ORIGINAL ARTICLE
Electrica 2026; 26: 1-10

Enhancing Software Development with Large Language Models: A Case Study of Kolay.ai

Sadi Evren Şeker[1], **Hatice Nizam-Özoğur**[2]

[1]Department of Computer Engineering, İstanbul University Faculty of Computer and Information Technologies, istanbul, Türkiy
[2]Department of Artificial Intelligence and Data Engineering, , İstanbul University Faculty of Computer and Information Technologies, İstanbul, Türkiye

Cite this article as: S. E. Şeker and H. Nizam-Özoğur, "Enhancing software development with large language models: A case study of kolay.ai," *Electrica*, 26, 0033, 2026. doi:10.5152/electrica.2026.25033.

WHAT IS ALREADY KNOWN ON THIS TOPIC?

- *Large language models (LLMs) are increasingly used in software engineering to automate coding tasks, assist in documentation, and accelerate development processes.*
- *Despite their benefits, LLM-based tools face challenges such as generating inaccurate or hallucinated code, managing complex dependencies, and maintaining contextual consistency.*

Corresponding author:
Hatice Nizam Özoğur

E-mail:
hatice.nizamozogur@istanbul.edu.tr

Received: February 24, 2025
Revision Requested: April 29, 2025
Last Revision Received: September 9, 2025
Accepted: September 14, 2025
Publication Date: January 30, 2026

DOI: 10.5152/electrica.2026.25033

Content of this journal is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

ABSTRACT

The integration of large language models (LLMs) into software development has transformed the field by streamlining coding processes, reducing manual workload, and enabling automation of documentation and testing. This paper presents a detailed case study of Kolay.ai, a project built using LLM-based development tools. It demonstrates how LLMs accelerate development cycles by 30–40%, reduce errors by 30%, and improve onboarding efficiency. However, the study also identifies challenges such as hallucinated outputs, context management issues, and integration complexities, which require careful oversight through human-in-the-loop (HITL) workflows. To address these challenges, the project employed a modular development strategy, structured prompt libraries, and continuous monitoring techniques. The findings emphasize that while LLMs offer significant advantages, manual oversight remains essential for ensuring code quality, consistency, and security. This paper proposes practical solutions, including enhanced prompt engineering and memory-augmented LLMs, to optimize future LLM-based workflows. It concludes by highlighting the need for balanced collaboration between human developers and LLMs, paving the way for scalable, efficient, and adaptive software development.

***Index Terms*—**Code quality and consistency, human-in-the-loop (HITL), large language models (LLMs), modular development and integration, software development automation

I. INTRODUCTION

The adoption of large language models (LLMs) in software engineering is significantly transforming traditional software development paradigms. LLMs, which are trained on extensive datasets including code repositories, technical documentation, and programming manuals, have introduced innovative solutions aimed at accelerating development, automating repetitive tasks, and enhancing both testing and documentation processes [1]. Prominent tools, such as GitHub Copilot, ChatGPT, and Replit Ghostwriter, serve as examples of the growing integration of LLMs into developer workflows. However, despite their contribution to efficiency, these tools also present challenges, including the generation of hallucinated code, dependency management issues, and limitations in contextual awareness [2, 3].

This section provides a comprehensive review of the existing literature on the use of LLMs in software engineering, emphasizing both their potential benefits and the challenges they introduce. The aim is to position the Kolay.ai project within the broader context of LLM applications, offering both theoretical insights and addressing the practical obstacles encountered in real-world use cases. The following sections will delve into key themes, including the acceleration of software development and reduction of costs, the challenges of hallucinated code and contextual inconsistency, the enhancement of code quality and security, and the role of LLMs in automating documentation and facilitating collaboration. These discussions will provide a thorough understanding of the evolving role of LLMs in software engineering and highlight areas that require further exploration.

WHAT THIS STUDY ADDS ON THIS TOPIC?

- *This study introduces a novel integration of structured prompt engineering within modular architecture, enabling scalable and maintainable LLM-assisted development.*
- *The Kolay.ai project provides practical strategies for improving parallel development, testing, and deployment of artificial intelligence modules, bridging theoretical concepts with real-world applications.*

A. Accelerating Development and Reducing Costs

Research has shown that LLM-powered tools significantly enhance development speed. According to [4], LLM tools enable developers to automate repetitive coding tasks, allowing them to focus more on the complex and creative aspects of software design. The LLMs are particularly effective in generating boilerplate code, streamlining prototyping, and reducing the time required for debugging. Studies by [5] and [6] indicate that projects leveraging LLMs can achieve a 30–40% reduction in development time, particularly in agile environments with iterative sprint cycles. Furthermore, LLMs play a significant role in cost optimization by reducing the manual effort required for routine tasks, such as documentation and test generation [7]. Developers can use LLMs to quickly create and refine prototypes, thereby supporting faster project delivery while effectively managing operational costs.

B. Challenges: Hallucinated Code and Contextual Inconsistency

Despite their advantages, LLMs are susceptible to hallucinations, a phenomenon in which models generate plausible yet incorrect or misleading code snippets. The study by [8] suggests that such hallucinated outputs pose significant risks if not promptly identified, as they may lead to software defects or security vulnerabilities. Research by [2] indicates that LLM hallucinations occur more frequently in complex coding scenarios, highlighting the need for thorough code reviews and manual oversight. Another significant challenge is context management. The LLMs often struggle to maintain consistency across long-term projects, requiring developers to provide repeated contextual inputs [9]. Since these models are optimized for individual prompts rather than comprehensive project management, the responsibility for maintaining continuity largely falls on the development team [10].

