THE USA JOURNALS
THE AMERICAN JOURNAL OF ENGINEERING AND TECHNOLOGY (ISSN – 2689-0984)
VOLUME 06 ISSUE12

PUBLISHED DATE: - 31-12-2024 **PAGE NO.: - 222-229**

RESEARCH ARTICLE **Open Access**

The Evolution of Cognitive Software Engineering: A Longitudinal Analysis of Large Language Models and Machine Learning in Architectural Synthesis, Fault Prediction, and Code Comprehension

Aristhanes K. Vardhaman
Department of Computer Science and Software Engineering, Global Institute of Technological Innovation, Singapore

Abstract
The landscape of software engineering is currently undergoing a fundamental paradigm shift, transitioning from manual, heuristic-driven development to an automated, cognitive-centric model powered by Large Language Models (LLMs) and Machine Learning (ML). This research article provides an extensive investigation into the integration of neural code comprehension and generative artificial intelligence across the software development lifecycle. By synthesizing contemporary advancements in architectural pattern detection, fault prediction, and automated code repair, this study elucidates how modern AI architectures-ranging from bilateral tree-based convolutional neural networks to transformer-based few-shot learners-are redefining the boundaries of software assurance and system design. We examine the transition from traditional source code metrics to learnable representations of code semantics, discussing the implications of neuro-symbolic program correctors and graph-based generative modeling. Furthermore, the paper addresses the emerging role of LLMs in identifying architectural smells, refactoring microservices, and maintaining consistency in low-code platforms. Through a rigorous analysis of existing empirical evidence and theoretical frameworks, this research identifies a critical "automation gap" in software architecting and proposes a trajectory for future autonomous systems. The findings suggest that while AI significantly enhances productivity and fault detection, issues regarding software fairness, carbon footprints, and the nuances of cross-language algorithm classification remain pivotal challenges for the next decade of academic and industrial pursuit.

Keywords Neural Code Comprehension, Large Language Models, Software Architecture, Fault Prediction, Machine Learning, Generative AI, Microservices.

INTRODUCTION

The quest for computational systems that can understand, generate, and repair their own source code has been a cornerstone of computer science since the mid-twentieth century. However, it is only within the last decade that the convergence of massive datasets, high-performance computing, and sophisticated neural architectures has transformed this vision into a functional reality. Historically, software engineering relied on static analysis and manual code reviews to ensure system integrity. While effective to a degree, these methods often failed to scale with the exponential complexity of modern distributed systems and microservices. The introduction of machine learning into the software domain marked the beginning of a move toward "Cognitive Software Engineering," where the system itself assists in the creative and corrective processes of development.

At the heart of this evolution is the concept of neural code comprehension. Traditional compilers treat code as a structured string of tokens, but as Ben-Nun et al. (2018) argue, code possesses a learnable representation of semantics that goes beyond simple syntax. This shift in perspective allows for the development of models that do not just read code, but understand its intent. Such understanding is crucial for tasks like software fault prediction, where the goal is to identify potential vulnerabilities before they manifest as system failures. Bhandari and Gupta (2018) have demonstrated that machine learning models utilizing source code metrics can predict faults with higher accuracy than human-centric heuristics, particularly when these models are trained on diverse datasets like the Juliet C/C++ and Java test suites (Boland and Black, 2012).

The problem, however, remains multifaceted. As software systems become more modular, the complexity of managing architectural integrity grows. The transition from monolithic architectures to microservices has introduced new challenges in pattern detection and service boundary identification. Duarte (2025) highlights the role of infrastructure-as-code artifacts and LLMs in detecting microservice patterns, suggesting that the "glue" holding modern systems together is increasingly becoming a subject for AI-driven analysis. Despite these gains, a significant literature gap exists regarding the "automation gap" in software architecting-the space between generating code snippets and synthesizing entire, coherent software architectures. Ivers and Ozkaya (2025) question whether generative AI can truly fill this gap, or if it merely provides a more sophisticated form of template-filling.

Furthermore, the ethical and environmental dimensions of this technological surge cannot be ignored. The growing carbon footprint of generative AI (An et al., 2023) and the necessity for software fairness (Brun and Meliou, 2018) introduce a socio-technical layer to the research. We must ask not only if a model can predict a fault, but if it does so without bias and at a sustainable energy cost. This article navigates these complex intersections, providing a comprehensive overview of the current state of the art and a roadmap for the future of AI-assisted software engineering.

METHODOLOGY

The methodology employed in this research follows a multi-dimensional systematic analysis of recent breakthroughs in machine learning (ML) and large language models (LLMs) as applied to software engineering. The study is grounded in a rigorous review of foundational datasets and modern empirical studies, specifically focusing on the transition from token-based analysis to semantic-based neural representations.

