Mrežna inačica: ISSN 2459-6302
Tiskana inačica: ISSN 1849-1995

DOI: 10.19279/TVZ.PD.2025-13-4-14

# EVALUACIJA VELIKIH JEZIČNIH MODELA U PROFESIONALNOM OKRUŽENJU: IMPLEMENTACIJA, DOKUMENTACIJA, REFAKTORIRANJE I ANALIZA SOFTVERSKOG KODA

# *EVALUATION OF LARGE LANGUAGE MODELS IN A PROFESSIONAL ENVIRONMENT: IMPLEMENTATION, DOCUMENTATION, REFACTORING AND CODE ANALYSIS*

**Mario Seider, Dominik Forjan, Renato Belošević, Ivan Cesar**

*Zagreb University of Applied Sciences, Vrbik 8, 10000 Zagreb, Croatia*

## SAŽETAK

U ovom radu provedena je usporedna analiza velikih jezičnih modela koji se danas koriste u profesionalnom radnom okruženju za potporu svakodnevnim zadacima znanja. Istraživanje se usredotočuje na praktičnu uporabu modela u scenarijima kao što su izrada tehničke dokumentacije, podrška razvoju softvera, obrada prirodnog jezika te pomoć pri donošenju odluka. Analiza je provedena kroz definirane radne zadatke koji odražavaju stvarne uvjete rada, pri čemu su modeli vrednovani prema kriterijima točnosti odgovora, konzistentnosti, razumljivosti generiranog sadržaja i prilagodljivosti kontekstu zadatka. Rezultati istraživanja pokazuju da se veliki jezični modeli značajno razlikuju u načinu obrade složenih upita i u kvaliteti odgovora u profesionalnim scenarijima. Analiza ukazuje da određeni modeli pokazuju izraženu prednost u većini promatranih kategorija, osobito u zadacima koji zahtijevaju napredno razumijevanje konteksta i složeniju tehničku obradu. Dobiveni rezultati naglašavaju važnost svjesnog i kontekstno prilagođenog odabira modela, ovisno o vrsti radnog zadatka i zahtjevima profesionalnog radnog okruženja. Rad doprinosi boljem razumijevanju stvarne primjenjivosti velikih jezičnih modela u profesionalnoj praksi.

***Ključne riječi:*** *veliki jezični modeli, profesionalno radno okruženje, umjetna inteligencija, analiza primjene, obrada prirodnog jezika*

## *ABSTRACT*

This paper presents a comparative analysis of large language models used in professional working environments to support everyday knowledge-based tasks. The research focuses on practical usage scenarios, including the creation of technical documentation, software development support, natural language processing, and assistance in decision-making processes. The evaluation is conducted through a set of defined work tasks that reflect real working conditions, with models assessed according to response accuracy, consistency, clarity of generated content, and adaptability to task context. The results demonstrate noticeable differences among the analyzed models in handling complex professional queries and in the quality of generated outputs, indicating that performance strongly depends on the nature of the task. The analysis shows that certain models exhibit a clear advantage across most evaluation criteria, particularly in scenarios requiring advanced contextual understanding and technical reasoning. These findings emphasize the importance of selecting an appropriate model based on specific professional requirements and work scenarios. This study contributes to a clearer understanding of the practical applicability and limitations of large language models in everyday professional use.

***Keywords:*** *large language models, professional working environment, artificial intelligence, application analysis, natural language processing*

## 1. UVOD
## *1. INTRODUCTION*

Artificial intelligence has become one of the most influential technological fields of the twenty-first century, significantly transforming the way information is processed, analysed, and utilized across various domains [1]. From early rule-based expert systems to contemporary data-driven approaches, artificial intelligence has continuously evolved in response to increasing computational power and the availability of large-scale data [2]. Today, artificial intelligence systems are widely applied in areas such as finance, healthcare, education, software engineering, and decision support, where they assist professionals in performing complex cognitive tasks more efficiently and accurately [3,4].

A breakthrough in artificial intelligence research occurred with the rapid development of machine learning methods, particularly those based on neural networks [5]. These approaches enabled systems to learn patterns directly from data rather than relying on explicitly programmed rules [5,6]. Over time, neural network architectures became deeper and more complex, leading to substantial improvements in tasks such as image recognition, speech processing, and natural language understanding [6,7]. However, the growing complexity of these models also introduced new challenges related to interpretability, reliability, and practical usability in professional environments.

