Revolutionizing Software Engineering with Large Language Models: Impacts Across the Development Lifecycle

Ragini Kalvade, University of Illinois at Chicago

I. INTRODUCTION

Large language models (LLMs) have introduced a new wave of transformation by enabling automation at a deeper cognitive level. LLMs can assist with complex tasks such as generating code snippets, suggesting improvements, automating code reviews, and even translating high-level requirements into functional code.

With progress being made in leaps and bounds, the role of a software engineer is gradually shifting from writing code to collaborating with AI-powered tools. It is crucial to understand how these tools impact the engineering workflow and how to integrate them. The introduction of such tools does not eliminate the necessity of an engineer's deep understanding of programming concepts. LLMs need to be leveraged as sounding boards for absolute truths, allowing engineers to focus on the human element of solution designs, critical analysis and creativity of the process. Engineers need to be aware of their capabilities and limitations to ensure they are used effectively without sacrificing quality or security.

The objective of this paper is to analyze the influence of Large Language Models (LLMs) on key stages of the software development lifecycle - requirements specification, code generation, testing, and code review. This paper aims to highlight how these processes are being automated, enhanced, and challenged in each phase, while addressing their ethical implications and inherent biases of the tools.

This analysis is structured as follows: Section II reviews how software requirements specifications (SRS) can be improved using natural language processing (NLP) techniques and briefly discusses the generation of functional code from high level requirements. Section III covers code generation tools and their potential as coding assistants. Section IV, discusses test case generation, bug fixing, and the ability of LLMs to parse business test cases. Section V focuses on code review automation tools, their effectiveness, and a case study of a Google tool. Finally, Section VI concludes the paper.