C. Enhancing Code Quality and Security

The LLMs contribute to enhanced code quality by reducing human errors, particularly in routine and repetitive coding tasks [11]. Additionally, tools like Amazon CodeWhisperer integrate security checks into the development pipeline, enabling developers to identify vulnerabilities early in the process [12]. However, reliance on LLMs for security-critical tasks should be approached with caution, as studies have shown that certain vulnerabilities persist in LLM-generated code, underscoring the necessity of manual testing and code audits [13]. To further improve code quality, developers should adopt a human-in-the-loop (HITL) approach. This practice combines the automated capabilities of LLMs, such as code generation and documentation, with human expertise to ensure compliance with architectural standards and security protocols [14].

D. Documentation and Collaboration in LLM-Enhanced Workflows

Another significant area where LLMs add value is in the automation of documentation. These tools can generate code comments, application programming interface (API) documentation, and onboarding materials, thereby significantly reducing the time required for manual documentation efforts [4]. This functionality proves particularly beneficial in collaborative projects, where clear and consistent documentation is essential for effective communication among developers. However, ensuring consistency across various modules and documentation outputs remains a challenge [9]. The quality of LLM-generated documentation is highly dependent on the clarity of the prompts provided, as inconsistencies in instructions can lead to fragmented or incomplete outputs [15].

E. Research Gaps and Objectives

While the existing literature provides valuable insights into the integration of LLMs in software engineering, several areas remain underexplored. These include:

- Long-term project management: Investigating the scalability of LLM-based practices across extended development cycles.
- Dynamic adaptability: Understanding how LLMs can be optimized for evolving project requirements.
- Security validation: Developing frameworks to assess and mitigate the risks associated with LLM-generated code.

This study aims to address these gaps through the Kolay.ai project, exploring practical solutions to the challenges associated with LLM-based development. The primary objective is to propose strategies for effectively integrating LLMs into real-world software engineering environments.

The remainder of this paper is structured as follows: Section 2 describes the methodology employed to analyze the impact of LLMs in the Kolay.ai project, including data collection and analysis frameworks. Section 3 presents the results of the study, focusing on the effectiveness of LLMs in enhancing productivity, code quality, and collaboration. Section 4 provides a discussion on the implications of the findings, comparing them with existing literature and identifying practical strategies for overcoming identified challenges. Section 5 concludes the paper with recommendations for future research and practical guidelines for integrating LLMs into software development workflows.

II. METHODOLOGY

This section outlines the research design, data collection, and analytical techniques employed to evaluate the effectiveness of LLMs in software development, focusing on the Kolay.ai project. The methodology follows a case study approach to provide in-depth insights into how LLMs are integrated into development processes, the challenges encountered, and strategies used to address them. This approach enables a detailed exploration of LLM-powered workflows and identifies actionable practices for enhancing their performance.

A. Research Design

This study adopts a comparative case study design, focusing on a comparison between traditional software development practices and LLM-enhanced workflows. The Kolay.ai project is utilized as the primary case study, providing an in-depth examination of LLM implementation in a real-world software engineering context. The design is structured to evaluate key performance metrics, including development speed, error rates, code quality, and documentation efficiency.

The research employs both quantitative and qualitative approaches. The quantitative analysis involves measuring LLM performance in terms of development time, error reduction, and documentation output. In contrast, the qualitative analysis explores the subjective experiences of developers using LLMs, investigating their perceptions of usability, collaboration, and the challenges encountered throughout the development process.

B. Data Collection

This study employs a multi-source data collection strategy to capture both quantitative performance metrics and qualitative user experiences associated with LLM-assisted software development. The data sources collected for this study are as follows:

- Development logs and code repositories: Throughout the duration of the project, all code generated by LLMs and the subsequent revisions were systematically recorded. These development logs served as a primary quantitative data source, providing measurable indicators such as coding speed, frequency, and types of errors, and the extent of manual intervention required to correct or refine the generated code.
- Personal reflections: Qualitative data were collected through reflective journals maintained by the developers. These entries captured subjective experiences, including challenges encountered, problem-solving strategies employed, and decision-making processes during development. The reflections also provided detailed accounts of how hallucinated outputs were identified and addressed, as well as insights into the evolution of modular development practices informed by LLM integration.
- Project documentation: The documentation outputs generated by LLMs were systematically analyzed, encompassing both the initial automatically produced content and the subsequent manual revisions. This analysis aimed to assess the extent to which LLMs contribute to documentation processes in software development, with a specific focus on the accuracy, coherence, and comprehensiveness of the resulting materials.

By integrating development logs, reflective narratives, and documentation artifacts, this study constructs a comprehensive dataset that enables both quantitative evaluation and qualitative interpretation of the practical impacts associated with LLM-enhanced software engineering workflows.

C. Analytical Framework

This study employs a dual approach of quantitative and qualitative analysis to evaluate the impact of LLMs on software development. The quantitative analysis focuses on measurable metrics such as development speed and error reduction, while the qualitative aspect explores personal experiences encountered during the Kolay.ai project. As the project relied heavily on LLM-based development tools, personal reflections, and observations are instrumental in identifying key challenges, including the management of hallucinations, resolution of modular inconsistencies, and refinement of LLM-generated documentation. This mixed-methods framework ensures a comprehensive understanding of the practical benefits and limitations of using LLMs across various development phases.

1) Quantitative Metrics:

This study adopts a quantitative evaluation framework by defining several key performance indicators to facilitate a systematic comparison between traditional and LLM-assisted software development processes. The metrics employed in this analysis include the following:

- Development time: Time-tracking methods are employed to measure the duration spent on coding tasks with and without LLMs, in accordance with the framework proposed by [16]. These measures provide insights into the acceleration of prototyping and feature delivery facilitated by LLMs.
- Error reduction rate: Automated test logs and manual testing records are analyzed to determine the frequency of bugs and coding errors, aligned with methodologies discussed by [17] in case study research.
- Documentation speed and completeness: The time and effort required to produce code documentation are tracked, following guidelines from [18], who emphasize the importance of documentation as part of empirical software engineering research.
- Onboarding efficiency: While external developer onboarding was not part of the Kolay.ai project, the effectiveness of LLM-generated comments and internal documentation in maintaining project continuity is assessed, inspired by the collaborative framework described by [19].