Within the broader field of artificial intelligence, natural language processing has emerged as a particularly impactful area, as it enables machines to interact with humans using natural language [8]. Early natural language processing systems relied on symbolic methods and handcrafted linguistic rules, which limited their scalability and adaptability [9]. The introduction of statistical methods and later deep learning approaches significantly improved the ability of systems to model linguistic structures and semantic relationships. This evolution eventually led to the development of large language models, which can generate coherent, context-aware, and semantically rich text based on vast amounts of training data.

Large language models represent a class of artificial intelligence systems designed to process and generate human language at a scale and level of sophistication previously unattainable [10]. Trained on extensive textual corpora, these models learn probabilistic representations of language that allow them to perform a wide range of tasks, including text generation, summarization, translation, question answering, and code assistance [10]. Unlike earlier natural language processing solutions that were tailored to specific tasks, large language models exhibit a high degree of generality, enabling their application across diverse professional contexts.

In recent years, large language models have transitioned from research laboratories into everyday professional use. They are increasingly integrated into workflows involving software development, technical documentation, customer support, data analysis, and knowledge management. In professional working environments, such models are often employed as interactive assistants that support users in drafting content, analysing requirements, exploring solutions, and accelerating routine tasks [11]. Their ability to understand context and generate relevant responses has positioned them as valuable tools for enhancing productivity and supporting decision-making processes.

The range of domains where large language models provide practical value extends well beyond software engineering. In healthcare, these models assist with the summarization of medical records, generation of clinical reports, and support of patient communication, helping practitioners manage the growing administrative burden of modern medicine. In the legal domain, they are applied to contract review, legal research, and the automated drafting of standardized documents, reducing the time professionals spend on repetitive, text-intensive tasks. Educational institutions and e-learning platforms have begun leveraging large language models as adaptive tutoring tools capable of generating personalized explanations and formulating exercises tailored to individual learners. In business contexts, applications span automated report generation, knowledge base maintenance, and analysis of customer feedback at scale. Across these varied domains, a common theme emerges: large

language models act as force multipliers, enabling professionals to focus on higher-order reasoning while delegating routine language-intensive work to the model.

Despite their growing adoption, large language models differ significantly in terms of architecture, training strategies, and practical behaviour. These differences directly affect their performance in professional scenarios, particularly when handling complex, domain-specific, or ambiguous tasks [12, 13]. While some models excel in generating fluent and well-structured text, others demonstrate strengths in logical reasoning, technical accuracy, or contextual consistency. As a result, the choice of an appropriate model has become an important consideration for professionals who rely on artificial intelligence systems in their daily work [14].

## 2. POSTAV I KRITERIJI EVALUACIJE

## *2. SETUP AND EVALUATION CRITERIA*

The setup of this study is designed to evaluate the practical usability of large language models in a professional software development environment. The evaluation focuses on everyday development tasks implemented in the C# programming language, reflecting common responsibilities encountered by software engineers in real-world projects. All experiments were conducted using a code editor environment integrated with an artificial intelligence assistant, ensuring consistent interaction conditions across all evaluated models.

Although many contemporary models perform exceptionally on standardized benchmarks, this performance does not always translate to real-world development, which was a main motivation behind the research In practice, ambiguity, legacy constraints and domain-specific context often expose limitations not visible in curated evaluations [15].

The selected models represent widely used large language models currently available for professional use, including GPT-4o, GPT-5.1, Claude Sonnet 4.0 and Claude Sonnet 4.5 [16, 17, 18]. Each model was accessed through the same integrated development workflow to minimize external variability. The evaluation was performed using identical task descriptions and prompts for all models to ensure fairness and reproducibility. Context was reset between independent test runs to avoid cross-task information leakage.

The first task involved the implementation of a basic create, read, update, and delete system. This task tested the model's ability to generate functional, compilable code while adhering to established architectural patterns. Emphasis was placed on separation of concerns, including controllers, handlers, repositories, and endpoints. Code was evaluated for compilation success, runtime correctness, and adherence to clean code principles.

Generating functional source code is a particularly demanding task for large language models, as software development requires strict syntactic correctness and consistency across multiple components. Even minor errors in variable definitions or method signatures may cause compilation failures, and maintaining logical coherence across interconnected layers — controllers, services, repositories, and data transfer objects — adds further complexity [20,21].

Research has consistently revealed a significant gap between syntactic plausibility and true functional correctness. Chen and colleagues introduced the HumanEval benchmark and showed that even Codex, a dedicated code-specialised model, correctly solved only approximately 28.8% of tasks on the first attempt [20]. Liu and colleagues further showed that benchmark performance commonly overstates real-world reliability once test suites are extended with broader input coverage [22]. In multi-component architectures, models additionally tend to generate incorrect or nonexistent method signatures and interface definitions, causing cascading errors across dependent layers. Security is a further concern: Pearce and colleagues found that approximately 40% of model-generated code snippets contained at least one vulnerability, a problem explored in depth by Chong and colleagues [19,22].

