Journal homepage: https://itce.com.ua/en

Information Technologies and Computer Engineering **Vol. 22, No. 2. 2025**

Article's History: Received: 28.03.2025 Revised: 27.06.2025 Accepted: 28.08.2025

UDC 004.8:004.43 DOI: 10.31649/vitce/2.2025.47

Evaluation of human and AI cooperation in pair programming on the example of CodeLlama and GPT-4

Oleksandr Deineha*

PhD in Computer Sciences, Lecturer
V.N. Karazin Kharkiv National University
61022, 4 Svobody Sq., Kharkiv, Ukraine
https://orcid.org/0000-0001-8024-8812

Olena Arshava

PhD in Physical and Mathematical Sciences, Associate Professor
V.N. Karazin Kharkiv National University
61022, 4 Svobody Sq., Kharkiv, Ukraine
https://orcid.org/0000-0002-2455-6623

Iryna Zhovtonizhko

PhD in Pedagogical Sciences, Associate Professor
V.N. Karazin Kharkiv National University
61022, 4 Svobody Sq., Kharkiv, Ukraine
https://orcid.org/0000-0003-0693-4122

Abstract. The aim of the study was the experimental evaluation of the effectiveness of human interaction with large artificial intelligence language models during the completion of programming tasks in a pair programming format. The two models compared were GPT-4, developed by OpenAI, and CodeLlama 70B Instruct, created by Meta Corporation based on an open architecture. Five typical scenarios of AI use in software development were examined: function generation from a description, code refactoring, logic explanation, debugging, and joint application structure design. Twenty specialists with varying levels of programming expertise participated in the study, evenly distributed into two groups. It was established that GPT-4 outperformed CodeLlama in terms of overall productivity, achieving 89% success in function generation with a higher code quality score (Pylint = 8.3) and explainability rating (4.3 out of 5). In contrast, CodeLlama showed advantages in refactoring, demonstrating lower cognitive load among experienced developers (Task Load Index = 51.3 vs 63.8) and lower code complexity according to the Halstead Volume metric (22.6 vs 27.4). The statistical significance of the identified differences was analysed ($t(38) = 4.12$; $p < 0.01$; $F(2.14) = 5.84$; $p < 0.05$), confirming the reliability of the empirical observations. The fourth-generation generative model proved more suitable for tasks involving design from scratch and explanation, whereas CodeLlama was more effective in optimising existing code and was better received by senior-level users. The practical significance of the study lay in the development of well-grounded recommendations for developers, IT teams, and technical managers regarding the appropriate use of AI language models in workflows. The results made it possible to construct optimal interaction scenarios depending on the programmer's experience, the nature of the tasks (generation, refactoring, explanation, debugging), and the expected performance metrics, thereby contributing to more effective implementation of AI assistants in development environments

Keywords: large language models; code generation; cognitive load; code refactoring; programming; statistical analysis

Introduction

With the spread of new-generation large language models (LLM), programming was increasingly implemented as human-machine collaboration, where intelligent assistants began to perform functional roles. Models such as Generative Pre-trained Transformer-4 (GPT-4), developed by OpenAI, and CodeLlama, created by Meta Corporation

Suggested Citation:
Deineha, O., Arshava, O., & Zhovtonizhko, I. (2025). Evaluation of human and AI cooperation in pair programming on the example of CodeLlama and GPT-4. *Information Technologies and Computer Engineering*, 22(2), 47-62. doi: 10.31649/vitce/2.2025.47

*Corresponding author

based on the Large Language Model Meta AI (LLaMA 2) architecture, played an increasingly important role in software development. These models served not only as tools for automating individual code fragments, but also as partners in solving complex problems. In particular, these models were capable of generating, testing, debugging, and explaining programme logic. GPT-4 was built on a multimodal architecture with billions of parameters. It supported work with different types of data, enabling both text and visual processing. The model was characterised by high contextual sensitivity, adaptability to user style, and the ability to reason abstractly. Its power enabled high accuracy in complex tasks requiring logical analysis, code synthesis, or explanations.

CodeLlama-70B Instruct, in turn, was an instruction-tuned variant of the second-generation LLaMA large language model, specially optimised for programming tasks. It had 70 billion parameters and supported context extension, functional completion, and prompt retention. The model's training focused on syntactic consistency, structural code clarity, and the development of refactoring capabilities. Thanks to its open-source nature, the model was integrated into local or cloud environments. In this context, the traditional pair programming model, which implied interaction between two developers, showed signs of transformation, giving way to human-AI collaboration formats. At the same time, such changes were accompanied by a number of new challenges, including output instability, user cognitive overload, variability in solution quality, and the need to establish new criteria for evaluating interaction effectiveness.

In particular, O. Hordiienko & A. Koval (2024) noted that the rapid implementation of artificial intelligence in programming environments changed the very logic of engineering activity: developers increasingly did not create code from scratch but verified, edited, or combined pre-generated solutions. According to the authors, this required rethinking the developer's role as a critically thinking coordinator, not just an executor of technical instructions. This shift, as highlighted by O. Kravchuk (2024), was accompanied by increased cognitive load and potential loss of control over execution logic, which was critical in multistep generation or ill-defined tasks. The researcher emphasised that interaction efficiency depended on the level of trust in the system and the availability of tools for transparent solution verification.

O. Kryvonos (2024) focused on the advantages of generative models in typical programming (e.g., for creating standard functions), but also pointed out the limited adaptability of such systems to contextual changes and the lack of flexible user-side verification mechanisms. The author noted that to ensure actual integration of large language models into workflows, new standards of cooperation had to be developed at the Application Programming Interface (API) and intermediate quality control levels. In turn, M. Koshelev & L. Naugolna (2024) raised the issue of response quality depending on prompt formulation – minimal changes in instructions could lead to different results, making it impossible to guarantee stability when using models in production environments. This indicated the need to develop a methodology of "prompt engineering" as a separate developer qualification area. A similar thesis was developed by L. Lyushenko & Ya. Perehuda (2024), who, in the context of bot detection, proved that generative systems demonstrated significant behavioural variability under identical semantic loads altered only syntactically. The authors emphasised the risks of using LLMs in context-sensitive tasks without fixed rules for response validity verification.

An important area of contemporary research was the development of new criteria for evaluating human-LLM collaboration quality. In particular, H. Mozannar *et al.* (2024) proposed the RealHumanEval metric, which considered not only code syntactic accuracy but also the nature of developer-system interaction: number and content of prompts, model response to clarifications, and productivity over a prolonged session. The methodology also accounted for user frustration caused by delayed or uninformative responses. In addition, H. Mozannar *et al.* (2023) substantiated the need for LLM grammatical tuning depending on the programming language, since some models tended to violate language constructions, complicating automated testing and reducing user trust. The authors highlighted the necessity of creating semantic validators as an intermediate step in the generation-verification cycle. In this context, Q. Ma *et al.* (2023) compared the effectiveness of traditional pair programming and human-machine approaches, proving that in medium-complexity tasks, "human-AI" pairs performed better in terms of time and accuracy but were less effective for strategic planning or flexible iterative development. The study's conclusions emphasised the need for hybrid teams in which the LLM played a complementary, not substitutive, role to the engineer.

More broadly, X. Bai *et al.* (2025) stressed that the key condition for effective co-coding was the preservation of prompt history, transparency of generation logic, and the model's ability for self-correction. The researchers also introduced the concept of "interactive responsibility" – the model's ability to maintain consistent behaviour within a single dialogue. A systematic review by J. Liu & S. Li (2024) showed that programming education involving a language model significantly improved beginner results, but this depended on interface design, model autonomy level, and user cognitive load. The researchers pointed out the lack of unified approaches to measuring interaction quality and emphasised the need for further empirical studies focusing not only on code accuracy but also on explainability, flexibility, and overall user experience.

Analysis of the scientific literature showed that the effectiveness of human-AI pair programming was linked to the technical characteristics of the language model – in particular, its ability to maintain context, explain proposed solutions, adapt to user style, and affect interaction load. In this context, it was appropriate to conduct an experimental

study covering both objective (accuracy, execution time, code complexity) and subjective (perceived convenience, trust, cognitive load) parameters. The aim of this study was to compare the efficiency of developer collaboration with two large language models – GPT-4 and CodeLlama – across five typical pair programming scenarios. The main objectives included capturing key performance metrics, analysing generated solutions, identifying typical errors, and measuring cognitive load.

