement. Another initiative is “vibe coding”, where AI systems write and repeatedly revise most (or even all) of the code. We foresee these two development paths will converge towards the same destination: AI systems participate in throughout the software development lifecycle, expanding boundaries of full-stack software development. In this paper, we present a vision of an iterative end-to-end automated software development paradigm AutoSW. It operates in an analyze-plan-implement-deliver loop, where AI systems as human partners become first-class actors, translating human intentions expressed in natural language into executable software. We explore a lightweight prototype across the paradigm and initially execute various representative cases. The results indicate that AutoSW can successfully deliver executable software, providing a feasible direction for truly end-to-end automated software development.

CCS Concepts

• **Software and its engineering** → **Software automation**; *Automated software development*.

Keywords

Software Automation, Software Development, Agent System

ACM Reference Format:

Jia Li¹, Zhi Jin^{1,2,*}, Huangzhao Zhang³, Kechi Zhang², Jiaru Qian¹, Tiankuo Zhao¹, ¹School of Computer Science, Wuhan University, China, ²Key Lab of High Confidence Software Technology (PKU), Ministry of Education, School of Computer Science, Peking University, China, ³Independent, jia.li@whu.edu.cn, zhijin@whu.edu.cn. 2018. A Viable Paradigm of Software Automation: Iterative End-to-End Automated Software Development. In *Proceedings of Make sure to enter the correct conference title from your*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06

<https://doi.org/XXXXXXX.XXXXXXX>

rights confirmation email (Conference acronym 'XX). ACM, New York, NY, USA, 5 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

Hand over the key of software development to users.

— Ruqian Lu

Software is the soul of the digital world and the cornerstone of contemporary society [10, 12, 17]. Currently, breakthroughs in artificial intelligence (AI), especially large language models (LLMs) and agent technology, have provided unprecedented hope for promoting software development [4, 6, 8, 16]. A growing number of researchers and industry professionals have explored development approaches that take advantage of the understanding, reasoning, and generation capabilities of LLMs, with the aim of achieving automated software development [24, 35, 36].

AI systems increasingly serve as an auxiliary tool throughout the software lifecycle [18, 21], helping tasks ranging from requirements analysis to code generation and bug fixing [20, 33]. A typical case is automated code completion [19, 32], where models predict tokens or lines given the code context. Although the workload is reduced as a result, human developers remain indispensable, as they need to provide supervision and intervention at each stage and during transitions between stages [26, 28].

Meanwhile, in the past year, early adopters have been energetically championing the “vibe coding” initiative [2, 3, 5, 7], in which LLMs write and iteratively revise most (or even all) of the code, while human beings steer via natural language prompts. The approach positions human beings as product managers while offloading implementation to LLMs. However, current practice often exhibits substantial variability and ad-hoc workflows, raising in the produced programs that are highly coupled and raising concerns about maintainability, quality, and comprehensibility.

We foresee these two lines of development converging on the same destination: AI systems participate across the entire software lifecycle, expanding the boundaries of full-stack software development. From the earliest days of computing, programmers interacted with machines via punched cards and paper tape; the advent of magnetic disks and interactive terminals reduced manual toil and enabled assembly programming; later, compilers and higher-level languages displaced assembly for most purposes. We argue that we are now at a similar inflection point: a new paradigm of software automation is coming into view, in which AI systems as human

*Corresponding author.

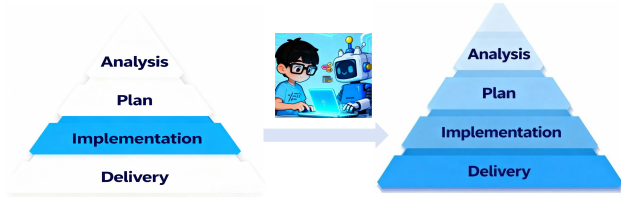


Figure 1: Transformation of automated software development paradigm.

partners become first-class actors in design, implementation, and maintenance. Humans play the role of primary drivers and supervisors in software development. Looking ahead, AI systems are poised to form a new abstraction layer for software development, translating the intent of humans expressed in natural language into executable software.

