Applying Large Language Models to Software Development: Enhancing Requirements, Design and Code

Gonçalo Santos[1[0009-0007-9322-9582]], Clara Silveira[1[0000-0003-2809-4208]], Vitor Santos[2[0000-0002-4223-7079]], Arnaldo Santos[3[0000-0001-5139-6728]] and Henrique Mamede[3[0000-0002-5383-9884]]

[1] Instituto Politécnico da Guarda, Guarda, Portugal
[2] Nova IMS, Universidade Nova de Lisboa, Lisboa, Portugal
[3] Universidade Aberta, INESC TEC, Lisboa, Portugal
1700800@sal.ipg.pt,mclara@ipg.pt,vsantos@novaims.unl.pt,
arnaldo.santos@uab.pt, jose.mamede@uab.pt

Abstract. This paper explores the potential of Large Language Models (LLM) to optimize various stages of the software development lifecycle, including requirements elicitation, architecture design, diagram creation, and implementation. The study is grounded in a real-world case, where development time and result quality are compared with and without LLM assistance. This research underscores the possibility of applying prompt patterns in LLM to support and enhance software development activities, focusing on a B2C digital commerce platform centered on fashion retail, designated LUNA. The methodology adopted is Design Science, which follows a practical and iterative approach. Requirements, design suggestions, and code samples are analyzed before and after the application of language models. The results indicate substantial advantages in the development process, such as improved task efficiency, faster identification of requirement gaps, and enhanced code readability. Nevertheless, challenges were observed in interpreting complex business logic. Future work should explore the integration of LLM with domain-specific ontologies and business rule engines to improve contextual accuracy in code and model generation. Additionally, refining prompt engineering strategies and combining LLM with interactive development environments could further enhance code quality, traceability, and explainability.

Keywords: Large Language Models, Prompt Engineering, Software Development Lifecycle.

1 Introduction

As Large Language Models (LLM) continue to advance, researchers are increasingly exploring their capabilities across specialized domains, including software engineering. These models have demonstrated significant effectiveness in tasks such as code generation and identifying software vulnerabilities [1]. While LLM have demonstrated potential in basic programming tasks, their role in more complex tasks requiring semantic understanding and contextual reasoning remains less explored. This research examines how LLM can support different stages of the software development lifecycle, from requirements engineering to design and implementation. This research uses the LUNA (a previously developed B2C digital commerce platform focused on fashion retail) as a

case study. It compares outputs produced with LLM assistance to those without, focusing on quality, standards compliance, semantic accuracy, and development time. The main research problem is whether LLM can meaningfully contribute beyond code generation, especially in critical activities like requirements elicitation and architectural modeling. This work weaves into an expanding corpus body of research on the use of LLM in software engineering. Beyond development acceleration, this research investigates how LLM can maintain or improve the quality and precision of software artifacts. The research gives handy insight into the application of LLM in complex software projects, particularly addressing gaps in semantic modeling and code quality assessment.

2 Related Work

The concept of pattern is often referenced in the works of Alexander [2] who collected and documented patterns in the field of architecture. A solution pattern is a generic solution to a specific problem in a way that it can be repeatedly applied in different contexts [3]. The idea of patterns was extrapolated to software, with patterns being developed for object-oriented modeling [4],[5], for object-oriented design and programing [6], for exploring techniques, strategies, and applications [7] and for writing software requirements [8], among many other studies.

Since 2020, prompt patterns have become a vital area of research for the evolution of generative AI systems. As models grow in complexity and capability, increased attention has been directed toward how prompts can be designed, refined, and optimized. Prompt patterns are analogous to software patterns, offering structured techniques to improve interaction with LLM [9]. Indeed, as LLM like ChatGPT become an integral part of software engineering, the need for prompt engineering emerges, similar to the development of software design patterns. This transition underscores the importance of systematic approaches to enhancing the effectiveness of prompts in software development. Recent research continues to explore new strategies to enhance human–machine interaction. Studies suggest that the use of structured prompt patterns with models such as ChatGPT can lead to a 41% increase in software development efficiency compared to traditional methods [10].