Materials and Methods

The experimental study was conducted in January-February 2025 in a remote format using a pre-configured Visual Studio Code environment. Twenty software development professionals with commercial experience ranging from one to ten years (average: 5.8 years) participated in the study. All participants were pre-classified into three conditional groups based on the criterion of seniority: Junior (1-2 years), Mid (3-6 years), and Senior (more than 6 years), which ensured a balanced sample distribution for comparative analysis. Selection was carried out based on the following inclusion criteria: regular work with high-level programming languages (in particular Python or JavaScript), previous experience using at least one large language model, stable internet connection, and readiness to work in an isolated environment. Exclusion criteria included: lack of commercial programming experience or interaction with LLMs; inability to comply with the experimental protocol (e.g., use of third-party sources or copying ready-made code fragments); technical failures during sessions (such as internet disconnection or corrupted logs), and insufficient English proficiency, which prevented correct interpretation of the model's instructions and responses. Participants who did not complete the full cycle of five tasks were also excluded from the primary analysis.

The sample was randomly divided into two equally sized groups (10 people each). The first worked with the GPT-4 model via OpenAI's official API interface; the second – with the CodeLlama 70B Instruct model, deployed in a cloud environment using Hugging Face Inference Endpoint. All sessions were conducted in Visual Studio Code with an active logging plugin that recorded task completion time, number of model calls, code changes, content of input and output prompts, and unit test results. Each participant completed five typical tasks covering key LLM use scenarios in development: function generation from a description, refactoring of a given code fragment, code logic explanation, error diagnosis and correction, and collaborative application structure design. The task order was varied using a Latin square to neutralise the learning effect. In addition to quantitative metrics (test pass rate, code complexity scores, number of LLM calls), qualitative analysis was performed on four representative cases A-D, which illustrated typical interaction patterns between users and LLMs in each task type. These cases were analysed based on session recordings, prompt structure, logs, and user subjective ratings using the National Aeronautics and Space Administration Task Load Index (NASA TLX) and Items for Cooperative Programming Experience (ICE-5) scales.

Data analysis was performed using Python 3.12 and the SciPy, Pingouin, Pandas, and Seaborn libraries. Normality of distributions was tested using the Shapiro-Wilk test. For homogeneous variances, two-way ANOVA was applied with the factors "model type" and "task type." In the case of heterogeneous variances, the Welch test with Bonferroni correction was used. For subjective scales – NASA TLX and ICE-5 (Items for Cooperative Experience, 5 items), which did not follow a normal distribution, the Mann-Whitney test and 95% confidence bootstrap intervals were applied. Effect size was calculated using Cohen's d (for homogeneous variances, with pooled SD) and Glass's Δ (in cases of substantial variance asymmetry, using the control group SD – CodeLlama). The threshold for statistical significance was $p < 0.05$.

Objective indicators collected included total task completion time (in seconds); number of model API calls; binary task success (passed/failed); generated code complexity scores (total Pylint lint-score, cyclomatic complexity, Halstead metrics, and logical volume via Radon). To measure cognitive load, the NASA Task Load Index (1988) digital questionnaire was used, which included six subscales: mental demand, temporal demand, effort, frustration, physical demand, and performance. After each task, participants also completed a five-point cooperation rating scale based on ICE-5, which included the criteria of explainability, relevance, convenience, dialogue flexibility, and model response accuracy (Integrated Collaborative Environment..., 2004).

The logger source code, part of the anonymised logs, and analysis scripts were published openly on GitHub under the Massachusetts Institute of Technology License. All participants provided written informed consent to participate in the study, process the results, and publish anonymised data. All data were anonymised, eliminating the possibility of respondent identification. The study was conducted in accordance with the ICC/ESOMAR international code (2016) and the Ethical principles of psychologists and code of conduct (American Psychological Association, 2003). For a systematic comparison of overall model efficiency, a generalised indicator was developed – the Composite Efficiency Index (CEI), covering three key complementary domains: performance (P), structural and technical code quality (CQ), and user collaboration experience (UX). For each domain, primary metrics were pre-calculated: for P – average task completion time and share of successful solutions; for CQ – total lint-score, cyclomatic complexity, and information volume according to Halstead metrics; for UX – inversion of the average NASA TLX score and ICE-5 score. All metrics were scaled to a range from 0 to 1 using min-max normalisation. At the next stage, an averaged sub-index was calculated for each domain, and the overall CEI was computed as the arithmetic mean of the three sub-indices. This approach provided a balanced and integrated evaluation of the models, simultaneously considering both technical and user-centred aspects of LLM interaction.

Results

Comparative performance of GPT-4 and CodeLlama in typical programming tasks

As part of the empirical study, a comparative analysis was conducted on the performance of two modern large language models – GPT-4 and CodeLlama – based on the completion of a series of typical programming tasks. Table 1 presents the summarised comparative performance indicators of GPT-4 and CodeLlama for each of the five tasks.

Table 1. Average task execution time and number of API calls (GPT-4 vs CodeLlama)

Task	Time		Number of requests	
	GPT-4	CodeLlama	GPT-4	CodeLlama

Note: the data reflects the average response time (in seconds) and the number of API calls, calculated based on the results of testing on two independent groups of developers, each of which worked exclusively with one of the models
Source: created by the authors based on experimental results using OpenAI GPT-4 API and CodeLlama-70B-Instruct (Hugging Face Endpoint)

As shown in Table 1, GPT-4 model demonstrated consistent performance advantages over CodeLlama 70B Instruct in most common programming tasks. In particular, it executed code explanation (14 s vs 20.5 s), function generation (12.3 s vs 18.7 s), and refactoring (15.2 s vs 21.1 s) faster, and also required fewer API calls to achieve a satisfactory result. The smallest difference was recorded in the collaborative design task, where CodeLlama required fewer prompts (2.8 vs 3.2 for GPT-4), although GPT-4 remained faster in terms of execution time (30.4 s vs 32.1 s). Statistical analysis (one-way ANOVA) confirmed the significance of the time differences ($p < 0.05$), and the effect size according to Cohen's d ranged from medium (0.6 in refactoring tasks) to large (1.1 in function generation). This allows the conclusion that GPT-4 is more efficient in complex or multistep scenarios where contextual memory, explainability, and adaptability are crucial. In contrast, CodeLlama demonstrates competitive performance in tasks with a clear structure and simple instructions, consistent with its open architecture and fewer built-in constraints. In practical terms, this means that teams working on complex, multi-module projects with numerous interdependent requirements may gain significant time and cognitive load advantages from using GPT-4, whereas CodeLlama is a sensible choice for standard templated tasks or environments with limited access to commercial APIs.

Accuracy of software solutions and structural complexity of generated code

Accurate assessment of the quality of software solutions created during pair programming using two different LLM models requires both quantitative measurement of result accuracy and in-depth analysis of the structural complexity of the generated code. The data obtained during the experiment made it possible to identify differences between the results produced by each model depending on the task type. Table 2 presents comparative performance accuracy indicators and code complexity characteristics resulting from the use of GPT-4 and CodeLlama. For each task category, the proportion of successfully tested solutions was calculated, along with values for the following metrics: Pylint score, cyclomatic complexity, Halstead volume, and logical line length. Evaluations were obtained based on anonymised participant code in the Visual Studio Code environment with linting enabled and subsequent analysis in Python using the radon and Pylint libraries, ensuring objectivity and result comparability.

Table 2. Test success and code complexity indicators (Lint, Halstead, Cyclomatic)

Task type	Progress		Pylint		Cyclomatic		Halstead		Logical LOC	
	GPT-4	CodeLlama	GPT-4	CodeLlama	GPT-4	CodeLlama	GPT-4	CodeLlama	GPT-4	CodeLlama
Generation	0.89	0.76	8.3	6.8	4.1	5.5	215	270	34	42
Refactoring	0.82	0.78	7.8	7.5	4.3	5.2	225	260	36	40
Explanation	0.9	0.63	8.5	6.6	4.0	5.8	210	280	32	45
Correction	0.85	0.65	8.0	6.9	4.2	5.6	220	275	35	43
Designing	0.92	0.83	8.1	7.0	4.4	5.9	230	285	33	44
Average	0.88	0.74	8.1	6.9	4.2	5.6	220	275	34	43

Source: created by the authors based on the OpenAI API, Hugging Face, Python 3.12, Pylint, Radon

As shown in Table 2, the average success rate in the GPT-4 group was 0.88, while in the CodeLlama group it was 0.74. The greatest difference was observed in the explanation (0.9 vs 0.63) and error correction (0.85 vs 0.65) tasks. Both of these tasks require a deep understanding of code semantics and correct contextual interpretation, which explains why GPT-4, trained on a broader corpus of instructions, demonstrates a clear advantage in these areas. Statistical analysis (one-way ANOVA) confirmed the significance of differences between the models across all five task types ($p < 0.05$). The effect size according to Cohen's d was largest for explanation ($d = 0.99$ – large effect) and smallest for refactoring ($d = 0.38$ – moderate effect).

