BULLETIN of Cherkasy State Technological University

Journal homepage: https://bulletin-chstu.com.ua/en

Vol. 30 No. 4. 2025

UDC 004.8:005.7
DOI: 10.62660/bcstu/4.2025.128

A comparative analysis of CodeBERT and CodeLlama models: Architecture, functionality and application in software coding tasks

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

Irina 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 relevance of the research was conditioned by the need to compare the large language models CodeBERT and CodeLlama, which were actively used for automating code generation and analysis with the aim of improving the efficiency and quality of software. The aim of the study was a comprehensive juxtaposition of the architectural and functional characteristics of the selected language models CodeBERT and CodeLlama. Interpretative, comparative, systemic and structural-categorical analyses were used to study the architectures, tasks, and relevance of the models. A comprehensive comparative analysis of the CodeBERT and CodeLlama models was carried out according to key parameters: model architecture (the RoBERTa encoder architecture in CodeBERT versus the Llama 2 decoder architecture in CodeLlama), the scale and sources of training data, the range of supported tasks, performance on benchmark datasets, advantages and limitations, typical areas of application, and conditions of accessibility and licensing. The results showed that the difference in architecture and training data significantly affected the effectiveness of the models in different types of tasks, and also determined the practical capabilities and limitations. Particular attention was paid to the issues of implementing the models in practical scenarios, taking into account hardware resources and licensing policy. The results showed that CodeLlama required significantly greater computational resources for effective operation, whereas CodeBERT was easier to implement on standard equipment. It was also established that the licensing conditions of CodeLlama were more restrictive, which could complicate its use in commercial projects, in contrast to CodeBERT with an open licence. It was concluded that these models performed predominantly complementary functions: CodeBERT was an effective tool for code-understanding tasks, whereas CodeLlama demonstrated high results in generation tasks. The conclusions outlined the challenges and prospects for the development of next-generation models with multitasking and multimodality. Practical value – assistance to developers and researchers in choosing the optimal tool, taking into account technical and licensing aspects

Keywords: large language models; encoder transformer architecture; decoder architecture; systemic and functional analysis; optimisation model; statistical analysis; natural language processing

Article's History: Received: 30.05.2025; Revised: 19.10.2025; Accepted: 15.12.2025.

Suggested Citation:
Deineha, O., Arshava, O., & Zhovtonizhko, I. (2025). A comparative analysis of CodeBERT and CodeLlama models: Architecture, functionality and application in software coding tasks. *Bulletin of Cherkasy State Technological University*, 30(4), 128-142. doi: 10.62660/bcstu/4.2025.128.

*Corresponding author

INTRODUCTION

The field of software development underwent a significant transformation under the influence of the rapid development of artificial intelligence (AI), in particular large language models (LLMs). These models, trained on large corpora of text and code, demonstrated high capabilities in understanding, generating and manipulating programming languages (PL) alongside natural languages (NL). The intellectualisation of code – that is, the integration of AI to increase developer efficiency – became important due to the growing complexity of software systems and the increase in the number of participants in the development process. Initial successes consisted in the creation of models capable of finding relevant fragments of code in response to natural-language queries or automatically completing code. However, in 2021-2025 more complex architectures appeared that performed the functions of full-fledged programmer assistants, capable not only of generating code, but also of debugging, explaining and optimising it. This significantly expanded the possibilities for automating development and contributed to improving the productivity and quality of software products.

The historical evolution of code-processing models became the subject of a study by Z. Zheng *et al*. (2023). The authors conducted a thorough review of the development of large language models oriented towards program code, noting the evolution from simple models to complex architectures that supported multimodal tasks. The researchers emphasised the importance of integrating both natural language and code to increase the flexibility of models in real applications. The authors also described in detail the challenges associated with an adequate understanding of programming semantics. This review served as a foundation for understanding the trends that influenced the design of modern LLMs and formed the context for comparing the CodeBERT and CodeLlama models. O. Deineha *et al*. (2024), in the study, developed a methodology for extracting data from λ-expressions (lambda terms), which was important for the analysis of functional programming languages. The authors proposed approaches to structuring and processing lambda terms to improve the automatic recognition and transformation of code. This study contributed to improving the efficiency of code analysis at the level of functional constructs. In turn, M.A. Hodovychenko & D.D. Kurinko (2025) carried out a review of existing methods of automated refactoring of object-oriented software systems. The authors analysed both traditional algorithms and modern intelligent approaches, including the use of machine learning to automate the maintenance and improvement of code. The article covered a comparison of methods and the identification of the strengths and weaknesses in the context of improving software quality. O.V. Budzynskyi (2025) researched intelligent analysis methods for detecting SQL-injection attacks in real time. The researchers applied deep neural networks to analyse network traffic and classify potential threats, which allowed dangerous queries to databases to be identified quickly and accurately. This study was aimed at raising the level of information security through the automation of cyber-attack detection.