The primary framework for evaluating code comprehension models is based on the work of Ben-Nun et al. (2018), which utilizes an Intermediate Representation (IR) to capture both the control flow and data flow of programs. By abstracting source code into a learnable format, we can assess the efficacy of deep learning models in tasks ranging from algorithm classification to vulnerability detection. This research contrasts these IR-based approaches with more recent transformer models, such as GPT-3 and its successors, which utilize few-shot learning to perform code-related tasks with minimal task-specific training (Brown et al., 2020).

To investigate the predictive capabilities of ML in software maintenance, we examine the utilization of software metrics as input features for fault prediction models. Bhandari and Gupta (2018) provide a baseline for this by using traditional source code metrics, which are then compared against mutation-aware fault prediction techniques (Bowes et al., 2016). The methodology also incorporates an analysis of the "Juliet" test suite (Boland and Black, 2012) and the SAMATE reference dataset (Black, 2007), which provide the standardized "ground truth" necessary for training supervised learning algorithms.

In the realm of software architecture, the methodology shifts toward structural and graph-based representations. Brockschmidt et al. (2019) and Brauckmann et al. (2020) emphasize that code is inherently non-linear; therefore, generative models must account for the graph-like structure of

software. We analyze the effectiveness of these graph-based models in extracting architectural components (Fuchs et al., 2025) and mapping source code to higher-level architectural abstractions (Johansson et al., 2024). This involve a comparative study of prompt pattern sequences (Maranhão and Guerra, 2024) and their ability to assist in architectural decision-making.

Finally, the study addresses the comparative performance of various LLM iterations, including GPT-3.5, GPT-4, and GPT-4o, specifically in the context of specialized domains like emergency medicine software and microservice query understanding (Liu et al., 2024; Quevedo et al., 2024). The methodology concludes with an evaluation of how these models handle "code smells" and "architectural smells," using systematic mapping studies to categorize the effectiveness of different ML techniques in detecting technical debt (Caram et al., 2019a; Pandini et al., 2025).

RESULTS

The findings of this study reveal a significant evolution in the capacity of AI to handle the nuances of software development. The results are categorized into four primary domains: Neural Representation and Comprehension, Fault and Vulnerability Prediction, Automated Synthesis and Repair, and Architectural Evolution.

Neural Representation and Comprehension One of the most profound results is the shift from character-level language modeling to high-level semantic representation. Early models focused on program synthesis for character-level tasks (Bielik et al., 2017), but these often failed to capture the logical intent of the programmer. The work of Ben-Nun et al. (2018) demonstrates that by using a learnable representation of code semantics, AI can achieve a deeper "understanding" of the code's purpose. This is further evidenced by the success of bilateral tree-based convolutional neural networks in cross-language program classification (Bui et al., 2018). These models are capable of identifying the same algorithm (e.g., QuickSort) regardless of whether it is written in C++ or Java, by focusing on the underlying structural dependencies (Bui et al., 2019).

Theoretical elaboration on this point suggests that code is not merely a language but a formal logic. While LLMs like GPT-3 (Brown et al., 2020) are trained on massive natural language corpora, their ability to generalize code tasks suggests that the underlying transformer architecture is adept at identifying the recursive and nested patterns inherent in programming. However, Cai et al. (2017) argue that for true generalization, neural programming architectures must incorporate recursion as a first-class citizen, rather than relying solely on the attention mechanisms of transformers.

Fault and Vulnerability Prediction The application of machine learning to software assurance has yielded highly accurate results in vulnerability prediction (Bilgin et al., 2020). By training on datasets like Juliet 1.1 (Boland and Black, 2012), models can now identify subtle security flaws that static analysis tools frequently miss. Bhandari and Gupta (2018) show that ML-based fault prediction utilizing source code metrics can significantly reduce the "mean time to repair" in industrial settings.

Furthermore, the integration of "mutation-aware" techniques (Bowes et al., 2016) has revolutionized the way we view code quality. Instead of looking for known patterns of failure, these models simulate potential faults to see how the system responds. This proactive approach is particularly useful in agile environments where rapid deployment is prioritized (Butgereit, 2019). The data indicates that machine learning is not just a tool for finding bugs, but a mechanism for prioritizing automated testing, ensuring that the most critical components of a system are tested first.