The taxonomy of prompt design dimensions proposed by Braun et al. [11] offers a valuable framework for future research on how prompt construction influences user interaction with AI systems. Similarly, the AutoDev framework introduced by Tufano et al [12] presents an AI-driven structure that automates software engineering tasks, enhancing both efficiency and security. Although the study does not explicitly analyze ChatGPT prompt patterns, AutoDev's capabilities simplify complex development tasks and improve productivity in real-world software engineering applications. Empirical studies applying LLM in software engineering environments have produced important lessons. Schmidt et al [13] emphasize the need to move beyond ad hoc prompting practices, advocating for more systematic and reusable approaches. Prompt patterns serve as foundational components of prompt engineering, providing proven solutions to recurring problems and accelerating task resolution across multiple phases of the software development lifecycle. Future work in this field of research should aim to enhance the

robustness of these models, reduce hallucinations, and improve their handling of complex modeling scenarios [13].

From Kim's perspective [14], prompt-based software engineering streamlines development by automating repetitive tasks and generating high-quality outputs. Liang et al. [15] take this idea further, arguing that certain types of prompts can be considered programs in themselves. They propose that prompt development constitutes a distinct phenomenon in programming, coining the term prompt programming to describe this emerging paradigm. Patil & Gudivada [16] present a comprehensive evaluation of LLM performance in tasks such as code generation, refactoring, and debugging. Their findings underscore the strengths of tools like OpenAI's ChatGPT and GitHub Copilot, particularly in producing syntactically correct code with relatively low error rates. Nonetheless, the study highlights several limitations, including hallucinations, limited semantic depth, and challenges in maintaining context throughout extended interactions. In the realm of software modeling, Cámara et al. [17] investigate the use of ChatGPT for generating Unified Modeling Language (UML) class diagrams with Object Constraint Language (OCL). While results show that LLM are proficient in tasks with structural similarity to programming languages—such as OCL expressions, the models struggle with complex modeling constructs, including association classes and multiple inheritance. Moreover, variability in responses to identical prompts raises concerns regarding consistency and repeatability—critical factors in model-based development. The quality of LLM-assisted output is also affected by factors such as prompt formulation and model configuration. While research such as [8] emphasizes the role of hyperparameter tuning in optimizing model behavior, tools like ChatGPT do not provide access to such parameters, instead relying on prompt design to guide output behavior. In summary, LLM hold substantial promises in software engineering tasks, particularly in code-centric activities [16]. However, their application to software modeling still presents limitations in reliability, structural consistency, and semantic precision [17].

Despite LLM advances in automating software development tasks, including code synthesis, program repair and test generation [19], it appears that comparative analysis in relation to traditional development methods remains insufficiently studied.

3 Methodology

The methodology used in this work – Design Science – is a research methodology suited to the fields of Information Systems and Software Engineering for the creation of an artifact, Design Science is the design and investigation of artifacts [18], serving as the theoretical foundation supporting the scientific validity of the work. This approach was selected because it enables systematic evaluation through iterative development cycles while maintaining practical relevance through continuous evaluation. Specifically, the study seeks to answer the following research questions:

1. What is the impact of LLM on the efficiency and quality of requirements elicitation, design, and coding tasks?

2. How does LLM-generated UML compare to human-created diagrams in terms of syntax, semantics, and practical value?
3. What limitations emerge when LLM are applied to domain-specific logic and complex software modeling?

Fig. 1. Wieringa Engineering activities

Fig. 1, presents the activities underlying the Wieringa Engineering cycle [18] involved in the artifact creation process. In the problem investigation activity, the needs related to the research questions under study were identified; treatment design: interaction between artifact and context; treatment validation allows identification of changes in the artifact and/or context; treatment implementation: transfer to practice; implementation evaluation involves checking whether the designed artifacts support the initial assumptions.

The manual LUNA software development process, serving as the baseline, adopted a traditional agile Scrum methodology. It involved iterative development cycles with manual coding, debugging, and deployment of components such as a web-based management platform (using React.js for frontend and Node.js with Express for backend) and a mobile commerce application (using Flutter). Each feature was manually planned, coded, reviewed, and tested through conventional user acceptance tests and functional usability tests. This manual approach relied heavily on direct developer input for coding standards adherence and semantic accuracy, as detailed in previous internal documentation and iterative sprint retrospectives.