In turn, M. Weyssow (2024) considered the issue of context alignment as a key factor for improving the quality of code generation by large language models. It was found that taking into account user preferences and the specifics of the current project made it possible to significantly improve the relevance and accuracy of results. The author showed that contextualisation helped to avoid typical errors and increased the practical value of models in developers' daily work. This approach was directly related to the analysis of the performance of CodeLlama, which supported work with large context windows. X. Bai *et al*. (2025) proposed a unique approach in which intelligent agents interacted with LLMs to improve the quality of the generated code, which significantly improved the accuracy and logical integrity of software fragments. The authors focused on the possibilities of combined AI use to overcome typical generation errors and automate complex development tasks. The proposed model also contributed to a better understanding of the internal processes of LLMs and the optimisation. This method supported the idea of a multi-agent architecture, which could be promising for future developments based on CodeLlama.

G. d'Aloisio *et al*. (2025) explored the possibilities of compressing CodeBERT models without significant loss of performance, which was a priority for use on devices with limited computational resources. The researchers proved that model optimisation allowed its size to be reduced, and its operation speeded up while maintaining a high level of accuracy. This expanded the potential spectrum of practical use of models, in particular in embedded systems. The conclusions of the study were important for understanding the trade-offs between scale and efficiency in the comparison of CodeBERT and CodeLlama. K. Huang *et al*. (2025) focused on fine-tuning large models for automatic code correction, demonstrating a significant increase in effectiveness in debugging tasks. The authors showed that adapting models to specific errors and coding styles allowed results to be improved. The study revealed the potential of LLMs in supporting developers, reducing the time spent on debugging. These developments were important for the practical application of CodeLlama, which was oriented towards code generation and correction. Z. Gao *et al*. (2024) proposed an innovative Virtual Compiler approach to improving search in assembly code with the help of an LLM that simulated compiler behaviour. This made it possible to improve the accuracy and relevance of search results in low-level programming. The researchers emphasised the importance of this technology for software security and analysis. The work opened new horizons for integrating LLMs into

specialised areas of development. In turn, N. Raihan *et al*. (2024) developed a systematised taxonomy of code-LLMs that classified models by architecture, functionality, and area of application. The researchers provided a basis for a more structured analysis and comparison of LLMs, which contributed to a better understanding of the diversity of models. The taxonomy helped to identify the strengths and weaknesses of each type of model and directions for improvement. This approach was useful for the informed choice between CodeBERT and CodeLlama depending on research goals.

Despite a significant number of studies devoted to individual large language models for working with code, most of these studies focused either on the technical description of one particular model or on its performance within a narrow set of tasks. At the same time, systematic comparative analyses of models of different architectural types (encoder and decoder), such as CodeBERT and CodeLlama, remained limited, which complicated a comprehensive understanding of the strengths and weaknesses in the context of practical application. In view of this, the aim of the study was to carry out a comprehensive comparative analysis of the CodeBERT and CodeLlama models. The tasks consisted in outlining the key differences and common features of these two influential models in the following critical areas: the origin and aims of the models, the architecture, and training features, core capabilities and tasks, performance on industry benchmarks, strengths and weaknesses, typical areas of application, as well as aspects of accessibility, licensing and implementation complexity. Such a comprehensive approach made it possible to obtain an integrated understanding of the potential and limitations of each model for practical use in the field of automating work with program code.

MATERIALS AND METHODS

The chronological boundaries of the research covered the period 2020-2025, which corresponded to the time of release and active development of the respective models. The study was conducted from February to May 2025. Attention was paid to comparing results from independent sources to increase the objectivity of the conclusions. The method of interpretative analysis (interpretation of primary sources) was used when analysing the architectures and the tasks on which the models were trained. It allowed the intentions of the developers to be comprehended, the declared and achieved goals of the models (for example, the emphasis of CodeBERT on bimodality vs the generative focus of CodeLlama) to be compared, and the correspondence between the structure of the model and its actual performance in tasks to be assessed. The method of comparative analysis was applied, which made it possible to identify the common and distinctive features of the models, in particular the basic architecture, type of transformer and direction of attention. The main areas of application of each model, training principles, the ability to generate text and the specifics of performing typical tasks were considered. Particular attention was paid to the flexibility of the models in working with long contexts, which was of key importance for solving complex tasks in software development. The characteristics of the training data and training objectives of the CodeBERT and CodeLlama models were considered, including the sources and volumes of data, the list of supported programming languages, training approaches, as well as the distinctive features and specialisation.