2) Qualitative Analysis:

Since the Kolay.ai project was developed entirely by a single individual with LLM assistance, the qualitative analysis relies on personal experiences and reflections. The study follows a self-reflective methodology, inspired by [20], who advocates for reflective practice in professional development. The personal narrative method captures firsthand observations of challenges such as code hallucinations, LLM context limitations, and strategies for manual intervention. This approach provides rich, subjective insights that complement the quantitative findings. The personal reflections are documented through the following means:

- Journal entries: Regular logs maintained throughout the project track key issues, resolutions, and insights into LLM performance.
- Iterative review sessions: After each development sprint, personal notes on the effectiveness of LLM-generated outputs are reviewed to identify recurring patterns or challenges.
- HITL strategy refinement: Decisions made during the projects such as when and how to intervene manually are evaluated to fine-tune the balance between automation and human oversight, as recommended by [21].

D. Workflow Analysis of Kolay.ai Project

The Kolay.ai project followed a structured, iterative development process that integrated LLM assistance alongside human oversight to optimize efficiency and quality. The Kolay.ai project followed a structured, iterative development process that integrated LLM assistance alongside human oversight to optimize efficiency and quality. Fig. 1 illustrates this iterative modular sprint-based development model, which organizes the development workflow into modular sprints supported by LLM-generated components.

As depicted in Fig. 1, each sprint begins with prompt engineering and module specification, followed by code generation, integration, and testing. This modular approach allows developers to isolate issues, monitor quality, and refine outputs iteratively, while preventing errors in one module from affecting others. Feedback loops—such as HITL validations and post-sprint evaluations—ensure that LLM contributions are critically assessed, and prompt strategies are continuously improved. The key stages of this development process are outlined below:

- Sprint planning: Tasks were divided into modular sprints. Routine coding tasks were delegated to LLMs, while more complex tasks requiring higher-level logic and architecture were handled manually.
- LLM-assisted code generation: Tools like ChatGPT and GitHub Copilot were employed for code production, particularly for generating Create, Read, Update, Delete (CRUD) operations and API integrations. The coder actively reviewed LLM outputs to ensure they aligned with project requirements.
- Automated and manual testing: LLMs generated test cases, but the coder performed additional manual testing for critical modules to ensure quality and consistency.
- Continuous integration (CI) and documentation: Completed modules were integrated into the codebase. LLMs generated documentation and code comments, which were reviewed and refined by the coder for clarity and completeness.
- HITL review: Regular reviews were conducted to correct hallucinations and inconsistencies in LLM-generated outputs. The iterative refinement of prompts and code became a key aspect of this process.

The Kolay.ai project utilized a categorized prompt library and a modular architecture to optimize development efficiency and system scalability.

- Structured prompts: To illustrate the implementation of structured prompts, the Kolay.ai project utilized a categorized prompt library tailored to specific development tasks. For instance, CRUD operations followed a reusable prompt template such as: "Generate a Django REST API for a model named [ModelName] with the fields: [field_list]. Ensure serializers, views, and URLs are included and follow best practices." Another category of prompts addressed integration tasks, such as "Refactor this API endpoint to include token-based authentication and return HTTP 401 if the token is invalid." These prompt templates were refined iteratively across sprints and stored in a shared JSON-based library accessible during development. This approach enhanced consistency, reduced rework, and helped mitigate hallucinated outputs.

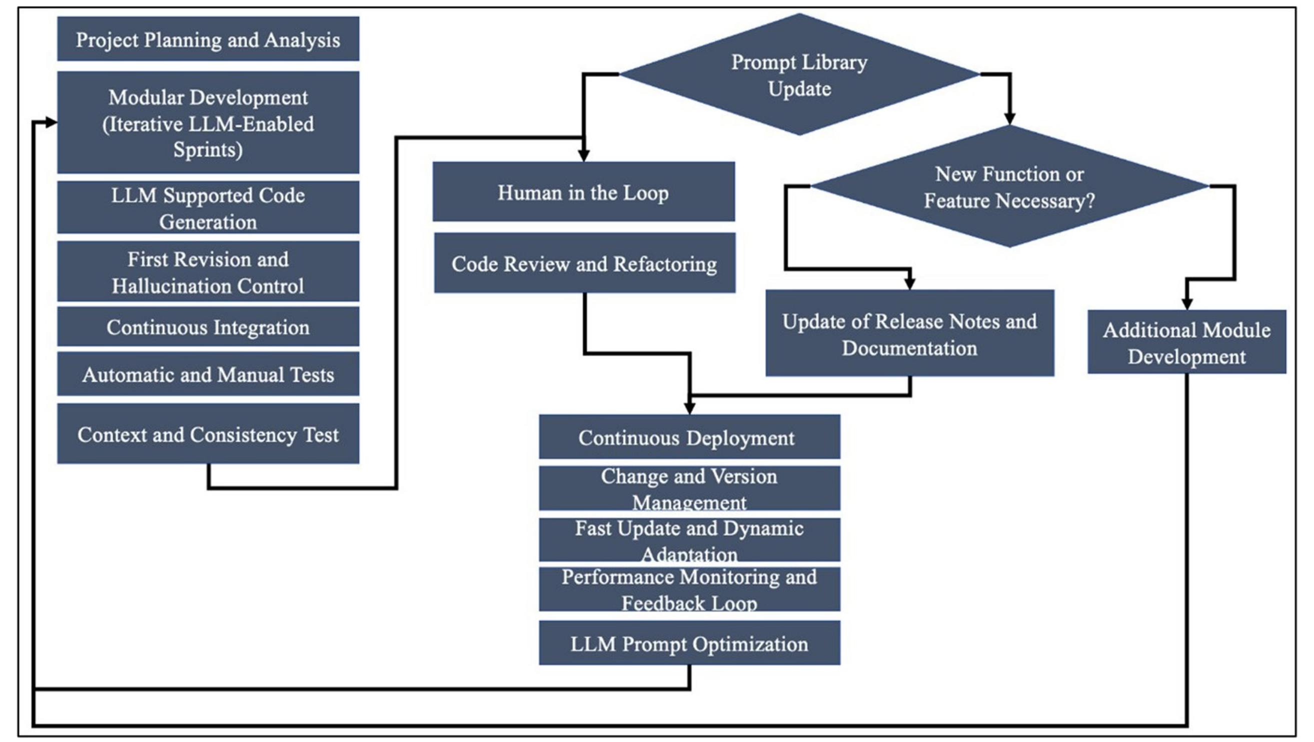
Project Planning and Analysis
Modular Development (Iterative LLM-Enabled Sprints)
LLM Supported Code Generation
First Revision and Hallucination Control
Continuous Integration
Automatic and Manual Tests
Context and Consistency Test
Prompt Library Update
Human in the Loop
Code Review and Refactoring
New Function or Feature Necessary?
Update of Release Notes and Documentation
Additional Module Development
Continuous Deployment
Change and Version Management
Fast Update and Dynamic Adaptation
Performance Monitoring and Feedback Loop
LLM Prompt Optimization