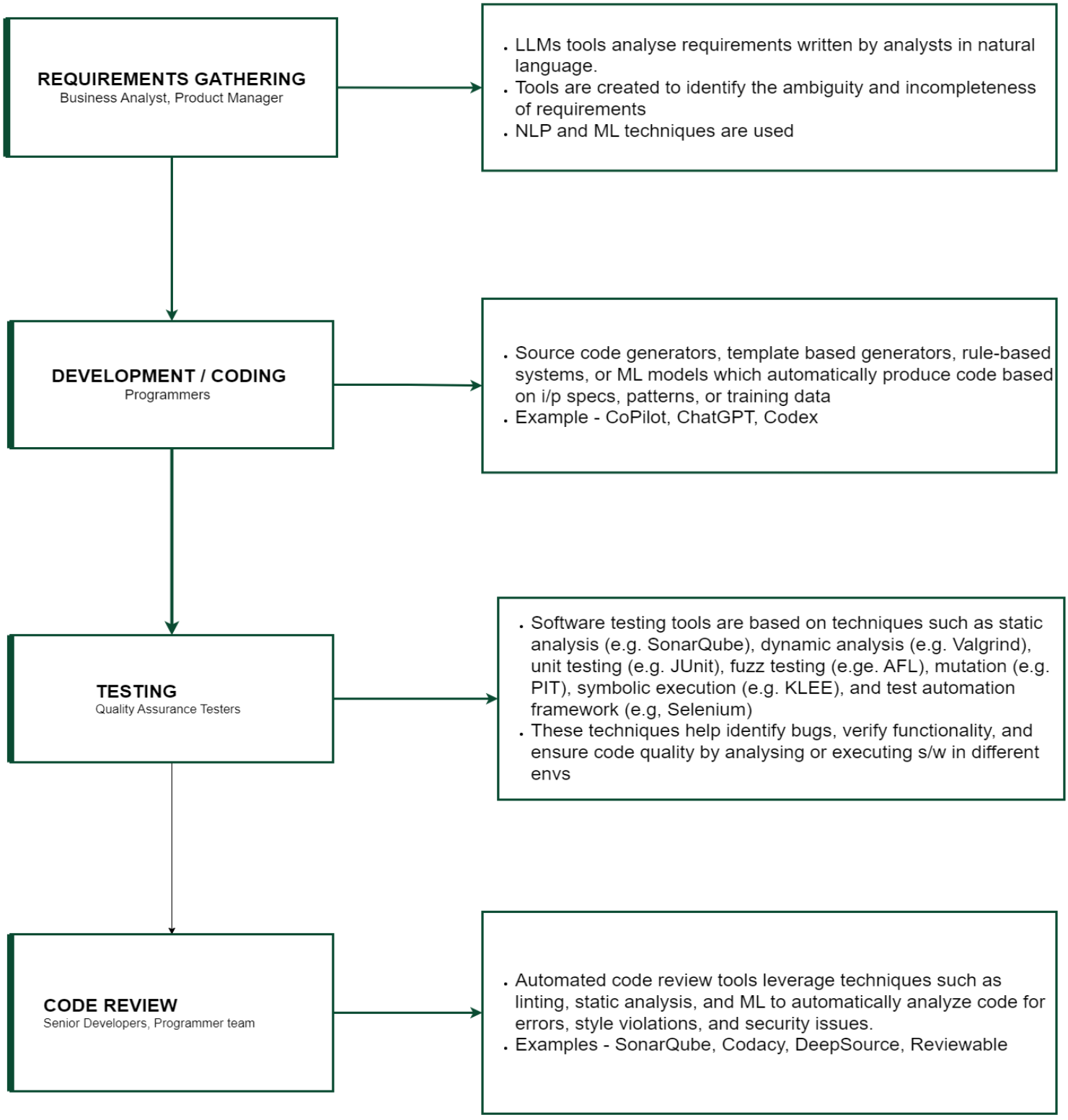

Figure 2: Simplified overview of LLM tools at each stage

I. IMPACT OF LLM ON REQUIREMENT SPECIFICATIONS

Failure to document requirements properly can set back a project by years. Due to the subjective analytical nature of this component, automation in requirements analysis has historically been limited due to the complexity and subjectivity involved, but advancements in Machine Learning (ML) and Natural Language Processing (NLP) are facilitating new approaches. The use of NLP allows for automated analysis of large volumes of requirements, reducing manual error and time spent on reviews. Previously, templates and guidelines have been proposed to ensure clarity but it is not ironclad. When mistakes occur during the requirements stage, and the succeeding steps are implemented, it often necessitates going back to the drawing board.

Analysis of requirements can be broken down into two notable components - Incompleteness and Ambiguity. *Incompleteness* arises from a lack of understanding of the user expectations and sound business operations knowledge. *Ambiguity* in requirements can lead to varied interpretations by different stakeholders, including developers and testers. This can lead to inconsistencies in both implementation and testing, ultimately resulting in defects and rework.

Requirements quality assurance tools analyze given requirements, and they differ in their approaches to defining ambiguity and incompleteness. For ambiguity, some tools utilize a dictionary to ensure that the natural language is expressed in a predefined grammar. Various techniques are employed for this, including part-of-speech (POS) tagging, lemmatization, keyword extraction, and machine learning (figure 2).

On the other hand, addressing incompleteness is more challenging, as it often requires domain-specific expertise that cannot be predefined in a tool. To provide reliable suggestions, tools need to be pre-trained using stories or data from similar products to help ensure that the requirements are comprehensive and well-defined.

Analysis of currently present/proposed requirement quality tools

Paska [1] : Paska is a tool that utilizes natural language processing (NLP) to identify requirement smells, including incompleteness, and offers suggestions based on a controlled natural language framework.

Smella [2] : Smella was designed for detecting requirement smells such as vagueness and incompleteness. Smella employs part-of-speech tagging and lemmatization techniques.
RequisitePro [3] : This is a requirements management tool that includes features for tracking completeness and linking requirements to ensure all necessary elements are present.

The above mentioned tools provide an edge over the traditional method which encompasses several structured techniques designed to anticipate and document stakeholder needs.

Traditionally, analysts conduct surveys and questionnaires to collect quantitative data, organize workshops to discuss requirements collectively and then do a document analysis to extract relevant information from existing materials to develop use cases or user stories to capture functional requirements, and create prototypes for visual feedback. Once this groundwork is complete there are consecutive rounds of back and forth between the technical team and analysts to negotiate and gain clarity.

The analysis of requirement tools helps to anticipate the more commonly seen problems and help bypass much of it. The requirement quality tools improve clarity by reducing ambiguity, ensure consistency in documentation, and provide enhanced traceability throughout the development lifecycle. Additionally, they help ensure compliance with industry standards.