The second task focused on technical documentation writing. Models were required

to generate documentation describing the implemented system, its components, and its intended usage. This task assessed the ability of the models to maintain consistency between code and documentation, use appropriate professional terminology, and present information in a structured and comprehensible manner.

The third task addressed refactoring and optimization of existing code. Models received a working codebase and were instructed to improve readability, structure, or performance without altering functional behaviour. This task evaluated the models' understanding of the existing implementation and their ability to identify improvement opportunities while following best practices.

The fourth task involved analysis and explanation of a given code segment. Models were required to interpret existing C# code and provide a clear technical explanation of its purpose, logic, and structure, emphasizing contextual understanding and accurate communication of technical concepts.

Evaluation was conducted using four predefined criteria: correctness and technical validity, context understanding, response consistency, and clarity and code structure. Correctness required that generated code compiled successfully and functioned as intended. Context understanding assessed how accurately the model interpreted task requirements, including the presence of hallucinated information. Response consistency was measured across three independent iterations of each task using identical prompts and reset context. Clarity and code structure evaluated adherence to clean code practices and logical system decomposition.

These criteria were selected because together they capture functional correctness, contextual reasoning, reliability, and maintainability, which represent the most relevant dimensions of model performance in everyday development tasks. Other potential criteria, such as execution performance or security analysis, were not included because they require more complex benchmarking environments and fall outside the scope of this study.

Each task contributed a predefined number of points to a total maximum score of 100 points per model. This scoring framework enabled quantitative comparison while also supporting qualitative analysis of observed strengths and limitations.

The distribution of points across tasks reflects the relative complexity and practical importance of the evaluated activities. The

***Tablica 1*** *Raspodjela bodovanja prema zadatku i kriterijima ocjenjivanja*

***Table 1*** *Scoring distribution by task and evaluation criteria*

| Task / Criterion | Correctness and Technical Validity | Context Understanding | Response Consistency | Clarity and Code Structure | Total |
|---|---|---|---|---|---|
| Task 1: Problem Implementation (CRUD) | 15 | 7 | 7 | 6 | 35 |
| Task 2: Documentation Writing | 5 | 5 | 5 | 5 | 20 |
| Task 3: Refactoring and Optimization | 10 | 5 | 5 | 5 | 25 |
| Task 4: Code Analysis and Explanation | 5 | 5 | 5 | 5 | 20 |
| Total | 35 | 25 | 20 | 20 | 100 |

CRUD implementation task was assigned the highest weight (35 points) because it represents a complete development scenario involving architectural structuring, functional correctness, and integration of multiple components. The refactoring and optimization task received 25 points as it requires deeper understanding of an existing codebase and the ability to improve its structure without altering functionality.

The documentation writing and code analysis tasks were each assigned 20 points. These tasks evaluate the models' ability to interpret and communicate technical information, which is important for development workflows but less directly related to system functionality than implementation tasks.

Within each task, points were distributed across the evaluation criteria to ensure that multiple aspects of model performance contribute to the final score. However, correctness was given slightly higher weight in implementation-oriented tasks, reflecting its central importance for functional software generation.

## 3. POSTUPAK PROVOĐENJA I BODOVANJA VELIKIH JEZIČNIH MODELA

## *3. PROCEDURE FOR TESTING AND SCORING LARGE LANGUAGE MODELS*

The procedure for testing and scoring large language models was defined to reflect realistic professional software development workflows while maintaining consistency across all evaluated systems. Each model was evaluated using identical prompts (see 6. Appendix) and under the same execution conditions to ensure that observed differences originated from model behaviour rather than external factors. The evaluation process was structured around four distinct tasks, each targeting a different aspect of professional software engineering practice, ranging from implementation and documentation to refactoring and deep technical reasoning. The exact prompts used for each task are provided in the Appendix.

The first part of the evaluation focused on the implementation of a Weather controller that exposes a set of create, read, update, and delete endpoints, extended with batch update and batch delete operations. The goal of this task was not limited to functional correctness but emphasized architectural quality and maintainability. The implementation was expected to follow real-world design practices, including a clear separation between controller, service, and repository layers. Additionally, explicit request and response Data Contracts were required for each endpoint, alongside dedicated model definitions. Code readability and logical organization were considered essential, as the resulting solution was evaluated as if it were part of a production-ready application.