Fig. 1. Iterative modular sprint-based development model of Kolay.ai.

- Modular architecture: Kolay.ai adopted a layered modular system, where each module—such as user authentication, invoice processing, and analytics—was developed as a self-contained Django app with clearly defined interfaces. The system followed a service-oriented design: each module communicated via RESTful APIs and shared a common data model through PostgreSQL. Dependency management was handled using a central orchestration layer, which enabled version control, rollback, and hot-swapping of individual modules. This structure enabled parallel development and testing, significantly improving maintainability and scalability. Fig. 1 already depicts this sprint-based model, but an extended architectural diagram could further visualize module interactions and integration points.

E. Risk Management

This study acknowledges the inherent limitations of LLMs, such as their propensity to generate hallucinated code or lose context over time. To mitigate these challenges, several risk management strategies were systematically implemented throughout the Kolay.ai project. These strategies included the use of structured prompt libraries, automated rollback plans, and a modular development approach. Each of these strategies is further explained below:

- Structured prompt libraries: The project maintains a library of structured prompts to improve LLM outputs and reduce variability across sprints.
- Rollback plans: In case of integration issues, automated rollback mechanisms are used to restore stable versions of the code.
- Modular development approach: The modular design ensures that errors in one module do not affect the entire system.

F. Validity and Reliability

To ensure the validity and reliability of the study's findings, several methodological strategies were implemented to rigorously assess LLM performance and outcomes. These strategies include the following:

- Triangulation: Development logs, testing results, and reflective journals were used to corroborate findings, providing a comprehensive understanding of LLM performance.
- Iterative testing and review: The coder's experiences were documented and reviewed after each sprint to refine strategies and enhance the study's reliability [22].
- Expert review: Although the study is based on a single coder's experiences, external feedback on key findings was sought to validate the results and conclusions [18].

G. Ethical Considerations

This study adheres to rigorous ethical guidelines, ensuring both transparency and data security throughout its execution, thereby maintaining the integrity and credibility of the research process. These strategies include the following:

- Transparency: All development decisions, successes, and challenges were documented to provide an honest account of the LLM-based development process.
- Data security: Any sensitive project-related data was anonymized to protect intellectual property and confidentiality.

H. Limitations

This study provides valuable insights into the integration of LLMs in software development; however, several limitations must be acknowledged, which may impact the generalizability and applicability of the findings.

- Subjectivity: The study heavily relies on the coder's personal reflections, which, while providing rich qualitative insights, introduce an inherent bias. This subjectivity may influence the interpretation of LLM-generated outputs and developer experiences, potentially limiting the objectivity of the findings. As a result, the findings may not fully capture the diversity of experiences that could arise in different contexts or with different developers.
- Project-specific scope: The analysis is centered on the Kolay.ai project, which features a specific architecture and set of requirements. As such, the findings may not be entirely applicable to projects with different technological stacks, workflows, or organizational contexts. The modular architecture used in Kolay.ai, for example, may present unique challenges and opportunities that do not necessarily align with other projects, thereby limiting the broader applicability of the conclusions.
- Evolving LLM technology: The landscape of LLMs is rapidly evolving, with frequent improvements in both their capabilities and limitations. As such, some of the insights and conclusions drawn from this study may become outdated as newer versions of LLM tools are developed. This constant progression of technology suggests that while the study provides valuable insights at the time of its execution, future advancements may address some of the challenges identified, such as context maintenance and dependency management.

Despite their advantages, such as improved development speed and reduced manual workload, LLMs still face significant challenges. Hallucinated outputs, plausible but incorrect code snippets, frequently arise, especially in complex projects. Managing long-term context and maintaining dependency alignment across modules remain ongoing issues [2]. A recent survey underscores these challenges, identifying key bottlenecks in LLM integration, such as dependency management and the inability of LLMs to maintain project-wide consistency [3].

III. RESULTS AND DISCUSSION

This section presents the findings from the Kolay.ai project, where LLMs were extensively utilized for software development. The results are based on a comparison between traditional development approaches and the LLM-enhanced process. Each result is discussed with reference to the table and chart from the presentation, providing detailed insights into how LLMs affected various aspects of the project. Table I below summarizes the comparison between traditional software development and LLM-enhanced software development, highlighting key differences in development time, prototyping, debugging, documentation, and other important criteria.