Automated Synthesis and Repair The results regarding automated program repair are particularly promising. Bhatia et al. (2018) introduced a neuro-symbolic program corrector that helps students in introductory programming assignments by providing automated feedback and corrections. This blend of neural "intuition" and symbolic "logic" allows the model to suggest fixes that are both syntactically correct and semantically relevant.

In more complex industrial applications, Cai et al.

(2019) demonstrated the use of model checking combined with machine learning to repair B-models. This suggests a future where AI does not just write code from scratch, but acts as a continuous "auto-repair" agent for existing legacy systems. Cambronero and Rinard (2019) have even moved toward "autogenerating" the supervised learning programs themselves, creating a meta-loop where AI builds the tools that help build the software.

Architectural Evolution and Microservices The most recent data (2024–2025) suggests that LLMs are now being applied to high-level architectural tasks. Duarte (2025) and Fuchs et al. (2025) have shown that LLMs can extract component names and detect microservice pattern instances with a high degree of accuracy. This is a significant leap from simple code completion (Bruch et al., 2009). The ability to map source code to software architecture (Johansson et al., 2024) allows developers to maintain a "living documentation" of their systems, reducing the gap between the designed architecture and the implemented code.

However, the results also highlight the "architectural smell" problem. Pandini et al. (2025) found that while LLMs can identify architectural smells-such as cyclic dependencies or overly complex hubs-refactoring these smells remains a challenge. The AI can point out the problem, but the holistic understanding required to refactor a distributed system without introducing regression remains a "human-in-the-loop" process for now.

DISCUSSION

The implications of these findings are vast, touching upon the very nature of what it means to be a "software engineer" in the age of AI. The discussion focuses on three critical areas: the automation gap, the carbon footprint of AI, and the necessity for fairness and consistency.

The Automation Gap and the Role of the Architect Despite the impressive performance of LLMs in generating code snippets and detecting faults, Ivers and Ozkaya (2025) rightly point out the "automation gap" in software architecting. Generating a function is a solved problem; generating a resilient, scalable, and secure cloud-native architecture is not. Architecture involves trade-offs-performance versus security, cost versus redundancy-that require contextual knowledge often missing from training data. Maranhão and Guerra (2024) suggest that prompt pattern sequences can help AI assist in decision-making, but the final "judgment call" remains a human responsibility.

The "state of practice" analysis by Jahić and Sami (2024) suggests that while LLMs are becoming ubiquitous in software engineering, their use in high-level architecture is still in its infancy. There is a risk that newcomers to the field might rely too heavily on AI-generated designs without understanding the underlying principles (Larsen and Edvall, 2024). This could lead to a generation of "copy-paste architects" who can assemble systems but cannot troubleshoot them when the AI's logic fails.

Environmental and Ethical Considerations An et al. (2023) bring a sobering perspective to the discussion: the carbon footprint of generative AI. The massive computational power required to train and run models like GPT-4 is significant. As we move toward more automated software engineering, we must weigh the productivity gains against the environmental costs. This leads to a theoretical counter-argument: perhaps the future of AI in software engineering is not in "larger" models, but in "smarter," more efficient neuro-symbolic models that require less data and power to achieve the same results.

Furthermore, software fairness is a growing concern. Brun and Meliou (2018) emphasize that ML models can inadvertently bake biases into software. If an AI is trained on code that contains biased algorithms (e.g., in hiring or credit scoring), it will perpetuate those biases in its own generations. Ensuring fairness in AI-generated code is not just a technical challenge, but a moral imperative.

The Future of Low-Code and Consistency Hagel et al. (2025) discuss the role of LLMs in ensuring consistency within model-based low-code platforms. This is a critical area for the future, as it democratizes software development. If AI can ensure that a "citizen developer" creates a system

that is architecturally sound and free of vulnerabilities, the barrier to entry for building complex software will vanish. However, maintaining consistency between a high-level model and the generated low-level code remains a technical hurdle that requires ongoing research.

Limitations and Future Scope A primary limitation of the current research is the "black box" nature of many LLMs. While we can measure their output, understanding the why behind a specific architectural recommendation remains difficult. Future research should focus on "explainable AI" for software engineering, where the model provides a rationale for its design choices. Additionally, the field of "software antipattern detection" (Miño et al., 2024) is ripe for expansion, as identifying what not to do is often as important as identifying what to do.

Another area for future investigation is the impact of AI on the psychological aspects of programming. How does the presence of a "perfect" AI assistant affect the creativity and problem-solving skills of a human developer? Larsen and Edvall (2024) provide an initial look at newcomers, but a longitudinal study on senior developers is needed.