To systematically define qualitative aspects such as 'enhanced code readability and semantic accuracy', we implemented a structured evaluation protocol. Code readability was assessed using automated metrics (e.g., SonarQube scores) complemented by peer review checklists ensuring consistency. Semantic accuracy in UML modelling was evaluated through a defined rubric where evaluators systematically categorized diagrams according to explicit criteria, such as the specificity of model elements, clarity of relationships, and risk of semantic misunderstanding. Inter-rater reliability was ensured through calibration sessions among evaluators, mitigating subjectivity risks.

4 Comparison with the LLM-assisted approach

This section provides benchmark data on the effort and accuracy of the various elements of the LUNA software development process, allowing a direct comparison between manual work and work assisted by LLM. The manual LUNA software

development process served as a basis for comparison with the LLM-assisted approach, covering requirements gathering, architectural design, UML modeling and coding activities. These activities followed traditional practices based on expert knowledge, standard design patterns, and iterative validation through peer review. The source code was written from scratch, and testing was fully manual.

4.1 Requirements Engineering

Requirements are the base of all software products, and consequently, requirements engineering plays an important role in system development [20]. To explore the application of LLM in requirements elicitation, a tailored prompt was developed based on IEEE 830-1998 requirements specification guidelines and the Software Engineering Body of Knowledge requirements engineering practices. The prompt design incorporated key principles of effective prompt engineering [10]: Contextualization: Clearly defined project scope, including mobile app and back-office functionalities. Explicit Intent: Clear objective to identify and categorize functional and non-functional requirements. Constraints: Responses limited to relevant requirements, categorized as mandatory or optional. Specificity: Targeted questions on security, scalability, usability, and social media integration. Logical Structure: Use of numbered lists and bullet points to enhance analysis and coherence. The prompt was applied to the LUNA platform. The output, Fig. *2*, was systematically structured into three major categories: Functional Requirements include user authentication, sales processing, inventory control, and social media integration. Non-Functional Requirements: covering security, performance, scalability, and usability—ensuring compliance with industry standards and good engineering practices. Behavioral Insights: Requirements tied to social media data collection and analysis, enabling personalized marketing strategies and behavioral analytics.

The model's response not only organized these elements coherently but also validated the technical and strategic feasibility of the proposed system.

Category	Sub-Category	Examples
Functional Requirement – Mobile App	User Management	Registration, Login, Profile Editing
Functional Requirement – Mobile App	Product Catalog & Search	Browsing, Filtering, Product Details
Functional Requirement – Mobile App	Shopping Cart & Checkout	Cart Operations, Payment Methods, Order Confirmation
Functional Requirement – Commercial Platform	Inventory & Order Management	Stock Management, Order Tracking, Sales Dashboard
Functional Requirement – Commercial Platform	Social Media Integration	Social Login, Behavior Data Collection, Campaign Tracking
Non-Functional Requirement	Security	GDPR, Encryption, Secure Payments
Non-Functional Requirement	Performance	Fast Response Time, API Performance, High Availability
Non-Functional Requirement	Scalability	Auto-scaling, Cloud Infrastructure
Non-Functional Requirement	Usability	Responsive Design, Accessibility, Intuitive Navigation
Social Media Integration Requirement	Data Analytics	Sentiment Analysis, Influencer Detection
Social Media Integration Requirement	Marketing Automation	Campaign Scheduling, Performance Analytics
Social Media Integration Requirement	User Behavior Insights	Trends Tracking, Personalized Recommendations

Fig. 2. Functional and Non-Functional Requirements identified, generated by ChatGPT.

The use of LLM significantly reduced the time required for requirements gathering and structuring, **Table *1***. In a controlled study, empirical results show that the traditional approach required approximately 7 hours, while the AI-assisted method completed the task in 3 hours, representing a 42.86% reduction in time. This improvement is due to the model's ability to rapidly synthesize information, detect patterns, and provide organized outputs. Moreover, the structured prompt strategy ensured that the results were relevant and aligned with the project scope.

Table 1. Comparison of time spent on requirements elicitation using traditional methods versus the AI-assisted approach.