In this paper, we present a vision of an iterative end-to-end automated software development paradigm, named **AUTO**SW. This paradigm operates in an analyze-plan-implement-deliver loop (see Figure 2), and applies to both greenfield and incremental development. ❶ *Analysis*. AI systems communicate with the human to collect the underlying intent and analyze the requirements gathered. ❷ *Plan*. Human provides physical constraints in the real-world to AI systems, aligning on the goals of selecting the appropriate technology stack, proposing an architecture design, and eventually producing an implementation plan. ❸ *Implementation*. Once aligned, AI systems take over and automatically generate source code along with tests, where all functionalities and non-functional requirements are conformed. ❹ *Delivery*. After all tests and verifications have been completed, AI systems produce the deliverable package (code, test reports, deployment status, etc.) and return it to the human. Through repeated iterations of this loop, the system converges on complex domain-specific software that faithfully conforms to the user’s intent.

Figure 1 shows that AUTO

SW automates the entire software development process, rather than focusing only on code implementation like most existing approaches [15, 23, 29, 34, 35]. Compared to the prevailing “vibe coding” paradigm, AUTOSW places a stronger emphasis on software quality. It clearly defines the workflow for automated software development and designs a reasonable way to involve human participants. Through intent alignment, transparent delivery, and traceable automation, AUTOSW realizes human requirements in an iterative end-to-end fashion. Please refer to §2 for a detailed introduction to AUTOSW, §3 for representative examples in scenarios, and §4 for retrospective and forward-looking discussions of future research directions.

2 AUTO

SW

This section presents AUTO

SW, an iterative end-to-end automated software development paradigm based on the orchestral agent.

2.1 Overview

AUTO

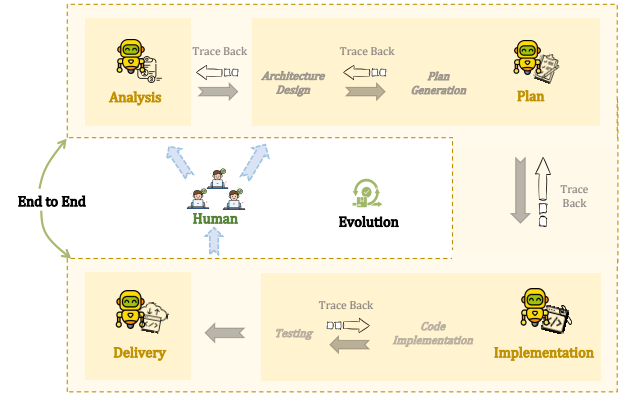
SW is outlined in Figure 2, where the AI system as human partner becomes a first-class actor in software automation. Inspired by the classic software development lifecycle, AUTOSW includes four


Figure 2: Overview of the iterative end-to-end automated software development paradigm AUTOSW, operating in an analyze-plan-implement-deliver loop.

stages: *Analysis*, *Plan*, *Implementation*, and *Delivery*. In the process, AUTO

SW maximizes the experience and automation capabilities of the AI system, while focusing on humans to provide constructive information through the conversation with the AI system.

2.2 Human

As the primary drivers of software development, humans provide constructive information to the AI system through conversation at two stages: describing requirements at *Analysis* stage, and providing the real-world physical constraints related to the software for system design at *Plan* stage, jointly completing software development from scratch or subsequent evolution.

In *Analysis* stage, humans describe new or changed requirements to the AI system for requirement analysis, while the AI system proactively initiates conversation by asking questions to progressively achieve requirements clarification. This allows humans to concentrate on describing “What” they want, rather than learning “How” to provide structured, unambiguous requirements like professional developers, significantly reducing the demand for human expertise.

Considering that system design is fraught with complex trade-offs [11, 14, 30], relying solely on the AI system for design decisions often overlooks real-world physical or business constraints (e.g. hardware limitations and specific regulatory requests). AUTO

SW situates the second human intervention in System Design in *Plan* stage. This intervention focuses humans on providing real-world physical conditions, ensuring the rationality of the selected technology stack and the feasibility of AI-produced architectural decisions.