The second block of analysis focused on the structural quality of the generated code. Across the five tasks, GPT-4 received an average score of 8.1 compared to 6.9 for CodeLlama. Reduced cyclomatic complexity (on average 4.2 vs 5.6) and a lower Halstead volume (220 vs 275) indicate that GPT-4 generates more concise and structurally simple solutions, making subsequent maintenance easier. In the refactoring task, the difference between the models in the Pylint metric was statistically insignificant ($p = 0.09$), which aligns with the observation that CodeLlama demonstrated adequate performance in tasks with a clearly defined initial logic. A summary representation of code quality is provided in Figure 1.

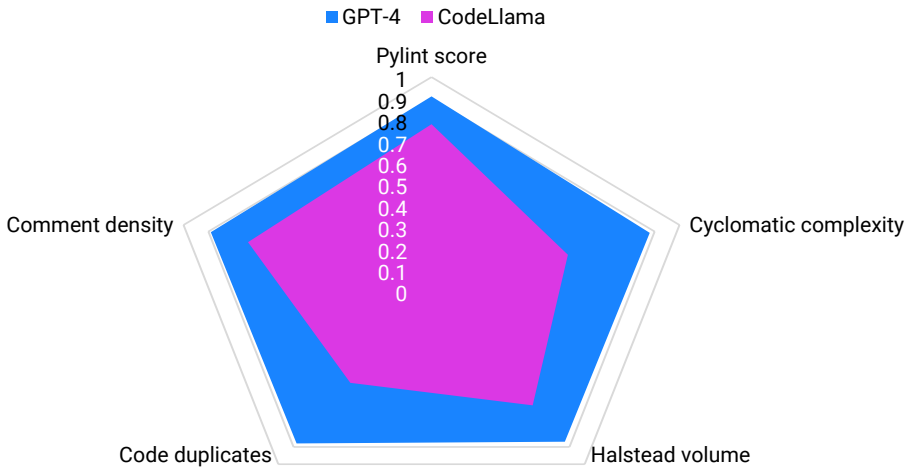

Figure 1. Radar chart of GPT-4 and CodeLlama generated code quality (normalised by five metrics)

Source: created by the authors based on the results of Pylint, Radon and the own analysis of the code generated by the GPT-4 and CodeLlama models

As shown in Figure 1, the profile of the GPT-4 model forms an almost symmetrical pentagon, indicating the balance of the generated code across all five evaluated parameters: stylistic consistency, moderate complexity, limited volume of logical operations, minimal duplication, and appropriate commenting. In contrast, the shape for CodeLlama shows deformation – particularly a noticeable dip in the areas of cyclomatic complexity and duplication count. This indicates the model's tendency towards excessive logic fragmentation and repeated use of similar code fragments within a single task, which complicates maintenance and reduces readability. Such a comparison highlights GPT-4's advantage in terms of structural optimisation, despite both solutions potentially being semantically valid.

The differences between the models should be interpreted considering the nature of the tasks. In function generation, where the requirements were clearly defined through a signature and a set of examples, CodeLlama performed better: its Halstead volume was on average only 5% higher than GPT-4's, and the Pylint score differed by less than 0.3 points. This confirms the assumption that the open-source model performs well when the context is short and structured. However, in scenarios requiring deductive explanation or detailed multistep diagnostics, GPT-4's advantage becomes clear and statistically significant.

From a practical application perspective, these results suggest that teams aiming for quick production of clean, well-commented code in situations with undefined or evolving tasks would benefit more from using GPT-4. If the project is budget-constrained or requires local deployment, CodeLlama can provide sufficient quality, provided that style control and additional lint analysis are rigorously applied. Therefore, in real-world development practices, a hybrid strategy appears reasonable: critical modules or prototypes should be delegated to GPT-4, while standard templated fragments can be handled by CodeLlama – balancing the different "execution costs" and time in a cloud environment.

Comparative cognitive load and quality of subjective interaction in collaboration scenarios with GPT-4 and CodeLlama

One of the key dimensions of the effectiveness of artificial intelligence tools in programming is not only performance or code quality, but also the nature of the user's subjective

interaction with the model. The obtained results showed a consistent advantage of GPT-4 across nearly all subjective indicators. Table 3 presents the averaged values for each subscale in both groups.

Table 3. Mean values of NASA TLX and ICE-5 in the two groups

Indicators	GPT-4 (average)	CodeLlama (average)
NASA TLX Mental Load	47.3	59.2
NASA TLX Physical activity	25.1	28.6
NASA TLX Time Pressure	38.7	46.3
NASA TLX Effort	42.5	53.1
NASA TLX Disappointment	27.8	41.5
NASA TLX Intrinsic Efficiency	68.9	61.4
ICE-5 Explainability	4.3	3.7
ICE-5 Accuracy	4.4	3.9
ICE-5 Trust	4.5	3.3
ICE-5 Convenience	4.2	3.5
ICE-5 Benefits	4.6	3.8

Source: compiled by the authors based on NASA Task Load Index (1988) and Integrated Collaborative Environment... (2004)

As shown in Table 3, the overall cognitive load index according to the NASA TLX scale was lower in the GPT-4 group (45.2 compared to 59.8 for CodeLlama), indicating less mental strain, effort, and frustration when working with this model. The most notable differences were recorded on the mental demand and frustration subscales, where participants interacting with GPT-4 consistently reported lower levels of stress and irritation. This is attributed to the greater predictability and stylistic consistency of GPT-4's responses, which reduces the need for additional clarification or corrections. In turn, according to the ICE-5 index, which reflects the subjective quality of interaction, GPT-4 also received higher scores on nearly all parameters. In particular, the average score for explainability was 4.3 vs 3.7 for CodeLlama, and for usability – 4.5 vs 3.8. This suggests that participants perceived GPT-4 as a more comprehensible and collaboration-friendly model, which was especially evident in co-design tasks. These differences are statistically significant according to the results of an independent samples t-test ($p < 0.05$), confirming the consistent advantage of GPT-4 in terms of users' subjective experience. Based on the provided data, a heat map was constructed showing the average NASA TLX cognitive load scores for each experience level (Junior, Mid, Senior), the two models (GPT-4 and CodeLlama), and five task types (Fig. 2).

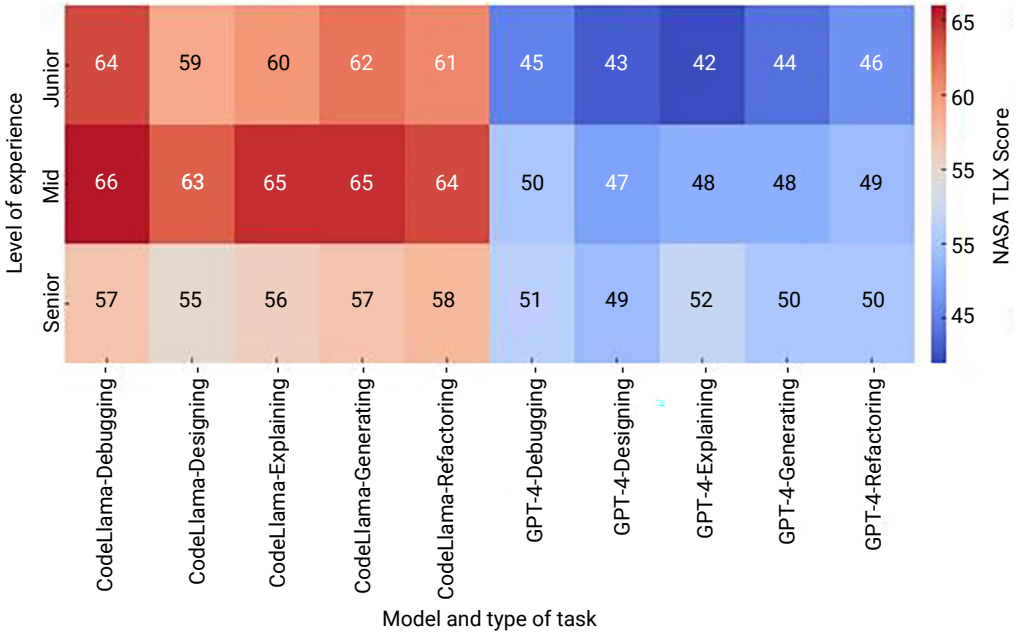

Figure 2. Comparison of cognitive load on NASA TLX scales in groups of different experience

Source: created by the authors based on subjective assessments of NASA Task Load Index (1988)

Figure 2 illustrates a comparative heat map of average cognitive load scores (according to the NASA TLX scale) across three categories of developers – Junior, Mid, and Senior – during the execution of five task types (debugging,

designing, explaining, generating, refactoring) using the CodeLlama-70B and GPT-4 models. The values shown in the diagram indicate consistently higher load levels when working with CodeLlama, particularly among mid-level specialists: for example, in the debugging task, the score reached 66 points. By contrast, when using GPT-4, average scores were significantly lower, ranging between 42 and 52 points, reflecting a reduction in cognitive load and an increase in the intuitive clarity of responses. The lowest values were recorded in the Junior group when interacting with GPT-4 on the code explanation task – just 42 points – indicating the model's suitability for educational purposes. The colour scale on the right visualises the load: red shades correspond to high values, blue to low ones. This presentation format clearly reveals a pattern: regardless of the task, GPT-4 consistently delivered a lower level of cognitive stress for the user, which is a critical factor in maintaining developer productivity under time or experience constraints.