Approach	Time Required (hours)	Time Saved	% Improvements	Qualitative Notes
Manual (Traditional)	7	-	-	Standard elicitation with interviews
AI-Assisted	3	4h	42.86%	Structure, low ambiguity, faster synthetis

In addition to time savings, the AI-assisted process enhanced the quality of the requirement specification by reducing ambiguity and improving precision. These findings highlight the potential of LLM as effective tools for critical software engineering processes and open avenues for further research into AI-assisted requirements engineering.

4.2 Architecture

The widely used three-layer architectural model served as the foundation for the original LUNA system design. It was organized to distinguish between the data layer (PostgreSQL), the application layer (Node.js with Express), and the presentation layer, which comprised a web platform (React.js) and a mobile application (Flutter). In accordance with established practices for software engineering, this strategy sought to

guarantee modularity, scalability, and maintainability. But as AI-assisted development techniques were investigated, it became clear that including LLM might inevitably result in a unique architectural vision, especially when it came to automation, modular code creation, and more infrastructure flexibility, Fig. 3.

Layer	Technologies / Tools	Description
Presentation Layer (Web)	Next.js (React Framework)	API-first web front-end with SSR and SSG support, optimized for scalability and performance
Presentation Layer (Mobile)	Flutter	Cross-platform development framework for mobile applications
Application Layer	Node.js with NestJS	Modular backend with dependency injection, promoting clean architecture and scalability
AI Layer	OpenAI API, LangChain	Dedicated layer for prompt management, code generation, automated testing and documentation
Database Layer	PostgreSQL (Cloud RDS)	Robust relational database for structured data storage
Infrastructure	AWS Lambda, API Gateway	Serverless architecture to ensure scalability, cost-efficiency, and event-driven execution
DevOps & Quality	GitHub Actions, SonarQube	Automated CI/CD pipeline with integrated code quality analysis
Monitoring & Observability	Sentry, Prometheus	Real-time error tracking, system monitoring, and performance metrics
API Management	AWS API Gateway or Kong	API lifecycle management, security, and analytics
LLM Assistants	GitHub Copilot, ChatGPT API	AI-assisted code completion, documentation generation, testing support

Fig. 3. AI-Assisted architecture proposed for LUNA

The layered structure is maintained while major alterations are introduced in the architectural concept covered in this study, which is based on AI-driven development processes. The substitution of React.js for Next.js in the web front-end was one of the primary changes. This decision was driven by Next.js's support for server-side rendering and API routes, which complement the "API-first" methodology promoted by LLM-assisted code generation. The substitution of Express for NextJS in the application layer was another significant change. Express was adequate for the manual development stage, but NestJS provides more organized, flexible, and scalable framework—features that are especially crucial when some code is generated or optimized with AI support.

Furthermore, a specialized AI layer is incorporated into the new design, which was absent from the original version. This layer was created to centralize prompt management, automatic code scaffolding, test creation, and documentation—elements that naturally arose during the AI model experimentation—in addition to consuming LLM APIs. The specialized AI layer introduced in the LUNA architecture acts as an intermediary facilitating interaction between frontend/backend layers and external AI services. This layer includes clearly defined API endpoints for task-specific AI interactions (e.g., code generation, model evaluation), robust error handling mechanisms, and an auditing module for monitoring AI-generated outputs. Detailed architectural diagrams and implementation guidelines for this AI layer have been documented to assist other practitioners aiming to integrate similar AI capabilities into their software systems. The shift to a serverless approach based on AWS Lambda and API Gateway is a logical progression from an infrastructure standpoint, emphasizing scalability and operational simplicity, particularly in situations where development speed is crucial. Although these architectural changes seek to utilize LLM-enhanced development features,

it is recognized that every replacement like switching to Next.js or NestJS brings trade-offs regarding learning curve, maturity of the ecosystem, and complexity of integration, warranting additional empirical assessment

4.3 UML Modeling

UML modeling plays a foundational role in visualizing and validating software structure prior to implementation. Traditionally, this task is handled by experienced engineers who manually derive diagrams such as class, use case, sequence, and component diagrams from requirements. With the emergence of LLM, we explored AI-assisted diagram generation by crafting detailed prompts and using the PlantUML tool to convert the LLM-generated textual descriptions into visual diagrams. PlantUML was selected due to its open-source nature and compatibility with textual UML generation across multiple diagram types (https://plantuml.com/). Prompts were designed to describe system architecture in detail, requesting UML outputs from models such as ChatGPT. The generated text was then processed by PlantUML to produce diagrams, Fig. *4*. The process revealed several critical limitations, including syntax issues, lack of deep contextualization, structural weaknesses, and static outputs. In contrast, manual modeling demonstrated significant advantages, such as higher accuracy, better structure, and iterative refinement.