The implemented API exposed both collection-level and single-resource endpoints. Collection operations included retrieving all forecasts (`GET /api/WeatherForecast`) and creating a new forecast (`POST /api/WeatherForecast`). Operations on a specific forecast used a route parameter (`/api/WeatherForecast/{id}`), where id represents the unique identifier of the resource and enables retrieval, update, and deletion of a single entity. In addition, batch update and batch delete endpoints (`PUT /api/WeatherForecast/batch and DELETE /api/WeatherForecast/batch`) were provided, allowing multiple resources to be modified or removed within a single request.

Following implementation, the evaluation shifted toward documentation quality. In this phase, the models were tasked with producing technical documentation on the existing large scale codebase[1] responsible for fraud detection within the Splitit payment platform on two existing ASP.NET MVC API controllers: *DetectController* and *InternalAnalysisController*, taken from the *Splitit.FraudDetection.Api* exposing REST API endpoints for real-time fraud scoring and internal risk analysis, and integrating with downstream services for transaction evaluation and decision routing. The documentation was expected to cover all available endpoints and to provide sufficient

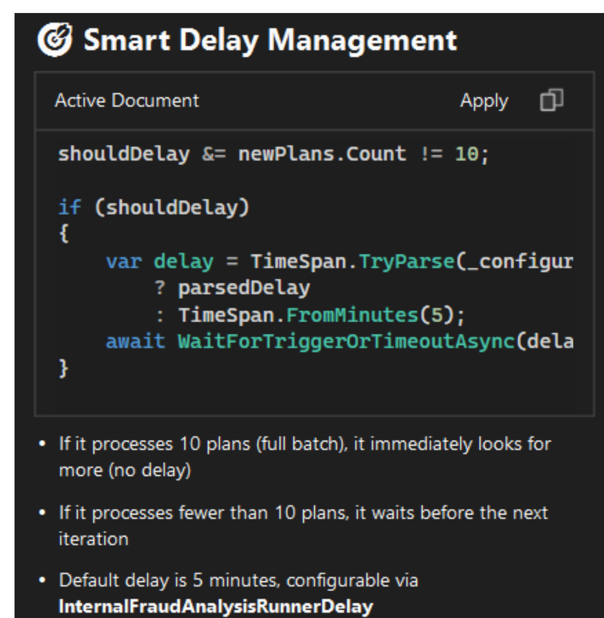
1 Splitit.FraudDetection.Api is a proprietary internal .NET service used for fraud detection within the Splitit payment platform. It is not publicly available and was used in this study with permission.

technical detail to support understanding by engineers unfamiliar with the codebase. This included descriptions of request and response structures, explanation of execution flow, and references to related services and components involved in request handling. More detailed and context-aware documentation was considered indicative of stronger professional applicability.

Attention was then directed to refactoring and optimization test followed, where the models were required to improve the structure of previously mentioned *DetectController*, in which business logic was originally concentrated within the controller itself. The objective of this task was to enhance readability and maintainability by extracting logic into dedicated services and organizing complex behaviour into smaller, well-defined functions. A critical constraint of this phase was the preservation of existing behaviour; the refactored code was required to function identically to the original implementation. This requirement ensured that the evaluation measured the ability to perform safe, production-oriented refactoring rather than superficial restructuring.

The final task addressed deep code analysis and explanation. Rather than focusing on endpoint-level logic, this task required the models to analyse *InternalFraudAnalysisRunner*, a processor component responsible for coordinating fraud analysis operations. High-quality explanations were defined by the ability to go beyond a superficial description of the class itself and instead capture its role within the broader system. This included identifying and explaining interactions with other services, dependencies, and execution flow across the surrounding components. The task therefore evaluated the depth of technical understanding and the ability to reason about interconnected system behaviour.

All responses were graded manually by four evaluators from Splitit: Senior software architect Cesar, experienced software engineer Seider, QA automation engineer Belošević, and AI developer Forjan. Each evaluator reviewed the outputs independently and from the angle most relevant to their background and role in Splitit: Cesar and Seider looked at things like architecture and whether the code was actually structured the way a developer would aim, Belošević focused on correctness and whether the output held up under less obvious conditions, and Forjan paid particular attention to how the models behaved across repeated runs and how sensitive they were to the way prompts were worded. Scores were given across four criteria: correctness and technical validity, context understanding, response consistency, and clarity and code structure; using a sliding scale rather than a simple pass/fail approach, since most responses fell somewhere in between. The final score for each response was the average of the four individual scores. In cae of bigger gaps between scores, the evaluators talked it through to make sure the difference came from a genuine disagreement about quality rather than individual interpreting the criteria differently.