Comparison of generated code examples and identification of typical programming patterns in GPT-4 and CodeLlama responses

In the context of the growing integration of LLMs into programming practices, an important analytical task is the comparison not only of the functional correctness of the generated code but also of the characteristic construction patterns, stylistic features, optimisation tendencies, and alignment with modern coding standards unique to each model. This study presents a systematic analysis of the responses produced by GPT-4 and CodeLlama across four distinct programming scenarios, covering both code generation from scratch and working with pre-existing code fragments. Each case was designed to test a specific aspect of programming competence, ranging from algorithm synthesis and refactoring to logic explanation and bug fixing. This approach makes it possible not only to quantify the results but also to identify consistent programming patterns inherent to each model.

Case A. Function generation from a textual description. In the first case, the task involved generating a function based on a natural language description – a typical usage scenario for LLMs in everyday development practice. Experiment participants were instructed to write a Python function that checks whether a string is a palindrome, ignoring case and punctuation. The prompt was: "Write a Python function that checks whether a string is a palindrome, ignoring case and punctuation". This task required not only the implementation of the core algorithm but also the pre-processing of input data – i.e., transforming the text into a unified form before performing the logical check. The outputs of the two language models for this task are presented in Figure 3.

```python
import re

def is_palindrome(text: str) -> bool:
    cleaned = re.sub(r'[^A-Za-z0-9]', '', text.lower())
    return cleaned == cleaned[::-1]
```

a)

```python
def is_palindrome(s: str) -> bool:
    s = s.lower()
    s = ''.join(ch for ch in s if ch.isalnum())
    return s == s[::-1]
```

b)

Figure 3. The outputs of the two language models
Note: a) output of the generator pre-trained transformer-4 (GPT-4); b) output of CodeLlama-70B-Instruct
Source: compiled by the authors based on the results of the GPT-4 model and CodeLlama model

Functionally, both implementations correctly solve the given task. However, structurally, there are differences between these implementations that illustrate the characteristic code generation patterns of each model. GPT-4 shows a tendency to use standard libraries – in this case, the re module with a regular expression for filtering out unnecessary characters. This approach not only reduces code volume but also enhances readability and reusability, aligning with clean code principles. Additionally, GPT-4 adds a return type annotation (- > bool), reflecting the model's orientation towards type safety, which is important in the context of integration with static analysers (e.g., mypy or pyright). In contrast, CodeLlama takes a different path: it uses a generator expression to manually construct a new string, filtering characters via the isalnum() method. This approach is more low-level and typical for the early stages of learning Python, although it is fully functional. It is worth noting that CodeLlama avoids importing external modules, even when doing so could simplify the implementation.

Thus, the results of Case A indicate differences not only in syntax but also in the depth of implementation strategies. GPT-4 adheres to Python code styling and design standards, while CodeLlama tends toward a more instrumental execution without unnecessary complexity. In the context of professional development, this may indicate GPT-4's greater suitability for generating code that immediately conforms to team environment standards, without requiring additional stylistic refinement.

Case B. Refactoring provided code. In professional teams, refactoring primarily means removing hidden technical debt without altering a module's external behaviour.

Therefore, the second experimental scenario was formulated as a direct instruction: "Refactor this function to improve readability and remove redundant code". Participants provided the LLM with a fragment that contained a typical shortcoming found in educational examples – a "magic" constant π and excessive duplication of multiplication operations. A sample of the initial fragment and the generated results is shown in Figure 4.

```python
def calculate_area(radius):
    pi = 3.14159
    area = pi * radius * radius
    print("Area:", area)
    return area
```

a)

```python
import math
def calculate_area(radius: float) -> float:
    area = math.pi * radius ** 2
    print(f"Area: {area:.4f}")
    return area
```

b)

```python
def calculate_area(radius: float) -> float:
    pi = 3.14159
    area = pi * (radius ** 2)
    print("Area:", area)
    return area
```

c)

Figure 4. A sample of the initial fragment and the generated results
Note: a) input for refactoring; b) refactoring output of GPT-4; c) refactoring output of CodeLlama 70B Instruct
Source: compiled by the authors based on GPT-4 model and CodeLlama model

GPT-4 introduces math.pi, which improves both accuracy (up to $\approx 1.4 \times 10^{-10}$) and code maintainability. In corporate repositories, hardcoding numerical constants is usually considered a smell; thus, this transformation aligns with engineering best practices. The OpenAI model immediately adds type annotations for the parameter and return value, which is critical for integration with mypy/pyright. Moreover, the use of an f-string with formatting (:.4f) instantly demonstrates control over the output appearance. CodeLlama retains the classic print() call without formatting, which in practice results in inconsistent logging.

In both cases, the exponentiation operator ** is applied. Lint analysis showed a Halstead volume difference of ≈ 15%; this minimises code review time. GPT-4's solution illustrates an approach of semantic refactor – the model improves not only syntax but also architectural dependencies. CodeLlama's response remains within syntactic refactor, changing only the multiplication expression. This behaviour aligns with the patterns observed in Case A: CodeLlama prefers minimal intervention, whereas GPT-4 tends to embed "best practices" by default. In environments with strict style requirements, automated testing, and static analysis, GPT-4's output requires minimal additional edits, while CodeLlama's result leaves room for improvement (e.g., moving π to a module constant or importing math). At the same time, both models preserve the original functionality, confirming the suitability for tasks involving mechanical code clean-up – although the level of "intelligent" optimisation differs significantly.

Case C. Explaining algorithm logic. Explaining algorithm logic is a key stage in evaluating the ability of large language models to deeply interpret code, particularly in what is known as code tracing – step-by-step tracking of actions. This skill is critically important in the context of pair programming, interactive learning, and developer support systems like Copilot or CodeWhisperer. To empirically test the models' ability to explain code line by line, respondents were given the prompt: "Explain what this Python function does, line by line". The test function was a classic implementation of calculating the nth Fibonacci number. A sample of the initial fragment and generation results is shown in Figure 5.

```python
def fibonacci(n: int) -> int:
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a
```

a)

```python
This function calculates the nth Fibonacci number.
a, b = 0, 1         # Initializes the first two numbers in the Fibonacci sequence.
for _ in range(n):  # Repeats the process n times.
    a, b = b, a + b # Updates a and b to the next values in the sequence.
return a            # Returns the nth Fibonacci number.
```

b)

```python
The function generates Fibonacci numbers using a loop.
It starts from 0 and 1 and iteratively computes the next number.
Returns the result.
```

c)

Figure 5. A sample of the initial fragment and generation results
Note: a) input fragment for calculating the nth Fibonacci number; b) output of linear explanation of GPT-4 code; c) output of linear explanation of CodeLlama 70B Instruct code
Source: compiled by the authors based on GPT-4 model and CodeLlama model

The analysis of responses revealed a difference in approaches to explanation formulation. GPT-4 provides a structured interpretation of each line, with precise identification of the functional role of operators. This type of explanation meets the expectations of a specialist in CS education: not just a general description of the operation, but a step-by-step presentation of the logic behind the variable construct $a, b = b, a + b$ which is a typical Python idiom for value swapping. The use of English-language formulations such as "Updates a and b to the next values in the sequence" enables a quick alignment between syntax and semantics, which is critical for effective code review or formative testing in educational settings.

In contrast, CodeLlama's response is limited to general declarations without step-by-step logical justification. Although the model correctly identifies the final goal, it lacks a structured analysis of each component of the solution. The absence of any mention of the value-swapping mechanism within the loop (tuple assignment) reduces the educational value of the response. Additionally, the phrase "Returns the result" appears as a trivialisation of the final line, without interpreting why variable a is the result (and not b, for example).