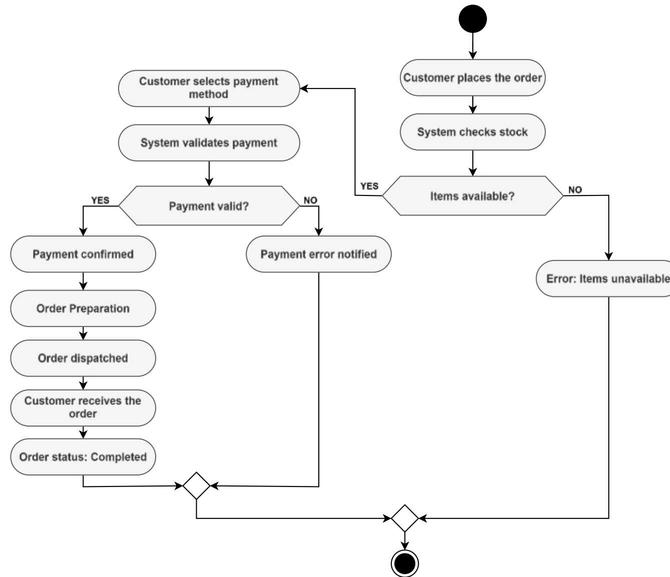

Fig. 4. Automatically generated flowchart from LLM prompt.

These findings reinforce that while LLM are useful in kickstarting UML modeling, final models still require human judgement, critical thinking, and architectural expertise to ensure robustness and accuracy. To consolidate these observations, the Table 2 presents a comparative summary between manual and AI-assisted UML modeling.

To provide a structured comparison between human-driven and AI-assisted UML modeling, this study adopted a set of quality criteria inspired by established literature on model quality assessment [22], [23], [24]. These dimensions reflect common attributes used to evaluate conceptual models in Model-Driven Engineering and software design. Table 2 summarizes the key criteria applied in this study.

Table 2. Comparative Evaluation of UML Models: Manual vs. LLM-Assisted Approaches

Criteria	Manual Modeling	LLM-Assisted Modeling
Accuracy	High – reflects system logic and domain requirements precisely	Medium – prone to semantic errors and generic representations
Syntax Reliability	Always syntactically valid	Often requires syntax correction (e.g., PlantUML errors)
Adaptability	Supports iterative refinement based on feedback	Requires full re-prompting for each adjustment
Semantic Context	Strong – interprets business logic and specific requirements	Weak – struggles with complex or domain-specific interactions
Speed (Initial Draft)	Slower for initial creation	Faster generation of basic structure
Final Usability	High – directly applicable in development workflows	Limited – requires human review and refinement

4.4 Coding

This section evaluates the quality of the source code generated by LLM, comparing it to code manually written by developers in the development of LUNA. The evaluation focuses on essential software quality characteristics aligned with the ISO/IEC 25010 standard [25], namely: maintainability, reliability, and efficiency. The objective is to analyze and compare code clarity, adherence to best practices, structural quality, and development productivity. The analysis focused particularly on the Maintainability characteristic, which includes sub-characteristics such as Analyzability, Modifiability, Modularity, and Testability. To operationalize these quality dimensions, specific metrics widely adopted in the software engineering field were selected and applied using SonarQube — a static code analysis tool. The evaluated metrics are summarized in Table 3.

Table 3. Code quality metrics selected for this study, aligned with ISO/IEC 25010 sub-characteristics.