In AUTO

SW, humans participate only in indispensable operations and are not required to have knowledge about professional development, maximizing the automation of software development.

2.3 AI

The AI partner in AUTO

SW is an orchestral agent system. Based on requirements and real-world physical constraints provided by humans, the agent system automatically executes the four stages

as described in §2.1, and finally delivers the deployed software to users. In this section, we elaborate on each stage.

2.3.1 Analysis. The failure in software development usually stems from the ambiguity or contradiction of requirements. In this stage, the AI system interacts with users to gather the underlying intent and analyze the collected requirements, ultimately producing a standardized requirements document. ❶ **Requirement Elicitation.** The agent gradually collects and clarifies user requirements through active questioning, which mimics the real-world communication pattern between developers and users (*i.e.* concretizing the vague requirements through conversation). This manner supports humans in describing what they want rather than providing structured requirements, allowing laypeople to participate in developing their desired software as well. ❷ **Requirement Document Generation.** Based on the conversation content, a standardized requirements document is generated that serves as a critical bridge between user intent and engineering practice. The document includes an overall functional description of the software and a series of user stories. To ensure traceability, each user story has a unique number. ❸ **Validation.** The system then validates requirements from three aspects: correctness, completeness, and consistency, preventing requirement errors from propagating to downstream stages, thus saving significant repair costs.

2.3.2 Plan. This stage aims to create a plan for writing code, involving System Design and Plan Generation, which is pivotal for software quality and maintainability. This is the last stage where humans interact with the AI system in order to provide physical constraints related to the software. Once the plan is generated, AUTO SW proceeds to *Implementation* stage.

System Design. Software development, especially intricate software, is inseparable from system design [13, 31]. The AI system performs the following actions: ❶ **Physical Constraint Acquisition.** Obtain real-world physical constraints related to software through active conversation with users, since they are crucial for selecting the technology stack and architecture. ❷ **System Design.** Generate an appropriate technology stack and architectures (*i.e.* 4+1 View Model [22]) based on requirements and physical constraints. ❸ **Validation.** Evaluate the system design generated. If errors are discovered, traceable remediation is performed to ensure consistency in all stages.

Plan Generation. Plan can help the agent organize the implementation process and prevent any omission of requirements. The AI system generates a series of executable tasks that satisfy the following requests: ❶ Each task must be explicit, enabling programming without further clarification. ❷ Each task is associated with one or more user stories, which supports a clear traceability matrix from requirements to tasks. ❸ The task list must cover all user stories, ensuring that the software can meet the user's intent. ❹ The generated plan is not a random list of tasks, instead, it strictly adheres to the dependencies and modularization defined in the architecture. This ensures the logic and feasibility of implementation, preventing being highly coupled or produced programs. The system then verifies the plan. If errors are found, not only the plan is changed, but the initial error point is also traced back for modification.

2.3.3 Implementation. The AI system takes over the plans and automatically generates source code along with tests. This stage contains Code Implementation and Testing.

Code Implementation. The AI system is responsible for implementing the tasks defined in the plan, focusing on one task at a time. When users propose new requirements or occur requirement changes, the system must first update the requirement and plan, instead of directly modifying programs.

Testing. Testing is a key contributor to software quality [27]. The AI system evaluates programs from two perspectives: ❶ **Functionality Testing.** It automatically generates test cases based on user stories and acceptance criteria produced by requirement document, testing whether the software behavior meets users' intent or if any redundant functionalities are introduced. ❷ **Vulnerability Detection.** Our ultimate goal is that non-professional humans can also use AUTO SW to build their desired software, where they typically provide function-related requirements but cannot mention quality-related demands (*e.g.* preventing SQL injection and XSS attacks). The agent also checks for vulnerabilities to ensure that the software delivered is robust and secure. Once the test fails, it will automatically trigger the traceability mechanism to correct all related errors until the code passes all test cases and does not have vulnerabilities.