CONCLUSION

The integration of Machine Learning and Large Language Models into software engineering represents a definitive turning point in the discipline's history. From the early days of simple source code metrics for fault prediction (Bhandari and Gupta, 2018) to the contemporary era of LLM-powered architectural synthesis (Duarte, 2025), the trajectory is clear: we are moving toward a symbiotic relationship between human intelligence and machine cognition.

Neural code comprehension has proven to be a robust foundation for understanding the semantics of programming (Ben-Nun et al., 2018), while graph-based generative models have paved the way for more sophisticated software synthesis (Brockschmidt et al., 2019). The ability of AI to detect vulnerabilities, correct errors in student code (Bhatia et al., 2018), and even refactor architectural smells (Pandini et al., 2025) demonstrates its versatility across the development lifecycle.

However, significant challenges remain. The "automation gap" in high-level architecting, the environmental impact of large-scale AI, and the critical need for software fairness and explainability must be addressed. As we look toward 2030 and beyond, the focus of the research community should shift from "can AI do this?" to "how can AI do this sustainably, fairly, and transparently?" The role of the software architect is not disappearing; rather, it is evolving into a role of high-level supervision, where the human provides the creative vision and the AI provides the technical execution and verification. In conclusion, the era of cognitive software engineering is not a future possibility-it is our current reality, and its successful navigation will define the next generation of technological progress.

REFERENCES

1. An, J., Ding, W., & Lin, C. (2023). ChatGPT: tackle the growing carbon footprint of generative AI. Nature, 615, 586.
2. Ben-Nun, T., Jakobovits, A. S., & Hoefler, T. (2018). Neural code comprehension: A learnable representation of code semantics. Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS '18), 3589-3601.
3. Bhandari, G. P., & Gupta, R. (2018). Machine learning based software fault prediction utilizing source code metrics. 2018 IEEE 3rd International Conference on Computing, Communication and Security (ICCCS), 40-45.
4. Bhatia, S., Kohli, P., & Singh, R. (2018). Neuro-symbolic program corrector for introductory programming assignments. Proceedings of the 40th International Conference on Software Engineering (ICSE '18), 60-70.
5. Bielik, P., Raychev, V., & Vechev, M. T. (2017). Program synthesis for character level language modeling. ICLR.
6. Bilgin, Z., Ersoy, M. A., Soykan, E. U., Tomur, E., Çomak, P., & Karaçay, L. (2020). Vulnerability prediction from source code using machine learning. IEEE Access, 8, 150672-150684.

7. Black, P. E. (2007). Software assurance with SAMATE reference dataset, tool standards, and studies.

8. Boland, F., & Black, P. (2012). The Juliet 1.1 C/C++ and Java test suite. IEEE Computer, 45, 10.1109/MC.2012.345.

9. Bowes, D., Hall, T., Harman, M., Jia, Y., Sarro, F., & Wu, F. (2016). Mutation-aware fault prediction. Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016), 330-341.

10. Braga, R., Neto, P. S., Rabêlo, R., Santiago, J., & Souza, M. (2018). A machine learning approach to generate test oracles. Proceedings of the XXXII Brazilian Symposium on Software Engineering (SBES '18), 142-151.

11. Brauckmann, A., Goens, A., Ertel, S., & Castrillon, J. (2020). Compiler-based graph representations for deep learning models of code. Proceedings of the 29th International Conference on Compiler Construction (CC 2020), 201-211.

12. Brockschmidt, M., Allamanis, M., & Gaunt, A. L. (2019). Generative code modeling with graphs. International Conference on Learning Representations (ICLR).

13. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. arXiv preprint arXiv:2005.14165.

14. Bruch, M., Monperrus, M., & Mezini, M. (2009). Learning from examples to improve code completion systems. Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '09), 213-222.

15. Brun, Y., & Meliou, A. (2018). Software fairness. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018), 754-759.

16. Bui, N. D. Q., Jiang, L., & Yu, Y. (2018). Cross-language learning for program classification using bilateral tree-based convolutional neural networks. AAAI Workshops.

17. Bui, N. D. Q., Yu, Y., & Jiang, L. (2019). Bilateral dependency neural networks for cross-language algorithm classification. 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 422-433.

18. Butgereit, L. (2019). Using machine learning to prioritize automated testing in an agile environment. 2019 Conference on Information Communications Technology and Society (ICTAS), 1-6.

19. Cai, J., Shin, R., & Song, D. (2017). Making neural programming architectures generalize via recursion. CoRR, abs/1704.06611.

20. Cai, C. H., Sun, J., & Dobbie, G. (2019). Automatic B-model repair using model checking and machine learning. Automated Software Engineering, 26(3), 10.1007/s10515-019-00264-4.