Metric	ISO/IEC 25010 Sub-Characteristic	Purpose
Cyclomatic Complexity	Analyzability / Modifiability	Measures code complexity and potential difficulty in understanding and modifying code.
Code Smells (SonarQube)	Analyzability / Modularity	Identifies bad practices affecting code clarity and maintainability.
Bugs Detected (SonarQube)	Reliability	Detecting potential defects compromising system stability.
Modularity Score	Modularity / Reusability	Evaluates component independence and cohesion.
Maintainability Index	Maintainability (Composite Indicator)	Aggregates multiple aspects of code quality.

Documentation Coverage	Analyzability	Evaluates clarity through comments and code documentation.
Linting Compliance	Maintainability / Modifiability	Verifies adherence to coding standards (e.g., ESLint for JavaScript).

All source code, both LLM-generated and human-written, was processed using SonarQube to detect issues such as bugs, code smells, duplicated code blocks, and documentation coverage. The results are shown in Table 4, in which the values reflect the averages of three LUNA modules: User Registration, Order Processing, and Inventory Management.

Table 4. Comparative results of code quality metrics between manually written and LLM-generated code. Values reflect averages across three core modules of the LUNA.

Metric	Human Code	LLM-Generated Code
Cyclomatic Complexity (avg)	4.2	3.1
Code Smells (per 1k LOC)	3	7
Bugs Detected (Number)	0	2
Modularity Score (0-10)	8.5	7.2
Maintainability Index (0-100)	82	74
Documentation Coverage (%)	65	40
Linting Compliance (%)	96	88

The results indicate that LLM-generated code is generally syntactically correct and logically consistent, but tends to exhibit higher complexity in certain situations, increased code smells, and lower documentation coverage. Despite these limitations, LLM-generated code was considered usable after minor adjustments. Beyond code quality, development productivity was also evaluated, considering the time spent creating and refining each module. Prompt engineering involved iterative refinement of input prompts tailored for various development tasks. For example, initial prompts for code generation were overly generic, yielding suboptimal code. Through successive refinement (e.g., specifying task context, expected coding standards, and framework-specific guidelines), prompt effectiveness improved significantly. A specific instance included refining a React.js component prompt from a generic 'create a UI component' to a detailed 'create a responsive login form using Tailwind CSS with validation logic in React Hook Form', dramatically enhancing generated code quality. The times spent are detailed in Table 5.

Table 5. Time spent on manual vs LLM-assisted development across three functional modules. Post-edit time includes reviewing and refining AI-generated outputs.

Module	Human Time (min)	LLM Time (min)	Post-edit Time (min)	Net Time Saved (min)
User Registration	90	40	20	30 (33%)
Order Processing	100	42	25	33 (33%)
Inventory Management	110	45	27	38 (35%)
Average	100	42.3	24	~34 (34%)

On average, the use of LLM reduced development time by approximately 34%, even considering the time required for reviewing and correcting the generated code. This

result aligns with findings from other studies [10] reporting productivity gains between 30% and 40% when applying prompt engineering techniques to LLM-based development. In practical terms, the LLM allows developers to dedicate less time to routine coding tasks and focus more on higher-level concerns such as architecture refinement, optimization, and testing strategies. However, it became evident that the effectiveness of LLM strongly depends on the quality of prompts and the developer's ability to guide and review the model's output.

The architectural decision to transition from React.js to Next.js and from Express to NestJS was driven by enhanced server-side rendering capabilities, improved modular architecture, and ease of integration with advanced AI functionalities. However, these shifts introduced trade-offs, including increased initial complexity and the necessity for developer re-training. Potential challenges included managing the learning curve associated with NestJS's advanced decorators and modular dependency injection system, which were mitigated through targeted training sessions and comprehensive documentation.

5 Discussion of results

This section discusses the results obtained from the development of the LUNA using both manual and LLM-assisted approaches. The evaluation focused on performance, efficiency and quality at different stages of the software development lifecycle.

Results indicate that the use of LLM, particularly ChatGPT, significantly accelerated early-stage tasks such as requirements elicitation and initial design modeling. The structured application of prompt patterns enabled a 42.86% reduction in the time required for requirements analysis, while improving clarity and organizational output. These results are in the same direction as [21], where they demonstrated the effectiveness of LLM, such as GPT-4 and CodeLlama, in generating and validating software requirements specifications with a high degree of accuracy and completeness. Nonetheless, manual validation remained necessary to ensure accuracy. In addition to requirements and design, LLM contributed to faster code generation, and documentation, allowing developers to focus more on architecture refinement and user experience improvements. These findings align with Hamdi & Lim [1], who reported productivity gains of approximately 41% in AI-assisted development using prompt patterns with ChatGPT-4. Despite these positive outcomes, several challenges were identified. Maintaining context throughout complex tasks, especially during integration phases, remains difficult for LLM. Moreover, the type and structure of prompts used continue to play a critical role in output quality — a topic that warrants further research.