In terms of Development Time, as shown in Table I, the Kolay.ai project achieved a 30–40% reduction in development time compared to traditional methods. Routine coding tasks, such as CRUD operations, were efficiently generated by LLMs, significantly accelerating the prototyping phase. This finding is consistent with studies that highlight the advantages of LLMs in expediting repetitive tasks [5]. Moreover, the modular sprint-based approach facilitated the rapid delivery of new features. As illustrated in the iterative development model chart in Fig. 1, LLMs enabled the project team to test and validate prototypes early in the development cycle. This iterative

TABLE I. COMPARISON BETWEEN TRADITIONAL SOFTWARE DEVELOPMENT AND LLM-ENHANCED SOFTWARE DEVELOPMENT

Criteria	Traditional Software Development (Pre-LLM)	LLM-Enhanced Software Development (Kolay.ai Example)
Development time	Longer due to manual coding processes	30–40% faster; routine code is generated quickly
Prototyping	Development stages are lengthy and iterative	Rapid prototyping with LLMs enables early testing
Debugging and testing	Entirely manual, with slow results	LLMs generate automated test cases, reducing errors by 30%
Documentation and code comments	Developers need extra time; documentation may be incomplete	LLMs provide automated documentation and code comments, improving onboarding speed by 20%
Code standards and consistency	Inconsistencies may arise from different developers; manual review is required	LLMs can create inconsistencies, requiring manual intervention
Context management	The entire project architecture is managed by the developer	LLMs struggle with long-term context, requiring frequent redirection
Errors in code generation	Prone to human errors, especially in repetitive tasks	Risk of hallucinations: LLMs may generate plausible but incorrect code, requiring manual review
Integration complexity	Integration between modules is manual and challenging	LLMs accelerate outputs with a modular approach, but integration issues persist
Developer productivity	Routine tasks consume time, limiting creative development	Routine tasks are automated, allowing developers to focus on creative solutions
Code quality	Depends on the experience of the developer	Despite LLM outputs, manual review and improvement are necessary
Adaptability to changing requirements	Changes take time and are costly	LLMs update code quickly, but frequent redirection is needed due to context loss
Collaboration and teamwork	Requires intensive coordination among team members	LLMs facilitate collaboration, but manual control is essential to ensure consistency

approach ensured that feedback loops were short, contributing to the continuous refinement of both the code and LLM-generated documentation.

Regarding Debugging and Testing, Table I indicates that the use of LLMs resulted in a 30% reduction in coding errors. Automated test cases were generated by LLMs, complementing the manual testing conducted during critical phases of development. While most routine tests yielded accurate results, hallucinated outputs—seemingly plausible yet incorrect code—were occasionally produced. These outputs necessitated manual intervention, as detailed in Table I. Despite these challenges, the integration of automated and manual testing proved to be effective. Automated test scenarios successfully identified common errors, while manual testing ensured that complex functionalities were implemented correctly. This finding aligns with the research conducted by [2], which underscores the importance of human oversight when utilizing LLMs in software engineering.

The LLMs also play a crucial role in automating documentation throughout the project. As presented in Table I, LLMs generated comprehensive API documentation and inline code comments following each sprint. This automation led to a 20% reduction in onboarding time for new developers, ensuring that documentation remained up to date and easily accessible. However, the presentation notes that manual reviews were sometimes necessary to rectify inconsistencies in LLM-generated documentation. Some descriptions lacked the precision required for advanced functionalities, highlighting the need for manual refinement to uphold quality standards [4].

One of the key challenges identified in Code Standards and Consistency, as presented in Table I, was maintaining long-term code consistency. Although LLMs efficiently generated modules, they occasionally struggled with contextual alignment across sprints. The presentation indicates that LLMs faced difficulties in retaining project-wide architectural decisions, necessitating frequent reorientation through structured prompts [20]. To address this challenge, a prompt library was developed, as illustrated in Fig. 1. This library functioned as a repository of structured prompts, aiding in aligning LLM-generated outputs with broader project goals. Nevertheless, maintaining contextual consistency continued to pose challenges, requiring developers to manually manage dependencies between modules.

The HITL workflow was essential for ensuring the quality of LLM-generated outputs. As highlighted in Table I, this workflow involved regular reviews and refinements to manage hallucinated code. The coder played a crucial role in evaluating LLM-generated modules, identifying errors, and ensuring that outputs aligned with the project's objectives. This iterative review process aligns with best practices in AI development, where human intervention is necessary to handle edge cases and maintain project integrity [21]. The project also implemented a rollback strategy, allowing previous versions of the code to be restored quickly in case of errors during integration.

The modular development strategy, illustrated in Table I, facilitated smooth integration of different components. The modular approach ensured that errors within one module did not affect the entire system, minimizing disruptions. The LLMs were particularly effective at generating independent modules, which were later integrated into the larger system. However, integration challenges persisted, as Table I. Some modules required additional adjustments to ensure compatibility, particularly when LLM-generated outputs differed from architectural expectations. This observation highlights the need for careful oversight during CI processes.

As detailed in the risk management section of Table I, several strategies were employed to mitigate the risks associated with LLM usage. The use of structured prompts and context files minimized the risk of hallucinated outputs. In addition, dynamic updates were implemented to handle evolving project requirements efficiently, with LLMs generating code changes on demand.

The automated rollback mechanisms implemented in the project, also outlined in Table I, effectively managed integration failures, enabling corrections to be made without significant downtime. This proactive risk management strategy aligns with the recommendations presented in [18], which emphasize the importance of maintaining software quality through continuous monitoring and timely corrections.

The findings from the Kolay.ai project illustrate both the potential and limitations of LLMs in software development. While LLMs effectively accelerated development cycles, minimized errors, and automated documentation, challenges such as hallucinated outputs, context management, and integration issues necessitate manual oversight and human intervention. These results align with existing literature emphasizing the significance of HITL workflows to maintain quality [11]. To optimize the advantages of LLMs while mitigating risks, organizations should adopt modular development strategies, structured prompt libraries, and rollback mechanisms.