However, to ensure that maximum potential is extracted from these tools, organizations must be prepared to invest in training and integration. An over-reliance on these tools must be discouraged to disabuse the notion of achieving 100% accuracy. Fostering a culture of continuous improvement in requirements management can allow users to enhance their processes and adapt to evolving needs effectively.

II. IMPACT OF LLMS ON THE CODING STAGE

LLMs are trained on huge amounts of code and data, allowing them to understand and generate programming code. They can assist developers by providing code suggestions, fixing bugs, or even writing entire sections of code based on a description given to them. For example, if you ask an AI to "write a function to add two numbers," it can quickly generate the necessary code in any supported language.This speeds up the coding process by offering quick solutions and code snippets. Developers spend less time on repetitive tasks and more time on complex problem-solving.

A source code generator is trained using a combination of supervised learning techniques and large-scale datasets of annotated code examples. The process begins with the collection of diverse and high-quality datasets consisting of programming languages, frameworks, and problem descriptions paired with corresponding code solutions. These datasets are often derived from open-source repositories, competitive programming archives, and educational resources. The model, typically a large-scale transformer architecture like GPT, is pre-trained on these datasets to capture the syntactic and semantic patterns of programming languages. During pre-training, the model learns to predict the next token in a sequence, enabling it to generate syntactically correct and semantically meaningful code snippets based on context.

Once pre-trained, the source code generator undergoes fine-tuning to specialize in specific tasks such as generating code from natural language descriptions or optimizing code based on performance metrics. Fine-tuning often involves task-specific datasets where examples are labeled with input-output pairs, such as a natural language query mapped to the corresponding code implementation. Reinforcement learning and user feedback are also used to refine the model further, aligning its outputs with real-world expectations. This comprehensive training approach ensures the generator not only produces syntactically accurate code but also adheres to best practices, follows established design patterns, and addresses the functional requirements specified in user prompts.

LLMs are reshaping the way coding is done. They're becoming a standard tool in many developers' toolkits, but there's a need for new rules and methods to ensure that their use remains safe and effective. For instance, teams now need to double-check AI-generated code for mistakes and security issues before using it in projects. There's also a risk that developers, especially those still learning, might depend too much on these AI tools. This can result in not developing a deep understanding of coding fundamentals, since they might rely on the AI to do the heavy lifting. In schools and universities, using AI tools blurs the line between using code as a resource and copying it outright. This raises concerns about originality and proper learning.

As AI tools continue to improve, they'll become even better at generating quality code. But to fully benefit from them, developers will need to understand how to work with AI effectively—balancing the convenience of AI assistance with the need for critical thinking and secure coding practices.

In simple terms, LLMs can be like having a smart assistant when writing code. They make the job faster and easier but can also introduce risks if not used carefully. Developers and students need to learn how to use these tools wisely, making sure they don't rely on them too much or ignore the potential security flaws they might introduce.

III. IMPACT OF LLMS ON SOFTWARE TESTING

LLMs like GPT-3 [11] and Codex [12] can automatically create unit tests and system test inputs. LLMs help in identifying and understanding bugs by analyzing error messages, suggest where the problem might be, and even propose fixes. This is particularly useful because debugging is often one of the most time-consuming parts of software development. LLMs can assist by providing explanations of what might be going wrong, making it easier for developers to track down the problem. *Fuzz testing* is a technique where random or unexpected inputs are fed into a software program to see if it breaks. LLMs can make fuzz testing more effective by generating a wide range of test inputs that are more diverse and realistic than those created by older random-based methods.

While LLMs are great at creating test cases, they don't always get everything right. Sometimes, the tests they create don't cover all possible problems or have errors themselves. Some parts of testing require understanding specific business rules or user needs, which can be hard for LLMs to grasp without very detailed instructions. For example, if a company's software has a special rule for calculating discounts, an LLM might not automatically understand how to test that.

Many testing teams are finding that LLMs work best when combined with older, proven methods like manual testing or automated scripts. They might use an LLM to generate an initial set of tests and then refine or extend those tests using traditional methods. This approach helps ensure that generated tests are accurate and thorough, while still saving time on the more repetitive parts of test writing.