***Slika 1*** *Objašnjenje generirano modelom Claude Sonnet 4.5*

***Figure 1*** *Explanation generated by Claude Sonnet 4.5 model*

Correctness was the most straightforward criterion to apply, though even here judgment was required. Generated code was compiled and executed where possible, and evaluators verified that the output satisfied the stated requirements: correct endpoints, proper layer separation, preserved behaviour after refactoring. However, code that compiled but silently violated architectural constraints, such as a service layer that leaked database concerns into the controller, was penalized just as code that failed to build entirely.

A technically functional solution that would be rejected in a real code review was not considered fully correct.

Context understanding proved to be the most revealing criterion. Professional development rarely comes with a complete specification, and the prompts used in this study were intentionally written the way a developer would phrase a request to a colleague: conversational, with implicit expectations. Evaluators assessed whether each model could read between the lines: did it recognize that a production-ready API should include error handling even when not explicitly mentioned? Did it apply naming conventions consistent with the surrounding codebase? Did it treat the prompt as a starting point or as a complete instruction set? Models that treated every unstated requirement as out of scope scored poorly here, while those that exercised professional judgment scored well.

Consistency captured something that only becomes apparent when reading a full response rather than isolated snippets. A model might generate a clean controller and a well-structured service, but if the interface defined in one does not match the method signatures used in the other, the output as a whole is unusable. Evaluators studied each response end to end, checking that component boundaries were respected, that variable names and conventions did not shift arbitrarily mid-response, and that the logic described in documentation matched the logic present in the accompanying code.

Clarity and structure were graded with the assumption that the output would be handed directly to another developer. Code was evaluated for readability: meaningful naming, appropriate decomposition into functions, absence of unnecessary complexity. Documentation was judged on whether it actually communicated something or merely restated the code in prose. Analytical explanations were assessed on whether they built understanding progressively or overwhelmed the reader with detail without direction. Throughout the evaluation, borderline cases were discussed between evaluators before a final score was agreed upon, ensuring that individual bias did not unduly influence the results.

## 4. USPOREDNI REZULTATI UČINKOVITOSTI VELIKIH JEZIČNIH MODELA

## *4. COMPARATIVE RESULTS OF LARGE LANGUAGE MODEL PERFORMANCE*

The evaluation revealed notable differences in performance among the tested large language models when applied to professional C# development tasks. GPT-4o achieved the lowest overall score, with a total of 51 points. In our testing, this model proved adequate for simpler tasks but struggled as task complexity increased. While it was able to generate functional code for basic scenarios, it exhibited significant difficulties during refactoring tasks, where its changes occasionally introduced unintended side effects. These issues indicate that GPT-4o requires close supervision in professional environments, as unmonitored use may lead to destructive modifications within an existing codebase. Its performance suggests limited suitability for complex or detail-intensive development tasks. This model was intentionally chosen as part of the experiment to further illustrate the rapid pace of advances in generative AI – only recently considered as state-of-the-art model, only few months later provides rather sub-par performance compared to more recent models.

GPT-5.1 demonstrated a slightly improved overall performance, achieving a total score of 58 points. One of its most prominent characteristics was a high level of response consistency across iterations. The model tended to adopt a conservative strategy, prioritizing compilation success and functional safety over architectural quality. In the implementation task, GPT-5.1 frequently centralized logic within controller components, effectively ignoring structural separation into handlers or repositories. While this approach reduced the occurrence of compiler errors, it significantly weakened code clarity and maintainability. Additionally, the model initially resisted generating technical documentation, requiring prompt reformulation to produce usable output. This behaviour limited its achievable score despite its otherwise stable performance. GPT-5.1

can therefore be considered reliable but overly restrictive, favouring safe solutions at the expense of professional architectural standards.

In contrast, the Claude Sonnet models emerged as the strongest performers in the evaluation. Claude Sonnet 4.0 achieved a total score of 88 points and consistently produced high-quality results across all tasks. The model demonstrated strong contextual understanding and was able to generate well-structured, maintainable code with clear separation of concerns. Refactoring tasks were handled particularly well, with improvements that preserved functionality while enhancing readability and structure. Documentation and code analysis outputs were detailed, accurate, and closely aligned with the implemented solutions.