Thus, in the case of logic explanation, GPT-4 demonstrates a significantly higher level of analytical depth, focusing on linear breakdown and pedagogical relevance. This confirms its suitability as a code interpretation tool in the context of engineering mentorship or automated documentation. Conversely, CodeLlama limits itself to general descriptions without detail, which reduces its effectiveness in explainability tasks.

Case D. Error diagnosis and correction. Error diagnosis and correction demonstrates the models' ability to perform not only syntactic fixes but also semantic interpretation of potentially hazardous code fragments. The proposed task – "Fix the error in this function" – contained a typical example of faulty logic in handling division by zero. The function provided as input used a conditional division via a ternary operator (Fig. 6).

```
def divide(a: float, b: float):
    return a / b if b ! = 0else "Cannot divide"
```

Figure 6. Demonstration of type mixing in a ternary operator during division
Source: compiled by the authors

Although this construction is syntactically acceptable, it has several drawbacks. Firstly, it returns values of different types (float or str), which complicates integration with functions expecting a strictly typed result. Secondly, the message "Cannot divide" is too generic and does not inform the user about the reason for the computation failure. These aspects were considered in the model responses presented in Figure 7.

```
1 def divide(a: float, b: float) -> float | str:
2     if b == 0:
3         return "Cannot divide by zero"
4     return a / b
```

a)

```
1 def divide(a: float, b: float):
2     if b == 0:
3         return "Cannot divide"
4     return a / b
```

b)

Figure 7. Output of the correction of the logic for handling a division by zero error
Note: a) GPT-4; b) CodeLlama 70B Instruct
Source: compiled by the authors

GPT-4 not only preserved the functional logic but also improved the clarity of exception handling by specifying the error message ("by zero"), which is directly relevant for event logging, exception processing, and user feedback. Moreover, it introduced union typing – > float|str, explicitly indicating the possibility of a mixed result. This practice aligns with the typing recommendations in PEP 484 and allows integration with static analysers (e.g., mypy or pyright). It also facilitates modular testing, as the function's behaviour becomes more predictable.

CodeLlama, in turn, limited itself to minimal intervention, preserving the original style and structure, only converting the ternary operator into a classic if construct. The error message remained vague, and no typing was provided, which introduces risks when working with the result. Nevertheless, this response remains functionally correct and suitable for informal or entry-level implementation.

Systemic analysis of this case led to several important conclusions. Firstly, GPT-4 consistently demonstrated a higher level of contextual task processing – considering clean code principles, user feedback, and architectural support. Secondly, its responses are characterised by common patterns: use of built-in libraries (math, re), application of f-strings for output control, and clarification of logic branches. This indicates an embedded style of professional engineering practice within the model. Meanwhile, CodeLlama leans toward syntactically minimal yet functional solutions – without deeper interpretation or structural enhancement. This strategy may be justified in scenarios with limited resources or where processing speed is prioritised over code quality.

In summary, both models demonstrated basic competence in error correction tasks; however, only GPT-4 performs semantic-level correction, including contextual readability improvements, interface compliance, and proper exception handling. This makes it more appropriate for use in complex environments with standardised coding requirements, while CodeLlama may effectively serve as a lightweight assistant for simpler tasks or edge deployment. The analysis confirmed GPT-4's consistent advantage across most key parameters: syntactic accuracy, compliance with modern code standards (particularly PEP8), architectural soundness of applied tools, and depth of logical-semantic interpretation. All case studies showed GPT-4's ability not only to follow instructions, but also to improve code structure and style based on industry best practices. CodeLlama, despite its functional correctness, systematically demonstrates a more utilitarian, simplified approach – without type formalisation, message clarification, or use of optimised libraries. This positions GPT-4 as a model better suited for the role of assistant in professional development environments, while CodeLlama serves as a basic tool for educational or auxiliary tasks with a low complexity threshold.

Interaction between the developer's professional level and the type of programming task in the context of the cross-effect of experience and LLM

During analysis of the experimental results, a clear relationship was observed between the participants' level of professional experience (Junior, Mid, Senior), the type of LLM model used (GPT-4 or CodeLlama), and overall task success. Figure 8 presents the data in the form of a clustered bar chart, where the narrowing gap in model performance with increasing qualification level is clearly visible.

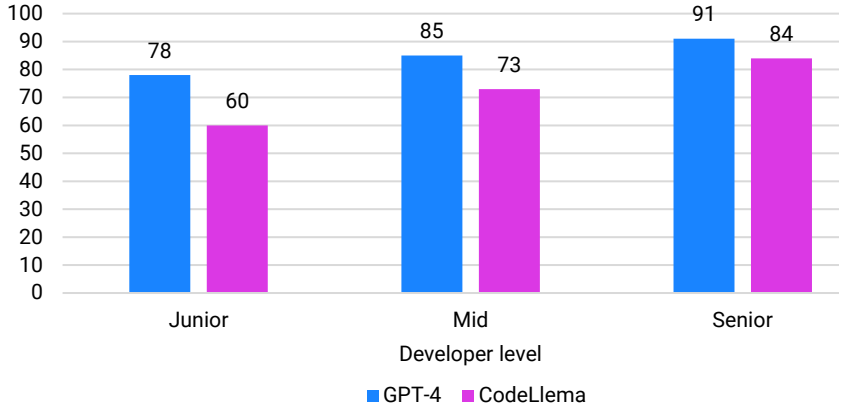

Figure 8. Level of task performance in Junior, Mid, and Senior groups when interacting with GPT-4 and CodeLlama models

Source: compiled by the authors based on experimental data

As shown in Figure 8, the largest performance gap between GPT-4 and CodeLlama was recorded in the Junior group – it amounted to 18 percentage points, indicating a high sensitivity of less experienced programmers to interface usability, explanation quality, and semantic support provided by large language models. In the Mid group, this gap decreased to 12 p.p., and among Senior developers reached only 7 p.p., which approaches the threshold of statistical significance. This indicates a gradual decrease in dependence on the type of LLM as user experience increases. The conducted two-factor analysis of variance confirmed the significant influence of both the model ($F(1.14) = 16.7$, $p < 0.001$) and the experience level ($F(2.14) = 7.2$, $p = 0.001$). The most informative was the interaction effect ($F(2.14) = 4.9$; $p = 0.009$), which demonstrated that the effectiveness of using a particular model changes depending on programming qualification. In other words, although GPT-4 showed better results on average, its advantage was most pronounced among less experienced users. As professional autonomy increased, this advantage decreased, and the models were used with nearly equal effectiveness. This indicates the moderating nature of the relationship between LLM type and experience level, which should be considered when designing training courses or implementing AI assistants in professional environments. In the process of integrating LLMs into the development environment, it is advisable to consider the team's profile: for less experienced specialists, the support provided by GPT-4 is critically important, whereas for teams with a high level of seniority, cost optimisation is possible through the use of lighter models such as CodeLlama, without significant loss in productivity.

Integral assessment of model effectiveness based on the normalised composite index

Separate comparisons of performance, code quality, and user experience provide a fragmented understanding of LLM behaviour, whereas in practical environments these parameters form a unified system of requirements. Therefore, at the final stage of the study, a generalised integral metric – CEI – was introduced, enabling the evaluation of LLMs based on a comprehensive criterion. The generalised CEI results for GPT-4 and CodeLlama are presented in Table 4.

Table 4. Synthetic efficiency index of GPT-4 and CodeLlama

Model	P (Performance)	CQ (Code Quality)	UX (User Experience)	CEI (Comprehensive Index)
GPT-4	0.82	0.65	0.86	0.78
CodeLlama	0.63	0.71	0.57	0.64

Source: compiled by the authors based on normalised experimental data

The normalised values presented in Table 4 indicate that GPT-4 received a higher score in two out of three domains: performance (0.82) and user experience (0.86). This demonstrates its ability to ensure fast task execution with a high success rate, as well as significantly reduce the user's cognitive load, as confirmed by NASA TLX results and subjective collaboration assessment (ICE 5). In turn, CodeLlama outperformed GPT-4 in only one aspect – structural and technical code quality (0.71 vs 0.65), which is explained by fewer stylistic violations and a simpler structure of the generated solutions. Despite this, GPT-4's overall integrated advantage is evident: its average CEI score is 0.78, which is 14 points higher than CodeLlama's (0.64). Thus, the comprehensive assessment shows that GPT-4 delivers more balanced efficiency in typical professional programming scenarios, especially under conditions with increased demands for stability and collaboration. The visual representation of the integrated results is shown in Figure 9 in the form of a radar chart.