2.3.4 Delivery. There is a gap between the tested code and the executable software. AUTO SW should deliver usable software to users, especially for laypeople, thus achieving an iterative end-to-end loop from the intent of humans to applicable software. In this stage, the AI system completes a series of deployment actions, such as compilation, packaging, configuring servers, and starting services. After deployment, humans can directly use the software and generate feedback. The feedback is then inputted into the AI system through conversations, triggering a new loop if necessary.

2.4 Key Features

The AI system serving as a human partner in AUTO SW is first-class and participates in the entire software development process, which maximizes the automation abilities of the AI system, while focusing on humans providing constructive information through conversation with them in specific stages. In this paper, AUTO SW has three key features.

- **Traceability.** AUTO SW supports traceability from requirements to testing, where each task in the plan is associated with user stories and can be further traced to code and test cases. When requirements change or errors occur, AUTO SW can precisely locate affected components in all stages. This allows for minimal and exact modifications, rather than redeveloping the software.
- **Software Evolution.** Inspired by the fact that users usually change their requirements, our AUTO SW supports software evolution. Users only need to provide new requirements to AUTO SW. It will update the corresponding content across relevant stages and then directly delivers a new version of the software to users.
- **End-to-End Automated Development.** Unlike most existing tools, AUTO SW starts with the user's intent and ultimately delivers the deployed software. It eliminates the barrier for software development from requiring development knowledge to ideas, empowering any human being to develop their own software.

3 Case Realization

3.1 Experimental Setup

To evaluate the feasibility of AUTO_{SW}, we initially implemented a lightweight prototype. In AUTO_{SW}, the orchestral agent is instantiated in the state-of-the-art LLM (i.e. GPT-5 [1]). The LLM has impressive understanding, generation, and reasoning abilities, which can support the agent in developing reliable software. We execute four representative and realistic scenarios with AUTO_{SW}. Each case is evaluated by two PhD students majoring in computer science. They verify whether the outcomes satisfy the users' intent.

3.2 Case Results

We present the results of AUTO_{SW} and analyze their feasibility. For input, we provide summarized user requirements and physical constraints for convenient demonstration.

3.2.1 Game Development. User Input. *Develop a Mine Sweeper game. It enables players to discover mines on the grid and choose different difficulty levels. During game play, it displays the remaining mine count and time statistics.*

Results. AUTO_{SW} automatically selected the model-view-controller design pattern, dividing the software into a game state management module, a cell-state tracking module, and a user interaction control module. Simultaneously, it separated the game logic from the user interface (UI) rendering and user input, ultimately delivering a fully functional game with smooth interaction.

Finding 1: AUTO_{SW} is capable of independently designing and implementing state-driven game, successfully converting complex rules of game into operational prototypes.

3.2.2 Management System. User Input. *Develop a digital artifact management system that supports the management of multiple types of data, including artifact information, membership data, and user feedback. The data is presented through a visual interface. Meanwhile, it enables users to quickly switch between various data modules through interactive operations, with the aim of meeting the daily management of museums.*

Results. AUTO_{SW} identified this scenario as a data-intensive application and designed a data-based layout. It successfully integrated a chart library to support data visualization and implemented dynamic filtering logic to support data module switch, where input changes would trigger real-time updates of the data charts.

Finding 2: AUTO_{SW} can build data-intensive applications. The software supports various data operations such as management, analysis, filtering, and visualization.

3.2.3 Personal Assistant. User Input. *Develop a travel assistant. It helps users plan their travel itineraries. The assistant first recommends a list of popular attractions and organizes them into an itinerary. When users select an attraction, it can display related weather, flight, accommodation, and navigation information for the location.*

Results. AUTO_{SW} successfully designed a multi-modular application, creating distinct modules for attractions, weather, tickets, accommodation, and other functions. The application utilizes a central manager to handle state synchronization across modules. When users selected a location in the attractions module, the manager was

updated, which in turn automatically triggered the re-rendering of other related modules.

Finding 3: AUTO_{SW} is capable of developing multi-module applications. The application can effectively keep the communication between modules and the synchronization of their states.