The functional features of the CodeBERT and CodeLlama models were compared, including the primary focus of operation (code and natural-language understanding in CodeBERT versus code generation and modification in CodeLlama), the types of tasks the models performed (search, classification, analysis versus generation, autocompletion, instructions), architecture (bimodal encoder versus generative decoder), training types (token masking and replaced-token detection versus autoregressive training, infilling, and instruction fine-tuning), support for long contexts, as well as practical application in tasks of understanding the semantics of code and natural language, analytics, and developer assistance. This method made it possible to identify both the functional complementarity of the models and the limitations in specific contexts of use. The method of systems analysis contributed to the integration of individual characteristics of the models into a general analytical framework for assessing the relevance to specific tasks in the field of software development. The method of structural-categorical analysis was applied to organise the obtained information into logical blocks. On its basis, an analytical comparison model with a clear classification of parameters was built. This made it possible to formulate conclusions regarding the strengths and weaknesses of each model in the context of specific software-development tasks.

RESULTS

Architectural features of the CodeBERT and CodeLlama models. The CodeBERT and CodeLlama models represent two conceptually different approaches to transformer architectures in the context of program-code processing tasks. The CodeBERT model is implemented as an encoder transformer architecture based on Robustly Optimised BERT Pretraining Approach (RoBERTa), which, in turn, is an optimised version of BERT. The main feature of this architecture lies in the use of a bidirectional self-attention mechanism, which allows the model to take into account the context both to the left and to the right of each token simultaneously. This approach provides deep contextualisation of input sequences, which is key for tasks of understanding natural language and program code, such as semantic search, classification, similarity comparison, documentation generation, and the detection of duplicates or errors. CodeBERT was trained using the masked language modelling objective, which involves filling in missing

tokens based on the surroundings. This approach works effectively for understanding tasks, but it does not support autoregressive text or code generation – that is, the model is not able to create sequences token by token in a logical order. In this way, CodeBERT specialises in analytical tasks rather than creative synthesis.

By contrast, CodeLlama is built on LLaMA 2 and uses the classical decoder transformer architecture with a unidirectional (causal) attention mechanism. This architecture implements autoregressive language modelling, where each token is generated sequentially on the basis of the previous ones, which is a typical approach for modern large language models. The decoder transformer of CodeLlama allows not only the analysis of provided information, but also the effective generation of new code, the filling in of infilling, and the execution of user instructions in Instruct-type variants. Owing to its architecture, CodeLlama supports the processing of long contexts (up to 100,000 tokens in modified variants), which makes it possible to work with large files and entire projects. The model is optimised for efficient execution on modern computing devices, in particular through the use of 16-bit numbers (FP16 or bfloat16). It is also better suited to an interactive usage scenario, when the user expects the generation, explanation, or transformation of code in a dialogue form (Li *et al.*, 2023).

Thus, the architectural differences between CodeBERT and CodeLlama determine not only the technical aspects of the functioning, but also the strategic approaches to the application: CodeBERT is optimised for the understanding and semantic analysis of code, whereas CodeLlama is oriented towards generation, completion and instruction-based programming. These models represent two opposing approaches to modelling program code: analytical (encoder) and creative (decoder), which allows these models to be chosen depending on the nature of the task set in the field of software engineering. In the context of analysing the architectural foundations of CodeBERT and CodeLlama, particular attention is deserved by the type of transformer architecture on which these models are based. CodeBERT implements an encoder architecture based on RoBERTa, oriented primarily towards deep semantic understanding of code, whereas CodeLlama is built as a next-generation decoder model, optimised for code generation and completion. Table 1 summarises the key differences between these two architectural approaches from the standpoint of the internal structure, principles of working with context, type of attention and target application. The comparison presented in Table 1 demonstrates the fundamental difference between the two main architectural approaches in the field of program-code models.