The avenue of LLMs makes it easier to create and run tests, catch bugs, and ensure software quality. They help automate many tasks, but, similar to its impact in other stages, it is not a complete solution on its own—developers still need to carefully check their work to ensure everything is running smoothly.

IV. IMPACT OF LLMS ON CODE REVIEWS

Automation of code review has advanced, particularly with the introduction of deep learning models like Transformer models. These models target two main tasks:

Code & Comment-to-Code: Automatically implementing a revised code based on reviewer comments.

Code-to-Comment: Generating reviewer-like comments for the code that needs revision.

Automated code reviews have been helpful in simple scenarios, such as fixing syntax issues, minor refactoring, or formatting code. They can significantly reduce the human effort required for repetitive and small changes, such as removing unnecessary whitespace. Tools like ChatGPT [11] were also evaluated and found to perform decently on some tasks like code generation, but still lack the precision needed for detailed code reviews.

Most models overfit on simple examples and struggle with more complex code changes that require understanding the context across multiple components of the codebase. The current automation methods fall short when dealing with intricate refactorings or changes that span across multiple files or functions, as their "view" of the code is often limited to individual functions or methods. Additionally, there are data quality issues in the datasets used for training the models, such as the presence of erroneous or biased data that weakens the reliability of the automation.

Case Study: Google's Code Review Automation Using ML

Google published a research paper [13] that describes an internal ML based assistant used within the organisation to resolve code review comments. This tool focuses on applying ML to suggest code changes automatically based on reviewer feedback.

Google developers typically spend around 60 minutes on average addressing review comments per code change. Google's system applies ML to automatically suggest changes based on reviewer comments, reducing the time spent in the review process. The assistant works in two ways:

1. **Asynchronous Suggestions**: After a reviewer sends feedback, the assistant asynchronously generates potential code changes.

2. **Interactive Experience**: The assistant suggests edits directly to the reviewer in real-time, allowing them to provide feedback instantly.

Google's ML assistant is able to resolve about **7.5% of all reviewer comments** in production, saving developers hundreds of thousands of engineering hours annually. The system significantly reduces repetitive tasks like typo corrections or small refactorings, enabling engineers to focus on more complex and creative tasks. The assistant is integrated within Google's existing code review infrastructure, allowing seamless use within their monolithic code repository (Piper).

Like other ML-based tools, Google's system still faces challenges when handling ambiguous or complex review comments.The system's accuracy is dependent on the quality and size of the training data, which consists of millions of code reviews. Despite Google's vast dataset, there are still issues in certain contexts.The need for high precision in predictions sometimes limits the number of suggestions provided, especially for non-trivial cases where review comments require deeper context or domain-specific knowledge.

V. ETHICAL CONSIDERATIONS AND BIAS IN LLMS

The use of LLMs introduces a range of ethical concerns that warrant critical attention. Privacy issues arise when LLMs inadvertently expose sensitive or proprietary information embedded within their training datasets. This concern is especially pronounced in contexts where datasets contain personal data or confidential intellectual property, risking violations of privacy regulations and trust. Additionally, intellectual property concerns emerge, as generated content may inadvertently replicate copyrighted material, raising legal and ethical questions about ownership and originality. The dependency risks associated with LLMs are equally significant; over-reliance on these models can undermine users' development of critical thinking and domain-specific skills, potentially creating an overdependence on automated systems.

LLMs are also prone to bias, primarily stemming from their training data. Training data bias can lead to outputs that reflect societal prejudices, perpetuating discriminatory or inaccurate information. By amplifying stereotypes, LLMs risk reinforcing harmful societal narratives or unfair assumptions about individuals and groups. Furthermore, the limited representation of diverse languages, cultures, and knowledge domains in training data skews outputs, privileging dominant narratives while marginalizing less represented perspectives. Addressing these concerns requires implementing robust data curation practices, rigorous evaluation frameworks,

and interdisciplinary oversight to ensure LLMs are equitable, ethical, and aligned with societal values.