3.2.4 Application Service. User Input. *Develop a tax filing system. It supports users to complete their tax payments, which enables users to fill out personal information and then automatically analyze their income and generate tax filing materials. To ensure its reliability, the system provides relevant tax policies, explanations, and a Q&A module to users.*

Results. AUTO_{SW} demonstrates an excellent ability to understand and apply domain-specific knowledge, abstracting complex tax policies into a tax filing system that strictly adheres to tax laws. The software not only presents the final results, but also generates a step-by-step breakdown of the calculation process. Additionally, AUTO_{SW} autonomously identify sensitive information and implement client-side encryption to protect user privacy.

Finding 4: AUTO_{SW} is able of converting professional and complex domain knowledge (such as tax policies) into precise algorithms and programs. Furthermore, it can autonomously identify and implement non-functional requirements.

These results show that AUTO_{SW} can complete automated software development in various scenarios, such as state-driven games, data-intensive management systems, multi-module systems, and domain-specific applications. They provide strong empirical support for AUTO_{SW} as a viable paradigm of software automation.

4 Challenges and Future Directions

Although the results of AUTO_{SW} are feasible, there are still some challenges for practical large-scale applications. We outline several directions for future research.

Scalability to Complex Scenarios. The current exploration focuses primarily on lightweight prototypes. In the future, we will extend AUTO_{SW} to complex software scenarios and evaluate its reliability in large-scale applications.

Better Form of Human-AI Interaction. In AUTO_{SW}, humans interact with AI system by multi-turn conversations. We will explore better human-AI interaction methods, such as formulating questionnaires that humans only need to fill out quickly, aiming to make it more convenient for humans to interact with AI systems.

Context Management. During software development, a large amount of context is generated, such as plans, code, test cases, and conversation histories. When developing complex software, it is possible that the context is too long, leading to forgetting long content, such as lost-in-the-middle [9, 25]. We will explore the appropriate memory modules to effectively manage context.

Benchmarking. The systematically validating of the new paradigm remains an open problem. We will publish a standardized benchmark and design all-sided metrics to evaluate AUTO_{SW}. The benchmark will consist of various software scenarios with varying complexity.

Enhancing Reliability and Robustness. We will investigate adversarial approaches to enhance its robustness. For example, we

will require humans to provide contradictory requirements, asking AutoSW to seek clarification rather than directly generating flawed software. Meanwhile, We will adopt a more comprehensive software verification strategy to improve reliability.

5 Conclusion

This paper presents a vision of an iterative end-to-end automated software development paradigm AutoSW. It operates in an analyze-plan-implement-deliver loop, placing a stronger emphasis on software quality. In the loop, the AI system serves as the human partner participating in the entire software development process. We implement a lightweight prototype and find that AutoSW can successfully deliver satisfactory software in various scenarios.