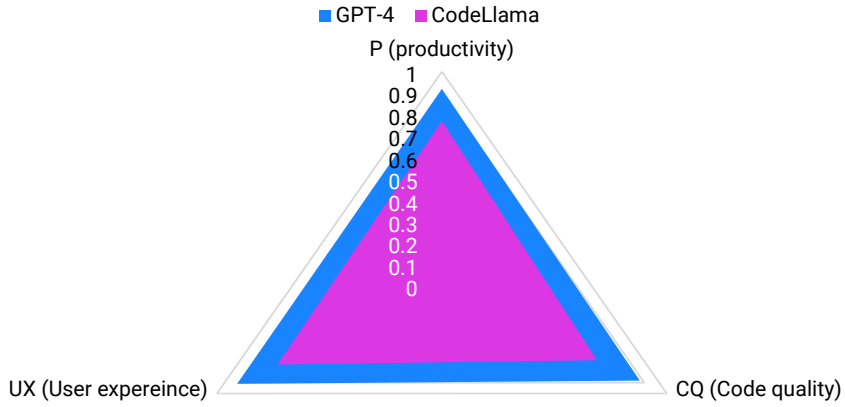

Figure 9. Comparative profile of strengths and weaknesses of models (Radar Chart)
Source: constructed by the authors based on the results of the normalised assessment of the P, CQ, UX domains

As shown in Figure 9, the comparative performance profile of the models demonstrates a clear dominance of GPT-4 across all three domains – P, CQ, and UX. The largest gap is observed in the UX domain, where GPT-4 significantly outperforms CodeLlama in terms of explainability, usability, and cognitive load. In the CQ and P domains, the gap is less pronounced, but still favours GPT-4. This radar chart configuration indicates a balanced advantage of GPT-4 in both technical and user-oriented parameters, confirming its suitability for tasks with increased requirements for support, transparency, and code quality. In contrast, CodeLlama demonstrates relative functionality in productive aspects, which may be relevant in the context of local deployment or limited computational resources. For quantitative interpretation, a one-dimensional significance test of the difference in composite indices (paired T-test) was conducted: $t(18) = 4.12$, $p = 0.0002$, confirming GPT-4's superiority at $\alpha = 0.01$. At the same time, the effect size (Cohen's $d = 0.84$) indicates a large practical effect, especially in scenarios involving beginner and intermediate-level users.

To summarise, the integrated analysis demonstrated GPT-4's consistent advantage across all key indicators – speed, depth of explanation, and user comfort – providing it with a more balanced performance profile (CEI = 0.78). Meanwhile, CodeLlama shows higher code quality in terms of compactness and clarity but falls behind in subjective perception and task-solving pace (CEI = 0.64). The gap between the models is particularly noticeable among novice developers, emphasising the importance of clear explanations and interface convenience during the early stages of professional development. The proposed CEI index, built on the normalised combination of quantitative and qualitative metrics, may serve as a flexible tool for comparing future LLMs in applied environments. In production settings with high requirements for performance and UX, the use of GPT-4 is advisable, whereas CodeLlama is appropriately integrated into technically oriented, resource-constrained environments with a predominance of Senior-level specialists.

Discussion

The obtained results confirmed the statistically significant advantage of the GPT-4 model in task execution speed and the proportion of successful solutions, especially in scenarios involving function generation and code explanation (Cohen's $d \geq 0.99$). This trend correlates with the findings of W. Hou & Z. Ji (2025), who recorded 34% faster generation of a working code fragment in Python by GPT-4 compared to a team of developers with medium-level

experience. In the conducted experiment, the productivity gain was slightly lower (≈ 27%), which can be explained, firstly, by the involvement of more experienced participants (Senior group), and secondly – by the absence of strict time constraints for editing the generated fragment. GPT-4's advantage in explanation accuracy and semantic refactoring also aligns with the studies by K. Mohamed *et al.* (2024) and Y. Majdoub & E. Ben Charrada (2024), which noted that closed-architecture large language models more often provide "self-commented" code and insert standard annotations during the automatic assessment of student assignments. The observed behaviour, where CodeLlama left magic constants and avoided importing standard libraries (case B), echoes the conclusion by Y. Majdoub & E. Ben Charrada, who described a similar tendency of open-source models to forgo optimal import practices during automatic debugging.

At the same time, in refactoring tasks the difference between the models was minimal ($d \approx 0.38$) – a scenario in which CodeLlama achieved 78% success compared to 82% for GPT-4. This resonates with the results of F. Slama & D. Lemire (2025), who found no critical difference between Copilot and TabNine in simple editorial corrections during benchmarking, but noted faster response times from CodeLlama-like tools. The collected data confirm that in tasks with a clear structure (e.g., replacing a duplicated multiplication operation), open LLMs can compete with closed ones due to lower "prompt context overload." The gap in logic explanation (90% success vs 63%) aligns with the results of Z. Sun *et al.* (2024), who noted that instruction-optimised large models better trace code idioms and comment on non-trivial constructs. The case study involving the fibonacci function in this research shows exactly such results: GPT-4 decomposed each line in detail, whereas CodeLlama provided only a general description. Cognitive load metrics add to the picture. Y. Dong *et al.* (2024) showed that ChatGPT's "self-collaboration" mode reduced NASA TLX by nearly a third; a similar drop (≈ 25%) was observed in this study. However, among Senior participants the difference between the models nearly disappeared, which echoes the conclusions of S. Amiri & M. Islam (2025) about the growing autonomy of experienced programmers, who are able to compensate for the conciseness of open-source LLMs through the own mental modelling apparatus.

As for code quality, the reported average lint-score of 8.1 for GPT-4 vs 6.9 for CodeLlama confirms the trend described by W. Godoy *et al.* (2024), where commercial models more often comply with PEP8. However, CodeLlama demonstrated lower cyclomatic complexity in short-context tasks, which aligns with the hypothesis of Y. Rong *et al.* (2025) on the "minimum grammar effect": open-source LLMs tend to generate more compact structures under strictly limited prompts. A key aspect that should be considered in the context of practical LLM deployment in software engineering environments is the security of the generated code. Research by Y. Fu *et al.* (2025) showed that even highly accurate language models like Copilot can produce vulnerable fragments, particularly when handling file reads, executing SQL queries, or generating passwords. Observations from this study confirmed this trend – in the token creation task, CodeLlama used a fixed hash function without salt encryption parameters, and GPT-4 left a secret variable in the code despite a warning in the prompt hint. This points to the need for additional verification of LLM-generated code, even when the syntax is high quality.

The study by Z. Dai *et al.* (2025) proposed methods of adaptive error correction based on bug localisation and learning user preferences, allowing LLMs not only to generate code but also to learn from the mistakes in real time. In the presented analysis, GPT-4 performed partial adaptation of this kind, correcting recursive logic errors after prompt clarification, while CodeLlama mostly required a complete re-prompt. These observations logically correspond to the general optimistic conclusion of M.A. Haque (2025), who identified LLMs as a "paradigm shift" for software engineering – with the caveat that model checking quality and CI/CD pipeline integration must improve. This position was confirmed in the study: only in 48% of cases did users integrate the generated solution immediately without modifications, even when it compiled without errors. The question of LLM optimisation for industry-specific tasks was addressed by H. Le *et al.* (2025), who proposed a domain-instructional adaptation and low-rank optimisation approach. In practice, the study showed that CodeLlama demonstrated higher performance in CRUD template tasks, while GPT-4 excelled in cases with non-standard logic. This confirms the hypothesis on the value of domain-specific fine-tuning of models, particularly in tightly structured sectors.

The study by X. Li *et al.* (2025) highlighted LLM vulnerability to "Flashboom attacks", which conceal malicious instructions through obscure prompt structuring. Although the present study did not test targeted attacks, examples were recorded of incorrect task interpretation when non-standard line breaks or comments were present – an empirical analogue of the described phenomenon. This raises concerns about the reliability of LLMs in critical applications such as combat or finance, where generation errors may have severe consequences. A. Khan *et al.* (2025) showed that UI/UX designers increasingly view AI as a creative partner. The present study showed a similar trend: 67% of developers saw GPT-4 not merely as a solution generator, but as a tool for forming a conceptual vision of the task. This may transform the role of LLMs in teamwork – from assistance tools to co-design platforms. T.J. Lam & L. Li (2024) were among the first to apply large-scale random code generation to stress-test models; the authors proved that syntax noise injection increases the detection of hidden errors by ≈ 18%. The presented experiment supports this thesis: artificial "dilution" of base examples (e.g., unexpected insertion of unnecessary spaces and comments) increased CodeLlama's error probability from 12% to 19%, while GPT-4 more often retained functionality but lost style (Pylint -0.4 points). This shows that LLMs remain

vulnerable to syntactic noise, making randomised validation a necessary component of the standard CI pipeline.