VI. CONCLUSION

LLMs are revolutionizing the software development lifecycle by automating key stages, including requirement specification, coding, testing, and code review. Their ability to analyze, generate, and improve outputs provides immense benefits, such as faster workflows, reduced manual effort, and enhanced productivity. Tools like Paska, Smella, and Google's ML-based assistant illustrate the practical application of LLMs in streamlining processes and improving quality across various stages. However, while LLMs excel in addressing repetitive and structured tasks, they fall short in managing complexity, domain-specific nuances, and tasks requiring deep contextual understanding.

The ethical and technical challenges associated with LLMs are equally critical. Issues such as privacy risks, intellectual property concerns, and dependency on automation demand careful consideration. Furthermore, biases in training data can perpetuate stereotypes and lead to inequitable outputs. These limitations underscore the need for balanced use, where LLMs augment human expertise rather than replace it.

Moving forward, integrating LLMs effectively requires rigorous oversight, continuous improvement, and investment in user training. By leveraging LLMs thoughtfully while addressing their limitations, developers and organizations can achieve a harmonious collaboration between AI and human creativity, fostering innovation without compromising ethical or technical standards.

REFERENCES

[1] A. Veizaga, S. Y. Shin and L. C. Briand, "Automated Smell Detection and Recommendation in Natural Language Requirements," in IEEE Transactions on Software Engineering, vol. 50, no. 4, pp. 695-720, April 2024, doi: 10.1109/TSE.2024.3361033.

[2] Henning Femmer, Daniel Méndez Fernández, Stefan Wagner, and Sebastian Eder. Rapid quality assurance with requirements smells. Journal of Systems and Software, 123:190–213, jan 2017. ISSN 01641212. doi: 10.1016/j.jss.2016.02.047.

[3] IBM. The IBM Rational RequisitePro.

[4] M. Zakeri-Nasrabadi and S. Parsa, “Natural Language Requirements Testability Measurement Based on Requirement Smells”, *arXiv [cs.SE]*. 2024.

[5] Yang, Chen et al. “Recent Advances in Intelligent Source Code Generation: A Survey on Natural Language Based Studies.” Entropy (Basel, Switzerland) vol. 23,9 1174. 7 Sep. 2021, doi:10.3390/e23091174 URL

[6] Judy Sheard, Paul Denny, Arto Hellas, Juho Leinonen, Lauri Malmi, and Simon. 2024. “Instructor Perceptions of AI Code Generation Tools - A Multi-Institutional Interview Study”. In Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024). Association for Computing Machinery, New York, NY, USA, 1223–1229

[7] Niclas Andersson, Peter Fritzson, "Overview and industrial application of code generator generators",Journal of Systems and Software, Volume 32, Issue 3, 1996,Pages185-214,ISSN0164-1212,https://doi.org/10.1016/0164-1212(95)00124-7

[8] Negri-Ribalta Claudia , Geraud-Stewart Rémi , Sergeeva Anastasia , Lenzini Gabriele “A systematic literature review on the impact of AI models on the security of code generation” JOURNAL=Frontiers in Big Data VOLUME=7 YEAR=2024 DOI=10.3389/fdata.2024.1386720ISSN=2624-909X

[9] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang and Q. Wang, "Software Testing With Large Language Models: Survey, Landscape, and Vision," in IEEE Transactions on Software Engineering, vol. 50, no. 4, pp. 911-936, April 2024, doi: 10.1109/TSE.2024.3368208.

[10] R. Tufano, O. Dabić, A. Mastropaolo, M. Ciniselli and G. Bavota, "Code Review Automation: Strengths and Weaknesses of the State of the Art," in IEEE Transactions on Software Engineering, vol. 50, no. 2, pp. 338-353, Feb. 2024, doi: 10.1109/TSE.2023.3348172.

[11] OpenAI ChatGPT. https://platform.openai.com/docs/api-reference

[12] Codex https://docs.codex.so/codex-docs

[13] Frömmgen, A., Austin, J., Choy, P., Ghelani, N., Kharatyan, L., Surita, G., Khrapko, E., Lamblin, P., Manzagol, P.-A., Revaj, M., Tabachnyk, M., Tarlow, D., Villela, K., Zheng, D., Chandra, S., & Maniatis, P. (2024). Resolving Code Review Comments with Machine Learning. 2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).