Table 1. Architectural division of CodeBERT and CodeLlama

Characteristic	CodeBERT (Encoder)	CodeLlama (Decoder)
Basic architecture	RoBERTa (based on BERT)	LLaMA 2
Transformer type	Encoder-only	Decoder-only
Attention direction	Bidirectional	Unidirectional (left-to-right)
Primary application	Semantic understanding	Text/code generation
Training principle	Masked Language Modelling	Causal Language Modelling
Text generation	No	Yes
Tasks	Search, classification, error detection	Generation, completion, infilling
Flexibility in long contexts	Low (up to 512 tokens)	High (up to 100,000 tokens)

Source: compiled by the authors based on M. Siavvas *et al.* (2024)

CodeBERT, as an encoder model with bidirectional attention, demonstrates high efficiency in tasks related to semantic analysis, classification, search, and error detection in code. However, its limitations in processing long contexts and the absence of generative capabilities narrow the range of its application in modern dynamic development environments. By contrast, CodeLlama, as a decoder model with a causal attention mechanism, specialises in generative tasks. It demonstrates an exceptional ability to handle long sequences, automatic completion and instruction following, which is critical in the context of modern integrated with development environments (IDEs) and programming-support tools. The unidirectional attention inherent to decoder transformers, although it does not provide a full global analysis of the input text, is compensated by the model's ability to "think ahead" in the process of generation. Thus, the architectural choice directly affects the range of tasks with which the model can work effectively and determines its functional specialisation: encoder models such as CodeBERT remain effective in analysis scenarios, whereas decoder models such as CodeLlama take on the role of universal generators of program text.

Model scale and number of parameters. The availability of models of different scales is an important aspect when selecting a particular architecture for applications with different requirements for performance, resources, and accuracy. A model's scale is determined primarily by the number of parameters – the number of weight coefficients that it optimises during training. CodeBERT is presented as a single base model, the size of which is approximately 125 million parameters. Such a scale provides sufficient performance for a wide range of code-understanding tasks, including semantic search, classification and defect detection,

while remaining relatively compact and suitable for deployment on standard computing resources. In contrast, CodeLlama offers several scalable model variants that vary significantly in size and computational requirements. Among these, the most common versions have approximately 7 billion, 13 billion and 70 billion parameters (CodeLlama-7B, CodeLlama-13B, CodeLlama-70B respectively). This differentiation allows users to choose a model depending on the specifics of the task: smaller versions are optimal for integration into environments with limited resources and needs for fast responses, whereas large models provide maximum accuracy, flexibility in code generation and support for complex contexts. This scalability of CodeLlama reflects the general trend of modern large language models, where variability in model size helps to balance performance and accessibility, and also allows solutions to be gradually adapted to the requirements of specific engineering tasks (Bhandari *et al.*, 2025). Thus, the availability of several CodeLlama variants significantly expands the potential of its application compared with the fixed architecture of CodeBERT. Table 2 below summarises the main model variants by number of parameters and the intended use.

Table 2. Model variants by number of parameters

Model	Number of parameters	Purpose
CodeBERT	~125 million	Fixed size, oriented towards code understanding
CodeLlama-7B	7 billion	Lightweight model for fast tasks and limited resources
CodeLlama-13B	13 billion	Balance between performance and resources
CodeLlama-70B	70 billion	Maximum accuracy and support for complex tasks

Source: compiled by the authors based on A. Gurjar *et al.* (2023), G. Bhandari *et al.* (2025)

The table shows that CodeBERT has a fixed size of about 125 million parameters, which makes it compact and more accessible for use on standard hardware. This ensures high speed and ease of integration, but imposes limitations on the scale of tasks. By contrast, CodeLlama offers three main versions that differ in the number of parameters, from 7 to 70 billion. This approach allows developers to choose the model that best meets the needs and available computing resources. Versions with a larger number of parameters provide higher accuracy and a better ability to handle complex tasks, but require more powerful equipment for training and inference. Hence, the availability of scalable CodeLlama variants makes it a flexible tool for a wide range of applications, from lightweight projects to large, resource-intensive AI systems. At the same time, CodeBERT remains an effective solution for tasks focused on the understanding and analysis of code with moderate hardware requirements.