21. Cambronero, J. P., & Rinard, M. C. (2019). AL: autogenerating supervised learning programs. Proceedings of the ACM on Programming Languages, 3(OOPSLA), 1-28.

22. Caram, F. L., Rodrigues, B. R. O., Campanelli, A. S., & Parreiras, F. S. (2019a). Machine learning techniques for code smells detection: a systematic mapping study. International Journal of Software Engineering and Knowledge Engineering, 29(02), 285-316.

23. Duarte, C. E. (2025). Automated microservice pattern instance detection using infrastructure-as-code artifacts and large language models. 2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C), 161-166.

24. Eisenreich, T., Speth, S., & Wagner, S. (2024). From requirements to architecture: An ai-based journey to semi-automatically generate software architectures. Proceedings of the 1st International Workshop on Designing Software, 52-55.

25. Fauzan, R., Siahaan, D., Rochimah, S., & Triandini, E. (2024). Structural similarity

assessment for multiple UML diagrams measurement with UML common graph. AIP Conference Proceedings, 2927(1).

26. Feng, Y., Vanam, S., Cherukupally, M., Zheng, W., Qiu, M., & Chen, H. (2023). Investigating code generation performance of ChatGPT with crowdsourcing social data. 2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC), 876–885.

27. Fuchs, D., Liu, H., Hey, T., Keim, J., & Koziolek, A. (2025). Enabling architecture traceability by llm-based architecture component name extraction. 2025 IEEE 22nd International Conference on Software Architecture (ICSA), 1-12.

28. Hagel, N., Hili, N., Bartel, A., & Koziolek, A. (2025). Towards llm-powered consistency in model-based low-code platforms. 2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C), 364-369.

29. K. S. Hebbar, "MACHINE LEARNING-ASSISTED SERVICE BOUNDARY DETECTION FOR MODULARIZING LEGACY SYSTEMS," International Journal of Applied Engineering & Technology, vol. 04, no.02, pp. 401-414, Sep. 2022, https://romanpub.com/resources/ijaet-v4-2-2022-48.pdf

30. Ivers, J., & Ozkaya, I. (2025). Will generative ai fill the automation gap in software architecting? 2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C), 41-45.

31. Jahić, J., & Sami, A. (2024). State of practice: Llms in software engineering and software architecture. 2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C), 311–318.

32. Johansson, N., Caporuscio, M., & Olsson, T. (2024). Mapping source code to software architecture by leveraging large language models. Software Architecture. ECSA 2024 Tracks and Workshops, 133-149.

33. Larsen, K. R., & Edvall, M. (2024). Investigating the impact of generative ai on newcomers' understanding of software projects.

34. Liu, C. L., Ho, C. T., & Wu, T. C. (2024). Custom GPTs enhancing performance and evidence compared with GPT-3.5, GPT-4, and GPT-4o? A study on the emergency medicine specialist examination. Healthcare, 12(17), 1726.

35. Lutze, R., & Waldhör, K. (2024). Generating specifications from requirements documents for smart devices using large language models (llms). Human-Computer Interaction, Springer Nature Switzerland, 94-108.

36. Maranhão, J. J., & Guerra, E. M. (2024). A prompt pattern sequence approach to apply generative ai in assisting software architecture decision-making. Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices (EuroPLoP '24).

37. Miño, J., Andrade, R., Torres, J., & Chicaiza, K. (2024). Leveraging generative artificial intelligence for software antipattern detection. Information Management.

38. Nayak, M. (2024). How is the Artificial Intelligence of Today's Time, ChatGPT and Blackbox. ai, Helpful in Machine Learning?. ChatGPT and Blackbox. ai, Helpful in Machine Learning.

39. Pandini, G., Martini, A., Videsjorden, A. N., & Fontana, F. A. (2025). An exploratory study on architectural smell refactoring using large languages models. 2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C), 462–471.

40. Quevedo, E., Abdelfattah, A. S., Rodriguez, A., Yero, J., & Cerny, T. (2024). Evaluating chatgpt's proficiency in understanding and answering microservice architecture queries using source code insights. SN Computer Science, 5, 422.

41. Suriya, S., & Nivetha, S. (2023). Design of UML Diagrams for WEBMED-Healthcare Service System Services. EAI Endorsed Transactions on E-Learning, 8(1).

42. Triandini, E., Fauzan, R., Siahaan, D. O., Rochimah, S., Suardika, I. G., & Karolita, D. (2022). Software similarity measurements using UML diagrams: A systematic literature review. Register: Jurnal Ilmiah Teknologi

Sistem Informasi, 8(1), 10–23.