Beyond empirical results, the Kolay.ai project also presents theoretical and practical contributions that expand upon current literature. These contributions are detailed in the following subsection.

A. Novelty and Contribution in the Context of Existing Literature

The Kolay.ai project introduces a novel integration of structured prompt engineering within a modular software architecture, a combination that is not extensively explored in existing literature. While prior studies have addressed aspects of prompt engineering and modular design separately, their combined application in a real-world software development context remains underrepresented.

For instance, the study [23] discusses the role of prompt engineering in collaborative AI design, emphasizing iterative prototyping and content-centric approaches. However, their focus is primarily on the design phase, without delving into the modular architectural implementation in production environments. Similarly, the study [24] proposes the MASAI framework, highlighting modular architectures for AI agents, yet they do not specifically address the integration of structured prompt engineering within such architectures.

In contrast, the Kolay.ai project demonstrates a practical application where structured prompts are systematically integrated into a modular architecture, facilitating scalable and maintainable AI-driven software development. This integration enables parallel development, efficient testing, and seamless deployment of AI modules, addressing challenges identified in previous studies.

By bridging the gap between prompt engineering and modular software design, Kolay.ai offers a replicable model for developing complex AI systems in dynamic environments. The project highlights how these two elements, when effectively combined, can improve both the quality and maintainability of AI-driven systems. This contribution adds a practical and applied dimension to the emerging body of research on HITL LLM workflows and AI-assisted software development practices.

IV. PROPOSED SOLUTIONS AND FUTURE DIRECTIONS

The findings from the Kolay.ai project reveal both the strengths and challenges of integrating LLMs into software development. While the use of LLMs accelerates development, automates routine tasks, and improves onboarding efficiency, issues like hallucinated outputs, integration complexities, and context management persist. The HITL approach is critical in managing the risks associated with LLM outputs, such as hallucinations and context inconsistencies. LLM-generated code and documentation should always undergo manual review and refinement by experienced developers to ensure correctness, security, and coherence [11]. Establishing automated checkpoints during the CI process can further enhance quality. For example, automated alerts can signal the need for human intervention when LLMs produce outputs that deviate from expected patterns or standards.

Furthermore, formalized review workflows should be integrated into agile processes. At Kolay.ai, iterative sprint reviews helped identify LLM-generated inconsistencies early, preventing them from propagating across the codebase. Future studies could explore the optimal balance between automation and manual oversight, addressing when and how developers should intervene without negating the productivity gains of LLMs [20].

LLMs rely significantly on structured prompts to generate accurate outputs. However, as seen in Kolay.ai, prompts may need continuous refinement to align LLM-generated code with project requirements. To address this, a centralized prompt library should be maintained and updated after each development sprint. This library serves as both a knowledge repository and a tool for improving LLM performance over time [9]. Furthermore, context-aware prompting techniques can enhance the effectiveness of LLMs in long-term projects. Advanced prompting strategies, such as chain-of-thought prompts or few-shot examples, can guide the LLM to maintain consistency across modules. Research on prompt engineering—the systematic crafting of prompts for optimal LLM performance—remains a crucial area for future work [4].

The modular development approach proved effective in limiting the impact of errors, as individual modules were isolated from the broader system. However, LLM-generated modules occasionally required manual alignment to ensure seamless integration. To mitigate integration risks, automated dependency checks and module validation tools should be implemented in future projects [21]. Additionally, rollback mechanisms must be a core part of the CI/CD pipeline to handle unexpected errors or failed integrations. Automated rollback capabilities, coupled with frequent code backups, can ensure that development remains uninterrupted even when issues arise during deployment [18].

LLMs often struggle with context retention over extended development cycles, leading to inconsistencies across modules. A potential solution is to implement context files—comprehensive documents containing key architectural decisions, variable definitions, and project goals. These files can be referenced during each sprint to maintain alignment between LLM outputs and project requirements [15]. Furthermore, memory-enhanced models could offer future solutions for context retention. Emerging research on contextual embeddings and fine-tuned models shows promise in enabling LLMs to remember information over longer periods [2]. Collaborative research between academia and industry can further explore how continuous learning mechanisms can be integrated into LLM workflows for sustained context awareness.

The LLMs facilitate collaboration by automating documentation and test case generation, allowing developers to focus on creative problem-solving. However, effective collaboration requires structured communication protocols to manage contributions from both developers and LLMs [13]. Developing interactive dashboards that visualize LLM-generated code, test results, and documentation updates can improve transparency and streamline team coordination. Future research should focus on LLM-powered collaborative tools that integrate seamlessly with version control systems and CI/CD pipelines. For instance, tools like GitHub Copilot already demonstrate how LLMs can assist individual developers, but more advanced tools tailored for team-based workflows could enhance productivity even further [4].

As the LLM-generated code can introduce security vulnerabilities, it is essential to integrate security checks into the development process. Future projects should employ LLM-assisted vulnerability scanners that automatically detect common coding flaws [12]. These scanners can complement traditional security audits, ensuring that both automated and manual checks are performed consistently throughout the development lifecycle.

Additionally, ethical considerations surrounding LLM usage must be addressed. Issues like bias in LLM-generated output and privacy concerns related to training data require further exploration. Establishing transparent LLM usage policies and promoting ethical practices will ensure responsible deployment of LLMs in software development [18]. The integration of LLMs into software development is an evolving field, with several promising research areas:

- Scalability of LLM usage: Future studies should explore how LLMs perform in large-scale, complex projects that involve multiple developers and long-term maintenance.
- Optimizing LLMs for agile workflows: Research on how LLMs can be further aligned with agile methodologies will ensure smoother integration into modern software development practices.
- Memory-augmented LLMs: Developing models with enhanced memory capabilities could address the challenge of context management over long projects.
- Collaborative LLM platforms: Designing platforms that allow multiple developers to interact with LLMs simultaneously could improve team-based workflows and enhance productivity.
- Continuous learning and adaptation: Future LLMs could benefit from online learning mechanisms, enabling them to adapt dynamically to changing project requirements.