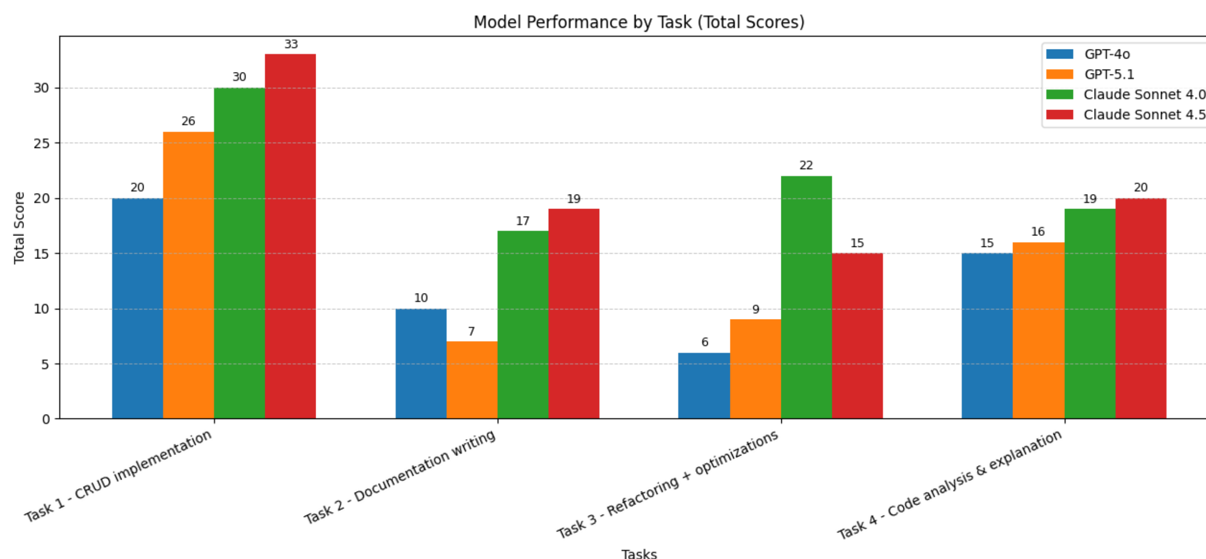
Overall, Claude Sonnet 4.0 exhibited a balanced combination of technical correctness, contextual awareness, and professional code organization.

Claude Sonnet 4.5 achieved a comparable overall score of 87 points and delivered similarly strong results in most evaluated tasks without significant observable improvement over its predecessor. Its outputs were detailed and technically sound, particularly in implementation, documentation, and analytical tasks. However, during the refactoring task, the model showed a tendency to diverge from the intended scope, proposing changes that extended beyond the original requirements. This behaviour suggests a sensitivity to prompt specificity, as the model appeared to interpret refactoring objectives more

***Tablica 2*** *Tablični prikaz rezultata modela prema pojedinim zadacima*

***Table 2*** *Tabular representation of model results by individual tasks*

| Task 1 | Correctness (15) | Context (7) | Consistency (7) | Clarity and Structure (6) | Total (35) |
|---|---|---|---|---|---|
| GPT-4o | 9 | 5 | 3 | 3 | 20 |
| GPT-5.1 | 14 | 4 | 7 | 1 | 26 |
| Claude Sonnet 4.0 | 11 | 7 | 6 | 6 | 30 |
| Claude Sonnet 4.5 | 14 | 7 | 6 | 6 | 33 |
| | | | | | |
| Task 2 | Correctness (5) | Context (5) | Consistency (5) | Clarity and Structure (5) | Total (20) |
| GPT-4o | 2 | 1 | 3 | 4 | 10 |
| GPT-5.1 | 1 | 1 | 2 | 3 | 7 |
| Claude Sonnet 4.0 | 4 | 5 | 3 | 5 | 17 |
| Claude Sonnet 4.5 | 5 | 5 | 4 | 5 | 19 |
| | | | | | |
| Task 3 | Correctness (10) | Context (5) | Consistency (5) | Clarity and Structure (5) | Total (25) |
| GPT-4o | 1 | 2 | 2 | 1 | 6 |
| GPT-5.1 | 1 | 2 | 4 | 2 | 9 |
| Claude Sonnet 4.0 | 8 | 5 | 4 | 5 | 22 |
| Claude Sonnet 4.5 | 6 | 1 | 3 | 5 | 15 |
| | | | | | |
| Task 4 | Correctness (5) | Context (5) | Consistency (5) | Clarity and Structure (5) | Total (20) |
| GPT-4o | 4 | 1 | 5 | 5 | 15 |
| GPT-5.1 | 4 | 2 | 5 | 5 | 16 |
| Claude Sonnet 4.0 | 5 | 4 | 5 | 5 | 19 |
| Claude Sonnet 4.5 | 5 | 5 | 5 | 5 | 20 |




***Slika 2*** *Grafički prikaz rezultata modela prema pojedinim zadacima*

***Figure 2*** *Graphical representation of model results by individual tasks*

broadly than intended. With more constrained task descriptions, its performance in this area could likely be improved. Despite this limitation, Claude Sonnet 4.5 remains highly effective for professional software development tasks, demonstrating capabilities comparable to the strongest model evaluated.