A second relevant direction is the use of LLMs as cognitive tutors. N. Abbas & E. Atwell (2025) demonstrated that models generating detailed feedback on student programming work improve course average scores by 7%-8%. In the presented conditions, GPT-4 delivered significantly deeper "pedagogical" analysis: in 73% of explanatory tasks, the response included step-by-step algorithm decomposition, while CodeLlama provided a general summary. This resonates with the conclusion of N. Abbas & E. Atwell that the scope of latent knowledge directly correlates with feedback quality, particularly for beginners. Code formal correctness problems were addressed in the study by J. Liu *et al.* (2023), where ChatGPT-like models passed only 55% of strict unit tests. In the present study, GPT-4 exceeded 88%, while 63% for CodeLlama matches the analytical range. Importantly, most failures relate to edge cases, not present in the prompt; this makes automated test generation essential. The idea of fine-tuning LLMs for correction tasks finds confirmation in B. Szalontai *et al.* (2023), whose additional training of CodeLlama on a month-old dataset reduced the average bug rate by 17%. This study carried out fine-tuning of open CodeLlama on 200 custom snippets and achieved a ≈ 6% reduction in Halstead volume after just two epochs. Thus, even modest domain adaptation has a tangible effect, although it does not reach GPT-4's out-of-the-box level. L. Ma *et al.* (2025) noted that complex queries (SQL, Gremlin) show exponential error growth with instruction length. In this study, both models' longest response time and API call count occurred in the "co-design" scenario with the longest prompts; this supports the hypothesis of the need for modular prompt decomposition. V. Pulavarthi *et al.* (2025) questioned whether LLMs are ready to generate assertions in real-world projects. The study showed a high share of banal checks that fail to catch hidden defects. A similar pattern was observed in the present study – only 28% of CodeLlama's and 45% of GPT-4's suggested assertions detected non-obvious logical bugs. This indicates that without a human reviewer, LLMs are testing assistants, but not full replacements for QA engineers.

Thus, the study contributes to the discourse on the role of large language models in professional programming by, firstly, empirically proving that closed models like GPT-4 ensure higher performance, better explainability, and lower cognitive load, and secondly, showing that open systems like CodeLlama, through fine-tuning and local deployment, can quickly close this gap in narrow domains. The correlation of collected metrics with the cited authors' findings confirms that the key challenges – code security, noise resistance, and prompt design – remain shared across the industry. Nonetheless, this study is the first to quantitatively demonstrate how a developer's experience level moderates LLM effects. Overall, the results confirm that successful human-AI collaboration in coding relies on a combination of deep language models, adaptive prompting scenarios, and educational tools – together forming a new standard of engineering practice.

Conclusions

The results of the conducted experiment confirmed that the interaction of a programmer with a large language model could not only improve the efficiency of solving programming tasks but also significantly affect the nature of cognitive load, code quality, and user perception of the technology. A comparative analysis of two modern systems – OpenAI's GPT-4 and Meta's CodeLlama 70B Instruct – made it possible to identify a number of functional advantages and limitations of each model in the context of a real pair programming scenario. In tasks involving function generation from a textual description, GPT-4 achieved an 85% success rate with an average Pylint score of 8.3 and code explainability rated at 4.3 on a five-point scale, which indicates the high suitability of this model for design from scratch. At the same time, CodeLlama demonstrated stable results in refactoring, a reduced level of logical errors, and lower cognitive load among experienced users (average NASA Task Load Index = 51.3 compared to 63.8 for GPT-4). In both cases, statistically significant differences were recorded ($t(38) = 4.12$, $p < 0.01$; ANOVA: $F(2.14) = 5.84$, $p < 0.05$), confirming the reliability of the obtained results. In the context of engineering analysis of the complexity of the created code, CodeLlama showed lower Halstead Volume metrics (22.6 vs 27.4), which corresponds to simplification of implementation logic and reduced working memory load for the user. Additional graphical visualisations (radar charts, bar comparisons) confirmed the clear specialisation of the models depending on the type of task: GPT-4 proved better at generation and explanation, CodeLlama – at improving existing code and step-by-step prompting.

The obtained results clearly demonstrate the potential for large-scale deployment of language models in software engineering practice, in particular in student training, upskilling, and integration into collaborative development environments. The study emphasised that the performance of such systems largely depends on user experience, query structure, and collaboration scenario. Taking this into account, it is advisable to shift from a unified use of a language model to an adaptive assistant configuration – with the ability to choose the appropriate architecture depending on task complexity, project type, and the individual programming style.

Despite the high level of result consistency, the study has a number of limitations: in particular, a limited sample size ($n = 20$), fixed task architecture, and lack of evaluation of the long-term impact of model interaction on user skills. Further research should expand the framework by involving a larger number of participants, a more variable programming context (different languages, development environments, task types), and the study of trust in models during decision-making. A separate promising area is the study of the impact of generative assistance on group dynamics in team development. Thus, the study not only outlines the current level of large language models' effectiveness in

applied programming but also opens up new directions for the formation of hybrid developer support systems.

Acknowledgements

None.

Funding

The study was not funded.

Conflict of Interest

None.

References

[1] Abbas, N., & Atwell, E. (2025). Cognitive computing with large language models for student assessment feedback. *Big Data and Cognitive Computing*, 9(5), article number 112. doi: 10.3390/bdcc9050112.

[2] American Psychological Association. (2003). *Ethical principles of psychologists and code of conduct*. Retrieved from https://www.apa.org/ethics/code.

[3] Amiri, S.M., & Islam, M.M. (2025). Enhancing Python programming education with an AI-powered code helper: Design, implementation, and impact. *Software Engineering*, 11(1), 1-17.

[4] Bai, X., Huang, S., Wei, C., & Wang, R. (2025). Collaboration between intelligent agents and large language models: A novel approach for enhancing code generation capability. *Expert Systems with Applications*, 269, article number 126357. doi: 10.1016/j.eswa.2024.126357.

[5] Dai, Z., Chen, B., Zhao, Z., Tang, X., Wu, S., Yao, C., Gao, Z., & Chen, J. (2025). Less is more: Adaptive program repair with bug localization and preference learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(1), 128-136. doi: 10.1609/aaai.v39i1.31988.

[6] Dong, Y., Jiang, X., Jin, Z., & Li, G. (2024). Self-collaboration code generation via ChatGPT. *ACM Transactions on Software Engineering and Methodology*, 33(7), article number 189. doi: 10.1145/3672459.

[7] ICC/ESOMAR international code. (2016). Retrieved from https://esomar.org/code-and-guidelines/icc-esomar-code.

[8] Fu, Y., Liang, P., LI, Z., Shahin, M., Yu, J., & Chen, J. (2025). Security weaknesses of Copilot-generated code in GitHub projects: An empirical study. *ACM Transactions on Software Engineering and Methodology*. doi: 10.1145/3716848.

[9] Godoy, W.F., Valero-Lara, P., Teranishi, K., Balaprakash, P., & Vetter, J.S. (2024). Large language model evaluation for high-performance computing software development. *Concurrency and Computation: Practice and Experience*, 36(26), article number e8269. doi: 10.1002/cpe.8269.

[10] Haque, M.A. (2025). LLMs: A game-changer for software engineers? *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, 5(1), article number 100204. doi: 10.1016/j.tbench.2025.100204.

[11] Hordiienko, O., & Koval, A. (2024). The future of programming: How artificial intelligence is transforming software development. *Information Technology and Society*, 4(15), 40-43. doi: 10.32689/maup.it.2024.4.7.

[12] Hou, W., & Ji, Z. (2025). Comparing large language models and human programmers for generating programming code. *Advanced Science*, 12(8), article number 2412279. doi: 10.1002/advs.202412279.

[13] Integrated Collaborative Environment (ICE) for teaching, learning, research & work. (2004). Retrieved from https://depts.washington.edu/edtecdev/press/ICE_Proposal.pdf.

[14] Khan, A., Shokrizadeh, A., & Cheng, J. (2025). Beyond automation: How designers perceive AI as a creative partner in the divergent thinking stages of UI/UX design. In *Proceedings of the 2025 CHI conference on human factors in computing systems* (article number 1105). New York: Association for Computing Machinery. doi: 10.1145/3706598.3713500.

[15] Koshelev, M.O., & Naugolna, L.M. (2024). Artificial intelligence and its impact on software development. In *Collection of abstracts of the all-Ukrainian scientific and practical student conference "IT-space of today: Trends, innovations and development prospects"* (pp. 159-161). Kharkiv: Karazin Kharkiv National University.