Preliminary code quality analysis showed that LLM-generated code is generally well-structured and readable. However, it often lacks robustness in error handling and domain-specific logic when compared to human-written code. Manual code demonstrated greater alignment with business rules and architectural constraints, despite taking longer to produce. Overall, these results reinforce the potential of LLM as valuable support tools in software development, particularly for accelerating repetitive or

structured tasks. Nevertheless, their current limitations prevent them from fully replacing human expertise in critical or context-sensitive development activities [16], [17].

Several threats to validity were identified and mitigated during the study. Participant-related bias was addressed by selecting a diverse developer group with varying experience levels with LLM-assisted development. The learning effect associated with LLM interaction was minimized by ensuring all developers underwent standardized LLM familiarization sessions. Baseline bias, resulting from the manual development approach, was mitigated by thoroughly documenting manual processes and using consistent criteria across both development scenarios to maintain fairness and comparability.

6 Conclusion

This study evaluated the practical use of LLM guided by prompt engineering across different phases of software development, using the LUNA as a real-world case study. The results indicate that LLM can significantly improve development efficiency, particularly in elicitation requirements.

Structured prompts proved essential to obtaining coherent and usable outputs, although human validation and refinement remained necessary, especially in complex tasks.

Preliminary results comparing AI-generated and manually written code suggest that LLM produce clean and syntactically correct code but struggle with complex business logic, edge cases, and contextual adaptation, areas where human developers continue to excel. The most significant challenges observed were: limited handling of edge cases and input validation; less consistent naming conventions and modularity; reduced documentation and comments. As AI adoption expands in software engineering, prompt engineering emerges as a critical practice, bridging the gap between human intent and LLM capabilities.

While LLM demonstrated potential in accelerating routine tasks, their application in system architecture and UML modeling still presents limitations. Human expertise remains essential to ensure quality, correctness, and alignment with complex business domains. In conclusion, the AI-focused design illustrates a potential future evolution of the LUNA, whereas the initial architecture adhered to traditional engineering principles and was implemented entirely by hand. It was influenced by the opportunities and difficulties found during the research as well as by the experience of utilizing LLM. It is crucial to emphasize that this history shows how AI-assisted development might impact the architectural choices of contemporary software systems rather than invalidating the original design.

Future work should explore more advanced LLM fine-tuning, domain-specific training corpora, and hybrid human-AI workflows to improve semantic accuracy, test coverage, and architectural consistency.

References

[1] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, and H. Chen, “From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future,” Aug. 05, 2024, *arXiv*: arXiv:2408.02479. doi: 10.48550/arXiv.2408.02479.

[2] C. Alexander and C. Alexander, *A Pattern Language: Towns, Buildings, Construction*. in Center for Environmental Structure Series. Oxford, New York: Oxford University Press, 1978.

[3] “About Solution-Patterns – VV-Patterns.”. Available: https://vvpatterns.ait.ac.at/about-vv-patterns/

[4] P. Coad, “Object-oriented patterns,” *Commun. ACM*, vol. 35, no. 9, pp. 152–159, Sep. 1992, doi: 10.1145/130994.131006.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[6] O. Coplien, D. Schmidt, “Pattern Languages of Program Design: 9780201607345: Amazon.com: Books.” Accessed: Apr. 12, 2025. Available: https://www.amazon.com/Pattern-Languages-Program-Design-Coplien/dp/0201607344

[7] L. Rising, Ed., *The patterns handbook: techniques, strategies, and applications*. Cambridge, U.K. ; New York : Cambridge University Press ; [New York] : SIGS Books, 1998. Accessed: Apr. 12, 2025. Available: http://archive.org/details/patternshandbook0000unse

[8] S. Adolph, A. Cockburn, and P. Bramble, *Patterns for Effective Use Cases*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] J. White *et al.*, “A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT,” Feb. 21, 2023, *arXiv*: arXiv:2302.11382. doi: 10.48550/arXiv.2302.11382.