## 5. ZAKLJUČAK
## *5. CONCLUSION*

In this paper, we conducted a comparative evaluation of large language models in the context of professional software development tasks using the C# programming language. The results demonstrate clear differences in the ability of the evaluated models to handle complex technical requirements, maintain contextual understanding, and apply architectural best practices in real-world development scenarios.

The analysis shows that conservative model behaviour provides stability but often limits solution quality, while models with stronger contextual reasoning deliver more maintainable and structured outcomes. Refactoring tasks proved particularly sensitive, as incorrect decisions in this phase may introduce unintended and potentially harmful changes to existing systems.

Overall, large language models can offer valuable support in professional software development, but their effectiveness strongly depends on task complexity, clarity of requirements, and active human supervision. Future research may further explore their application across different programming environments and extended development workflows.

## 6. DODATAK
## *6. APPENDIX*

Task 1

Prompt:

"*Hey, can you build me a WeatherForecastController in C#? I need proper layered architecture — so controller, service, and repository should all be separate. The API needs the usual CRUD endpoints plus batch update and batch delete. Make sure each endpoint has its own request/response contracts and model classes. Keep it clean and production-ready.*"

Task 2

Prompt:

"*Can you write technical documentation for DetectController and InternalAnalysisController from the Splitit.FraudDetection.Api project? For each endpoint I'd like to know what it expects as input, what it returns, and how the request flows*

*through the system — what services get called, what depends on what. Assume the reader hasn't seen this code before.*"

Task 3

Prompt:

"*This DetectController is a bit of a mess — all the business logic is crammed inside the controller itself. Can you clean it up? Move the logic into proper services, break down anything complex into smaller functions, and make it easier to read. One important thing though — don't change what it actually does, it needs to behave exactly the same as before.*"

Task 4

Prompt:

"*Can you take a look at this InternalFraudAnalysisRunner class and explain what's going on? I don't just want a surface-level description — I want to understand its role in the bigger picture: what other services it talks to, what depends on, and how the whole flow works around it.*"

## 7. REFERENCE

## *7. REFERENCES*

[1.] Rashid, A. B., & Kausik, M. K. (2024). AI revolutionizing industries worldwide: A comprehensive overview of its diverse applications. Hybrid Intelligence Advances, 100277. DOI: 10.1016/j.hybadv.2024.100277 https://doi.org/10.1016/j.hybadv.2024.100277

[2.] Russell, S. J., & Norvig, P. (2020). Artificial intelligence: A modern approach (4th ed.). Pearson. ISBN: 978-0134610993. http://lib.ysu.am/disciplines_bk/efdd4d1d4c2087fe1cbe03d9ced67f34.pdf

[3.] Weng, Y., Wu, J., Kelly, T., & Johnson, W. (2024). Comprehensive overview of artificial intelligence applications in modern industries. arXiv. DOI: 10.48550/arXiv.2409.13059. https://arxiv.org/abs/2409.13059

[4.] Hamdan, A., Hassanien, A. E., Khamis, R., Alareeni, B., Razzaque, A., & Awwad, B. (Eds.). (2021). Applications of Artificial Intelligence in Business, Education and Healthcare. Springer. DOI: 10.1007/978-3-030-72080-3. https://doi.org/10.1007/978-3-030-72080-3

[5.] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT Press. ISBN: 978-0-262-33737-3 https://www.deeplearningbook.org

[6.] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning (in Nature). DOI: 10.1038/nature14539. https://doi.org/10.1038/nature14539

[7.] Khan, S. H., & Iqbal, R. (2025). A comprehensive survey on architectural advances in deep CNNs: Challenges, applications, and emerging research directions. arXiv. DOI: 10.48550/arXiv.2503.16546. https://arxiv.org/abs/2503.16546