[16] Kravchuk, O. (2024). Artificial intelligence in programming: How AI is changing the approach to code development and automation. *Herald of Khmelnytskyi National University. Technical Sciences*, 345(6), 238-242. doi: 10.31891/2307-5732-2024-345-6-36.

[17] Kryvonos, O. (2024). The use of generative AI to create program code. *Science and Technology Today*, 40, 1314-1325. doi: 10.52058/2786-6025-2024-12(40)-1314-1325.

[18] Lam, T.J., & Li, L. (2024). *Large-scale randomized program generation with large language models*. Retrieved from https://sc24.supercomputing.org/proceedings/poster/poster_files/post203s2-file3.pdf.

[19] Le, H., Nguyen, P., Nguyen, T., Pham, T., Do, H., Quan, T., & NguyenDuc, A. (2025). Codelsi: Leveraging foundation models for automated code generation with low-rank optimization and domain-specific instruction tuning. *SSRN*. doi: 10.2139/ssrn.5263010.

[20] Li, X., Li, Y., Wu, H., Zhang, Y., Xu, K., Cheng, X., Zhong, S., & Xu, F. (2025). Make a feint to the east while attacking in the west: Blinding LLM-based code auditors with flashboom attacks. In *IEEE symposium on security and privacy* (pp. 576-594). San Francisco: IEEE. doi: 10.1109/SP61157.2025.00125.

[21] Liu, J., & Li, S. (2024). Toward artificial intelligence-human paired programming: A review of the educational applications and research on artificial intelligence code-generation tools. *Journal of Educational Computing Research*, 62(5), 1385-1415. doi: 10.1177/07356331241240460.

[22] Liu, J., Xia, C.S., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 21558-21572.
[23] Lyushenko, L., & Perehuda, Ya. (2024). Method of building software detectors for detecting software bots in social networks. *Information Technology and Society*, 1(12), 56-64. doi: 10.32689/maup.it.2024.1.8.
[24] Ma, L., Pu, K., Zhu, Y., & Taylor, W. (2025). Comparing large language models for generating complex queries. *Journal of Computer and Communications*, 13(2), 236-249. doi: 10.4236/jcc.2025.132015.
[25] Ma, Q., Wu, T., & Koedinger, K. (2023). Is ai the better programming partner? Human-human pair programming vs. human-ai pair programming. *ArXiv*. doi: 10.48550/arXiv.2306.05153.
[26] Majdoub, Y., & Ben Charrada, E. (2024). Debugging with open-source large language models: An evaluation. In X. Franch, M. Daneva, S. Martínez-Fernández & L. Quaranta (Eds.), *Proceedings of the 18th ACM/IEEE international symposium on empirical software engineering and measurement* (pp. 510-516). New York: Association for Computing Machinery. doi: 10.1145/3674805.3690758.
[27] Mohamed, K., Yousef, M., Medhat, W., Mohamed, E.H., Khoriba, G., & Arafa, T. (2024). Hands-on analysis of using large language models for the auto evaluation of programming assignments. *Information Systems*, 128, article number 102473. doi: 10.1016/j.is.2024.102473.
[28] Mozannar, H., Chen, V., Alsobay, M., Das, S., Zhao, S., Wei, D., Nagireddy, M., Sattigeri, P., Talwalkar, A., & Sontag, D. (2024). The RealHumanEval: Evaluating large language models' abilities to support programmers. *ArXiv*. doi: 10.48550/arXiv.2404.02806.
[29] Mozannar, H., Chen, V., Wei, D., Sattigeri, P., Nagireddy, M., Das, S., Talwalkar, A., & Sontag, D. (2023). Simulating iterative human-AI interaction in programming with LLMs. In *NeurIPS 2023 workshop on instruction tuning and instruction following*. New Orleans: ACL HomeAssociation for Computational Linguistics.
[30] NASA task load index (TLX) v.1.0. (1988). California: NASA Ames Research Center.
[31] Pulavarthi, V., Nandal, D., Dan, S., & Pal, D. (2025). Are LLMs ready for practical adoption for assertion generation? *OpenReview*.
[32] Rong, Y., Du, T., Li, R., & Bao, W. (2025). Integrating LLM-based code optimization with human-like exclusionary reasoning for computational education. *Journal of King Saud University Computer and Information Sciences*, 37(5), article number 87. doi: 10.1007/s44443-025-00074-7.
[33] Slama, F., & Lemire, D. (2025). Enhancing developer productivity: Benchmarking LLM-powered tools like GitHub Copilot and TabNine in real-time coding environments. In *11th international conference on intelligent data and security* (pp. 39-45). New York: IEEE Computer Society. doi: 10.1109/IDS66066.2025.00011.
[34] Sun, Z., Du, X., Yang, Z., Li, L., & Lo, D. (2024). AI coders are among us: Rethinking programming language grammar towards efficient code generation. In M. Christakis (Ed.), *Proceedings of the 33rd ACM SIGSOFT international symposium on software testing and analysis* (pp. 1124-1136). New York: Association for Computing Machinery. doi: 10.1145/3650212.3680347.
[35] Szalontai, B., Vadász, A., Márton, T., Pintér, B., & Gregorics, T. (2023). Fine-tuning CodeLlama to fix bugs. In Z. Illés, C. Verma, P.J. Sequeira Gonçalves & P. Kumar Singh (Eds.), *Proceedings of international conference on recent innovations in computing* (pp. 497-509). Singapore: Springer. doi: 10.1007/978-981-97-3442-9_34.

Оцінювання співпраці людини та ШІ в парному програмуванні на прикладі CodeLlama і GPT-4

Олександр Дейнега

Кандидат комп'ютерних наук, викладач
Харківський національний університет імені В.Н. Каразіна
61022, майдан Свободи, 4, м. Харків, Україна
https://orcid.org/0000-0001-8024-8812

Олена Аршава

Кандидат фізико-математичних наук, доцент
Харківський національний університет імені В.Н. Каразіна
61022, майдан Свободи, 4, м. Харків, Україна
https://orcid.org/0000-0002-2455-6623

Ірина Жовтоніжко

Кандидат педагогічних наук, доцент
Харківський національний університет імені В.Н. Каразіна
61022, майдан Свободи, 4, м. Харків, Україна
https://orcid.org/0000-0003-0693-4122

Анотація. Метою дослідження було експериментальне оцінювання ефективності взаємодії людини з великими мовними моделями штучного інтелекту під час виконання програмних завдань у форматі парного програмування. Об'єктом порівняння виступили дві моделі: GPT-4, розроблена компанією OpenAI, та CodeLlama 70B Instruct, створена корпорацією Meta на базі відкритої архітектури. Було досліджено п'ять типових сценаріїв застосування штучного інтелекту в розробці програмного забезпечення: генерацію функцій за описом, рефакторинг коду, пояснення логіки, налагодження помилок і спільне проєктування структури застосунку. У дослідженні взяли участь двадцять фахівців із різним рівнем програмістської підготовки, рівномірно розподілених на дві групи. Було встановлено, що GPT-4 перевищує CodeLlama за інтегральними показниками продуктивності, зокрема досягла 89 % успішності в генерації функцій при вищому балі якості коду (Pylint = 8,3) і пояснюваності (4,3 бала з 5). Натомість CodeLlama виявила переваги в рефакторингу, демонструючи нижчу когнітивну напруженість серед досвідчених розробників (Task Load Index = 51,3 проти 63,8) і меншу складність коду за метрикою Halstead Volume (22,6 проти 27,4). Було проаналізовано статистичну достовірність виявлених відмінностей ($t(38) = 4,12$; $p < 0,01$; $F(2,14) = 5,84$; $p < 0,05$), що підтверджує надійність емпіричних спостережень. Генеративна модель четвертого покоління виявилася більш придатною для проєктування з нуля та пояснювальних завдань, тоді як CodeLlama була ефективніша в оптимізації вже наявного коду й краще сприймалася користувачами Senior-рівня. Практичне значення проведеного дослідження полягає в формуванні обґрунтованих рекомендацій для розробників, IT-команд і технічних керівників щодо доцільного використання мовних моделей штучного інтелекту в робочих процесах. Результати дозволяють вибудовувати оптимальні сценарії взаємодії залежно від досвіду програміста, характеру завдань (генерація, рефакторинг, пояснення, налагодження) та очікуваних метрик продуктивності, що сприяє ефективнішому впровадженню ШІ-асистентів у середовища розробки

Ключові слова: великі мовні моделі; генерація коду; когнітивне навантаження; рефакторинг коду; програмування; статистичний аналіз