Future research should explore solutions to the limitations of LLMs, such as enhancing memory-augmented models to manage context over long-term development cycles. Collaborative platforms that integrate LLMs with version control systems and CI/CD pipelines could improve team-based workflows. Recent studies suggest that LLM scalability in large-scale projects, alongside dynamic adaptability, will be crucial to the success of future software engineering initiatives [3]. Optimizing prompt engineering techniques and improving LLMs' contextual understanding will further refine these workflows.

A. Memory-Augmented Large Language Models: Concrete Research Directions and Technical Foundations

While memory-augmented language models have been proposed as a solution to context retention challenges, more concrete research directions are needed to translate this vision into implementable architectures. This section outlines actionable research questions and grounds them within the technical and theoretical foundations of recent literature. Recent work [25] introduced Retrieval-Augmented Generation (RAG) frameworks where LLMs retrieve relevant information from external token databases during inference. This suggests that external memory stores—such as vector databases trained on prior sprints' code—could be integrated into LLM-enhanced development environments. Similarly, Transformer-XL [26] proposes segment-level recurrence and relative positional encodings, allowing language models to preserve longer dependencies. For modular software systems like Kolay.ai, such mechanisms could be adapted to retain architectural decisions across iterative sprints. The following specific and researchable questions are identified:

- RQ1: How can memory-augmented architectures such as Transformer-XL or RAG be adapted to support evolving software projects, where past design choices must influence future module generation?
- RQ2: What methods can synchronize vector-based memory representations with dynamically changing codebases? Can LLMs learn to distinguish stable and volatile memory segments in code repositories?
- RQ3: Which evaluation metrics best capture contextual fidelity in memory-augmented code generation (e.g., architectural consistency, variable reuse accuracy, modular cohesion)?
- RQ4: How can reinforcement learning objectives—such as ReLoRA (Reinforcement Learning with Long-Range Attention)—be fine-tuned to maintain alignment with large-scale modular systems?

These directions not only provide a concrete roadmap for advancing the integration of memory-aware LLMs in software engineering but also highlight the need for hybrid architectures that blend symbolic and neural components. Future implementations may benefit from combining semantic code search with prompt orchestration engines and continual learning pipelines [27].

IV. CONCLUSION

This study investigated the integration of LLMs into software development through the case of Kolay.ai, offering valuable insights into both the opportunities and challenges of LLM-assisted workflows. The findings indicate that LLMs can significantly accelerate development cycles, reduce repetitive tasks, and enhance documentation quality, thereby contributing to overall efficiency. However, challenges such as hallucinated outputs, context loss, and integration complexities persist, highlighting the necessity of HITL strategies.

The modular development approach employed in the project, complemented by automated testing and a centralized prompt library, underscores the potential of LLMs to streamline development processes. Nonetheless, manual oversight remains essential, particularly in detecting and correcting hallucinated code or addressing inconsistencies across modules. The success of the Kolay.ai project illustrates that a well-balanced collaboration between human developers and LLMs can yield productive outcomes, provided that appropriate risk management mechanisms—such as rollback strategies and continuous monitoring—are established.

Looking forward, future research and development efforts should prioritize addressing the limitations of LLMs, especially in areas such as context retention, dynamic adaptability, and security assurance. The development of memory-augmented LLMs, enhanced collaborative platforms, and advanced prompt engineering techniques presents promising directions for ensuring that LLMs continue to serve as dependable assets in complex and long-term software projects.

In conclusion, while LLMs offer powerful capabilities for augmenting software development, human expertise and strategic oversight remain indispensable. By leveraging the complementary strengths of both LLMs and human developers, future initiatives can achieve higher levels of agility, creativity, and reliability. The Kolay.ai project serves as a practical illustration of the evolving role of LLMs in contemporary software engineering, demonstrating that, with thoughtful integration, these models can unlock new possibilities for innovation and productivity.

Data Availability Statement: The data that support the findings of this study are available on request from the corresponding author.

Peer-review: Externally peer-reviewed.

Author Contributions: Concept – S.E.Ş. ; Design – S.E.Ş.; Supervision – S.E.Ş.; Resources – S.E.Ş.; Materials – S.E.Ş.; Data Collection and Processing – S.E.Ş.; Analysis and Interpretation – S.E.Ş., H.N.Ö.; Literature Search – S.E.Ş., H.N.Ö.; Writing – S.E.Ş., H.N.Ö.; Critical Review – S.E.Ş., H.N.Ö.

Declaration of Interests: The authors have no conflicts of interest to declare.

Funding: The authors declared that this study has received no financial support.