[10] M. Hamdi and L. D. Kim, “A Prompt-Based Approach for Software Development,” *2023 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 1612–1614, Dec. 2023, doi: 10.1109/CSCI62032.2023.00267.

[11] M. Braun, M. Greve, F. Kegel, L. M. Kolbe, and P. E. Beyer, “Can (A)I Have a Word with You? A Taxonomy on the Design Dimensions of AI Prompts,” in *Proceedings of the 57th Annual Hawaii International Conference on System Sciences, HICSS 2024*, Hawaii International Conference on System Sciences (HICSS), 2024, pp. 559–568. Accessed: Apr. 12, 2025.

[12] M. Tufano, A. Agarwal, J. Jang, R. Z. Moghaddam, and N. Sundaresan, “AutoDev: Automated AI-Driven Development,” Mar. 13, 2024, *arXiv*: arXiv:2403.08299. doi: 10.48550/arXiv.2403.08299.

[13] D. C. Schmidt, J. Spencer-Smith, Q. Fu, and J. White, “Towards a Catalog of Prompt Patterns to Enhance the Discipline of Prompt Engineering,” *Ada Lett.*, vol. 43, no. 2, pp. 43–51, Jun. 2024, doi: 10.1145/3672359.3672364.

[14] D.-K. Kim, “Prompted Software Engineering in the Era of AI Models,” Sep. 07, 2023, *arXiv*: arXiv:2311.03359. doi: 10.48550/arXiv.2311.03359.

[15] J. T. Liang, M. Lin, N. Rao, and B. A. Myers, “Prompts Are Programs Too! Understanding How Developers Build Software Containing Prompts,” 2024, doi: 10.48550/ARXIV.2409.12447.

[16] R. Patil and V. Gudivada, “A Review of Current Trends, Techniques, and Challenges in Large Language Models (LLMs),” *Applied Sciences*, vol. 14, no. 5, Art. no. 5, Jan. 2024, doi: 10.3390/app14052074.

[17] J. Cámara, J. Troya, L. Burgueño, and A. Vallecillo, “On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML,” *Softw. Syst. Model.*, vol. 22, no. 3, pp. 781–793, May 2023, doi: 10.1007/s10270-023-01105-5.

[18] R. J. Wieringa, “The Design Cycle,” in *Design Science Methodology for Information Systems and Software Engineering*, R. J. Wieringa, Ed., Berlin, Heidelberg: Springer, 2014, pp. 27–34. doi: 10.1007/978-3-662-43839-8_3.

[19] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, “Agentless: Demystifying LLM-based Software Engineering Agents,” Oct. 29, 2024, *arXiv*: arXiv:2407.01489. doi: 10.48550/arXiv.2407.01489.

[20] E.-M. Schön, J. Thomaschewski, and M. J. Escalona, “Agile Requirements Engineering: A systematic literature review,” *Computer Standards & Interfaces*, vol. 49, pp. 79–91, Jan. 2017, doi: 10.1016/j.csi.2016.08.011.

[21] A. Hemmat, M. Sharbaf, S. Kolahdouz-Rahimi, K. Lano, and S. Y. Tehrani, “Research directions for using LLM in software requirement engineering: a systematic review,” *Front. Comput. Sci.*, vol. 7, Mar. 2025, doi: 10.3389/fcomp.2025.1519437.

[22] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 2nd edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2005.

[23] J. Krogstie, G. Sindre, and H. Jørgensen, “Process models representing knowledge for action: a revised quality framework,” *Eur J Inf Syst*, vol. 15, no. 1, pp. 91–102, Feb. 2006, doi: 10.1057/palgrave.ejis.3000598.

[24] O. I. Lindland, G. Sindre, and A. Solvberg, “Understanding quality in conceptual modeling,” *IEEE Software*, vol. 11, no. 2, pp. 42–49, Mar. 1994, doi: 10.1109/52.268955.

[25] ISO/IEC 25010:2023, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, International Organization for Standardization, 2023.