References

- [1] 2024. GPT-5. <https://openai.com/index/introducing-gpt-5/> (2024).
- [2] 2025. Augment Code. <https://www.augmentcode.com/> (2025).
- [3] 2025. Claud Code. <https://www.anthropic.com/claude/code> (2025).
- [4] 2025. Claude Sonnet 4. <https://www.anthropic.com/claude/sonnet> (2025).
- [5] 2025. Cursor. <https://cursor.com/cn> (2025).
- [6] 2025. Gemini-2.5-Pro. <https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-pro> (2025).
- [7] 2025. Windsurf. <https://windsurf.com/> (2025).
- [8] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altmenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
- [9] George Arthur Baker, Ankush Raut, Sagi Shaier, Lawrence E Hunter, and Katharina von der Wense. 2024. Lost in the Middle, and In-Between: Enhancing Language Models' Ability to Reason Over Long Contexts in Multi-Hop QA. *arXiv preprint arXiv:2412.10079* (2024).
- [10] Vladana Celebic. 2022. The Role of Software Engineering in Society 5.0.
- [11] Jim E Cooling. 2013. *Software design for real-time systems*. Springer.
- [12] Ronnie de Souza Santos, Italo Santos, Robson Santos, and Cleyton Magalhaes. 2025. Hidden Figures in Software Engineering: A Replication Study Exploring Undergraduate Software Students' Awareness of Distinguished Scientists from Underrepresented Groups. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 786–796.
- [13] David Garlan and Dewayne E Perry. 1995. Introduction to the special issue on software architecture. *IEEE Trans. Software Eng.* 21, 4 (1995), 269–274.
- [14] Hassan Gomaa. 1994. Software design methods for the design of large-scale real-time systems. *Journal of Systems and Software* 25, 2 (1994), 127–146.
- [15] Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. Longcoder: A long-range pre-trained language model for code completion. In *International Conference on Machine Learning*. PMLR, 12098–12107.
- [16] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).
- [17] William E Halal. 1993. The information technology revolution: Computer hardware, software, and services into the 21st century. *Technological Forecasting and Social Change* 44, 1 (1993), 69–86.
- [18] Ahmed E Hassan, Gustavo A Oliva, Dayi Lin, Boyuan Chen, Zhen Ming, et al. 2024. Towards AI-native software engineering (SE 3.0): A vision and a challenge roadmap. *arXiv preprint arXiv:2410.06107* (2024).
- [19] Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. 2025. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces* 92 (2025), 103917.
- [20] Dongming Jin, Zhi Jin, Xiaohong Chen, and Chunhui Wang. 2024. Mare: Multi-agents collaboration framework for requirements engineering. *arXiv preprint arXiv:2405.03256* (2024).
- [21] Mansi Khemka and Brian Houck. 2024. Toward Effective AI Support for Developers: A survey of desires and concerns. *Commun. ACM* 67, 11 (2024), 42–49.
- [22] P Krutchen. 1995. Architectural blueprints—the “4+1” view model of software architecture. *IEEE software* 12, 6 (1995), 42–50.
- [23] Jia Li, Chongyang Tao, Zhi Jin, Fang Liu, and Ge Li. 2023. ZC 3: Zero-shot cross-language code clone detection. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 875–887.
- [24] Jia Li, Chongyang Tao, Jia Li, Ge Li, Zhi Jin, Huangzhao Zhang, Zheng Fang, and Fang Liu. 2025. Large language model-aware in-context learning for code generation. *ACM Transactions on Software Engineering and Methodology* 34, 7 (2025), 1–33.
- [25] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172* (2023).
- [26] Yasir Mahmood, Nazri Kama, Azri Azmi, Ahmad Salman Khan, and Mazlan Ali. 2022. Software effort estimation accuracy prediction of machine learning techniques: A systematic performance evaluation. *Software: Practice and experience* 52, 1 (2022), 39–65.
- [27] S Murugesan. 1994. Attitude towards testing: a key contributor to software quality. In *Proceedings of 1994 1st International Conference on Software Testing, Reliability and Quality Assurance (STRQA'94)*. IEEE, 111–115.
- [28] Ketai Qiu, Niccolò Puccinelli, Matteo Ciniselli, and Luca Di Grazia. 2025. From today's code to tomorrow's symphony: The AI transformation of developer's routine by 2030. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–17.
- [29] Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. 2025. Paper2code: Automating code generation from scientific papers in machine learning. *arXiv preprint arXiv:2504.17192* (2025).
- [30] Eduardo Tavares, P Maciel, Pedro Dallegrave, Bruno Silva, Tiago Falcão, B Nogueira, G Callou, and P Cunha. 2010. Model-driven software synthesis for hard real-time applications with energy constraints. *Design Automation for Embedded Systems* 14, 4 (2010), 327–366.
- [31] Zhiyuan Wan, Yun Zhang, Xin Xia, Yi Jiang, and David Lo. 2023. Software architecture in practice: Challenges and opportunities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1457–1469.
- [32] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. RlCoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487* (2024).
- [33] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024).
- [34] Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2024. Codedpo: Aligning code models with self generated and verified source code. *arXiv preprint arXiv:2410.05605* (2024).
- [35] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339* (2024).
- [36] Yuqi Zhu, Jia Li, Ge Li, Yunfei Zhao, Zhi Jin, and Hong Mei. 2024. Hot or cold? adaptive temperature sampling for code generation with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 437–445.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009