REFERENCES

1. X. Hou, Y. Zhao, Y. Liu, K. Wang, and H. Wang, "Large language models for software engineering: A systematic literature review," *IEEE Trans. Softw. Eng.*, vol. 49, no. 3, pp. 1188–1201, 2023.
2. Z. Liu, Y. Tang, X. Luo, and L. F. Zhang, "No need to lift a finger anymore? Assessing the quality of code generation by ChatGPT," *IEEE Trans. Softw. Eng.*, vol. 33, no. 5, pp. 1–26, 2024.
3. S. E. Seker, "Experiences and challenges in AI-driven modular software development using large language models for code generation," 2024.
4. A. Agarwal, A. Chan, S. Chandel, J. Jang, S. Miller, and M. Tufano, "Copilot Evaluation Harness: Evaluating LLM-guided software programming," *arXiv Preprint*, 2024. arXiv:2402.14261.
5. S. Wang et al., "Machine/deep learning for software engineering: A systematic literature review," *IEEE Trans. Softw. Eng.*, vol. 49, no. 3, pp. 1188–1231, 2023. [CrossRef]
6. S. Rasnayaka, G. Wang, and R. Shariffdeen, "An empirical study on usage and perceptions of LLMs in software engineering projects," *arXiv Preprint*, 2024. arXiv:2401.16186.
7. A. Madaan, S. Zhou, U. Alon, and G. Neubig, "Language models of code are few-shot commonsense learners," *arXiv Preprint*, 2022. arXiv:2210.07128.
8. F. Tambon, A. M. Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and G. Antoniol, "Bugs in large language models generated code," *arXiv Preprint*, 2024. arXiv:2403.08937.
9D. Nam, V. Hellendoorn, B. Vasilescu, A. Macvean, and B. Myers, "Using an LLM to help with code understanding," in Proc. IEEE/ACM Int. Conf. Software Eng., Association for Computing Machinery, New York, NY, United States, 2024, pp. 1–13.
10. M. Riemer et al., "Learning to learn without forgetting by maximizing transfer and minimizing interference," *arXiv Preprint*, 2018. arXiv:1810.11910.
11. F. Liu et al., "Exploring and evaluating hallucinations in LLM-powered code generation," *arXiv Preprint*, 2024. arXiv:2404.00971.
12. R. Tóth, T. Bisztray, and L. Erdődi, "LLMs in web development: Evaluating LLM-generated PHP code vulnerabilities," in Int. Conf. Computer Safety, Reliability, and Security, Springer, Cham, 2024, pp. 425–437.
13. N. Nguyen, and S. Nadi, "An empirical evaluation of GitHub Copilot's code suggestions," in Proc. Int. Conf. Mining Software Repositories. New York, NY, USA: ACM, 2022, pp. 1–5. [CrossRef]
14. Z. Chen, and B. Liu, *Lifelong Machine Learning*. Morgan & Claypool Publishers, 2022.
15. Lomonaco, Vincenzo, Davide Maltoni, and Lorenzo Pellegrini. "Rehearsal-Free Continual Learning over Small Non-IID Batches." In CVPR workshops, Seattle, WA, USA, vol. 1, no. 2, p. 3. 2020.
16. R. Baskerville, and M. D. Myers, "Special issue on action research in information systems: Making IS research relevant to practice—Foreword," *MIS Q.*, vol. 28, no. 3, pp. 329–335, 2004. [CrossRef]
17. P. Yin, and G. Neubig, "A syntactic neural model for general-purpose code generation," *arXiv Preprint*, 2017. arXiv:1704.01696.
18. P. Runeson, M. Höst, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*. Chichester, UK: John Wiley & Sons, 2012. [CrossRef]
19. D. I. Sjøberg et al., "A survey of controlled experiments in software engineering," *IEEE Trans. Softw. Eng.*, vol. 31, no. 9, pp. 733–753, 2005. [CrossRef]
20. D. A. Schön, *The Reflective Practitioner: How Professionals Think in Action*. Basic Books, 1983.
21. A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Q.*, vol. 28, no. 1, pp. 75–105, 2004. [CrossRef]
22. R. K. Yin, *Case study research and applications: Design and methods*, 6th ed. Thousand Oaks, CA, USA: Sage Publications, 2017.
23. H. Subramonyam, D. Thakkar, A. Ku, J. Dieber, and A. Sinha, "Prototyping with prompts: Emerging approaches and challenges in generative AI design for collaborative software teams,"*arXiv*. CHI Conference on Human Factors in Computing Systems, 2025. Available: https://arxiv.org/html/2402.17721v2.
24. D. Arora et al., "Masai: Modular architecture for software-engineering AI agents". *arXiv preprint arXiv:2406.11638*, 2024. Available: https://arxiv.org/abs/2406.11638.
25. S. Borgeaud et al., "Improving language models by retrieving from trillions of tokens," *Proceedings of the 39th International Conference on Machine Learning (ICML), PMLR*, vol. 162, pp. 2206–2240. 2022. Available: https://arxiv.org/abs/2112.04426
26. Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-XL: Attentive language models beyond a fixed-length context," *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019, pp. 2978–2988. Available: https://aclanthology.org/P19-1285.
27. S. Yao, J. Zhao, and D. Zhang, "Long-term memory for language models: A survey. *arXiv preprint*, arXiv:2306.05201," 2023. Available: https://arxiv.org/abs/2306.05201.

Şadi Evren Şeker completed his undergraduate, graduate, and Ph.D. degrees in Computer Science and Engineering. Throughout his academic journey, he specialized in Natural Language Processing and Artificial Intelligence. He also conducted post-doctoral research at the University of Texas at Dallas (UT Dallas), focusing on streaming data mining and social networks. Dr. Şeker has worked at universities in six different countries, gaining experience across diverse cultures and academic environments. Currently, he serves as the Dean of the Faculty of Computer and Information Technologies at Istanbul University. He holds numerous patents and has authored several books and research papers in the fields of Machine Learning and Artificial Intelligence. His current research interests include Explainable AI, Responsible AI, AutoML, and the democratization of AI technologies for small and medium-sized enterprises (SMEs).

Hatice Nizam Özoğur completed her B.Sc., M.Sc., and Ph.D. degrees in Computer Engineering. She is currently an Assistant Professor in the Department of Artificial Intelligence and Data Engineering at Istanbul University. Her primary research interests encompass a wide range of topics within Artificial Intelligence and data-driven methodologies, including data mining, machine learning, and deep learning. Additionally, she has developed expertise in natural language processing, texture classification, fuzzy systems, and image processing techniques. Her work focuses on designing and implementing algorithms to improve the accuracy and interpretability of data classification. Through her contributions, she aims to advance computational efficiency and broaden the applicability of AI across interdisciplinary fields.