[8.] Jurafsky, D., & Martin, J. H. (2009). Speech and Language Processing (2nd ed.). Prentice Hall. ISBN: 978-0-13-187321-6. https://www.researchgate.net/publication/220355425_Speech_and_Language_Processing_second_edition_Daniel_Jurafsky_and_James_H_Martin_Stanford_University_and_University_of_Colorado_at_Boulder_Pearson_Prentice_Hall_2009_xxxi988_pp_hardbound_ISBN_978-0-13

[9.] Jegan, R., & Henrich, A. (2025). A structured literature review on traditional approaches in current natural language processing. arXiv. DOI: 10.48550/arXiv.2505.12970. https://arxiv.org/abs/2505.12970

[10.] Mienye, I. D., Jere, N., Obaido, G., Ogunruku, O. O., Esenogho, E., & Modisane, C. (2025). Large language models: An overview of foundational architectures, recent trends, and a new taxonomy. Discover Applied Sciences, 7, 1027. DOI: 10.1007/s42452-025-07668-w. https://doi.org/10.1007/s42452-025-07668-w

[11.] Davenport, T. H., & Miller, S. (2022). Working with AI: Real stories of human-machine collaboration. MIT Press. ISBN: 978-0-262-04716-6. https://www.researchgate.net/

publication/361693960_Working_with_AI_Real_Stories_of_Human-Machine_Collaboration_Davenport_T_H_Miller_S_M_2022_MIT_Press

[12.] Deligiannidis, L., Dimitoglou, G., & Arabnia, H. (2024). Artificial intelligence: Machine learning, convolutional neural networks and large language models. Walter de Gruyter GmbH & Co KG. DOI: 10.1515/9783111344126. https://doi.org/10.1515/9783111344126

[13.] Moradi, M., Yan, K., Colwell, D., Samwald, M., & Asgari, R. (2025). A critical review of methods and challenges in large language models. Computers, Materials & Continua. DOI: 10.32604/cmc.2025.061263. Available at: https://doi.org/10.32604/cmc.2025.061263

[14.] Atkinson, J., & Abutridy, J. (2024). Large language models: Concepts, techniques and applications. CRC/Taylor & Francis. DOI: 10.1201/9781003475576. https://doi.org/10.1201/9781003475576

[15.] Banerjee, S., Agarwal, A., & Singh, E. (2024). The vulnerability of language model benchmarks: Do they accurately reflect true LLM performance? (Preprint). DOI: 10.48550/arXiv.2412.03597. https://arxiv.org/abs/2412.03597

[16.] Anthropic. Claude Sonnet model overview. Available at: https://www.anthropic.com/claude/sonnet

[17.] OpenAI. (2023). GPT-4 Technical Report. Available at: https://arxiv.org/abs/2303.08774

[18.] OpenAI. GPT-5.1: A smarter, more conversational ChatGPT. Available at: https://openai.com/index/gpt-5-1/

[19.] Chun Jie Chong, Zhihao Yao, Iulian Neamtiu (2024). Artificial-Intelligence Generated Code Considered Harmful: A Road Map for Secure and High-Quality Code Generation DOI: 10.48550/arXiv.2409.19182. Available at: https://arxiv.org/abs/2409.19182

[20.] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., ... & Zaremba, W. (2021). Evaluating Large Language Models Trained on Code. DOI: 10.48550/arXiv.2107.03374. Available at: https://arxiv.org/abs/2107.03374

[21.] A. Patel, K. Z. Sultana, and B. K. Samanthula, A Comparative Analysis between AI Generated Code and Human Written Code: A Preliminary Study, Proc. IEEE Int. Conf. on Big Data (BigData), 2024. DOI:10.1109/BigData62323.2024.10825958 Available at: https://ieeexplore.ieee.org/document/10825958

[22.] Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2024). Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In Advances in Neural Information Processing Systems (NeurIPS 2023). DOI: 10.48550/arXiv.2305.01210. https://arxiv.org/abs/2305.01210

## AUTORI · *AUTHORS*

• **Mario Seider** - vanjski suradnik na Tehničkom veleučilištu u Zagrebu, na kolegiju Programiranje, prijediplomski studij Računarstvo.

• **Dominik Forjan** - vanjski suradnik na Tehničkom veleučilištu u Zagrebu, na kolegiju Baze podataka, prijediplomski studij Računarstvo.

• **Renato Belošević** - vanjski suradnik na Tehničkom veleučilištu u Zagrebu, na kolegiju Baze podataka, prijediplomski studij Računarstvo.

• **Ivan Cesar** - viši predavač na Tehničkom veleučilištu u Zagrebu; nositelj kolegija Programiranje i Razvoj web aplikacija u ASP.NET MVC tehnologiji, prijediplomski studij Računarstvo.