The processing type in the CodeBERT and CodeLlama models determines the primary way of interacting with code and, accordingly, influences the application in various AI-for-programming tasks. CodeBERT is a model with an encoder architecture oriented towards code understanding. Its primary task is the analysis, indexing and semantic representation of code to perform tasks such as code search, classification, documentation generation, and the detection of defects and clones. This model is not intended for generating new code, but focuses on deep semantic understanding of existing program constructs, which provides high-quality support for intelligent analysis. CodeLlama, by contrast, is built on a decoder architecture and is optimised for code generation. It is able to create new code, supplement existing fragments, perform infilling (filling in missing parts of code) and follow user instructions in the course of generation. Owing to its scalability and support for long contexts, CodeLlama effectively copes with complex tasks of autocompletion, the creation of software modules and even the writing of full-fledged programmes (Gain *et al.*, 2025). Hence, the main difference in processing type lies in the fact that CodeBERT specialises in the understanding and analysis of code, whereas CodeLlama is intended for generative tasks that require the creative creation and modification of program code. The choice between these models depends on the developer's specific needs: analysis of existing code or generation of new code.

Training data for the CodeBERT and CodeLlama model. The CodeBERT and CodeLlama models differ significantly in the sources, volumes, and types of training data, which affected the capabilities and areas of application. CodeBERT was trained on open GitHub repositories within the CodeSearchNet methodology. The total volume of data amounted to about 2.1 million bimodal "natural language – programming language" (NL-PL) pairs and approximately 6.4 million unimodal functions without accompanying documentation. The main programming languages represented in the data included Python, Java, JavaScript, PHP, Ruby and Go. The model was trained on two main tasks: masked language modelling (MLM), which consists in predicting masked elements in natural and programming languages, and the replaced token detection (RTD) task, which promotes better contextual understanding.

In turn, CodeLlama used deduplicated open data from GitHub, Stack Overflow and other sources, including textual discussions and code explanations. The volume of training data for models of different sizes ranged from 500 billion tokens for the 7B-34B versions to 1 trillion tokens for the 70B model. Approximately 85% of these data were code, 8% – natural language related to code, and 7% – general natural language. CodeLlama supports a wide set of programming languages,

such as Python, C++, Java, PHP, TypeScript, C#, Bash and others. For the CodeLlama-Python versions, around 100 billion Python-oriented tokens were additionally used. The training strategy included autoregressive next-token prediction, infilling (predicting missing code fragments), instruction fine-tuning, which enabled the model to respond effectively to assistant-style prompts, as well as fine-tuning for long-context operation (up to 16 thousand tokens) using adapted RoPE positional encoding. For clarity, a comparative table of the main characteristics of the training data and training objectives of the models is provided (Table 3).

Table 3. Main characteristics of the models' training data and training objectives

Characteristic	CodeBERT	CodeLlama
Data sources	Open GitHub repositories (CodeSearchNet)	GitHub, Stack Overflow, textual discussions, code explanations
Data volume	~2.1 million bimodal NL-PL pairs, ~6.4 million unimodal functions	500 billion tokens (7B-34B versions), 1 trillion tokens (70B version)
Programming languages	Python, Java, JavaScript, PHP, Ruby, Go	Python, C++, Java, PHP, TypeScript, C#, Bash and others
Training objectives	MLM (token masking), RTD (detection of substituted tokens)	Autoregressive prediction, infilling, instruction fine-tuning, long-context fine-tuning
Features	Balance between NL and PL, focus on semantic understanding	Predominantly code, emphasis on generation and interactivity, support for long contexts
Specialisation	General model without language-specific adaptation	Separate variants for Python with additional training

Source: compiled by the authors based on B. Gain *et al.* (2025)

The comparison shows substantial differences in the models' training strategies. The data scale for CodeLlama was much larger – from hundreds of billions to a trillion tokens – whereas CodeBERT was trained on 2-3 billion tokens. This provided CodeLlama with better generalisation and code-generation capabilities across diverse scenarios. CodeBERT demonstrated a balance between natural language and programming language, making this model more effective for code-search tasks based on natural-language queries. By contrast, CodeLlama was oriented predominantly towards deep understanding and code generation, which was reflected in the use of autoregressive training and additional fine-tuning methods. CodeLlama also showed greater flexibility thanks to support for long contexts and an "assistant" mode of operation, enabling complex programming tasks to be performed interactively. CodeBERT, meanwhile, was more directed towards contextual understanding and the formation of high-quality code representations.

Capabilities and tasks of the CodeBERT and CodeLlama models. The different architectures and training strategies of the CodeBERT and CodeLlama models determined the different roles in the field of "code intelligence". CodeBERT was created as a bimodal model that combines the understanding of natural language and programming language. The main goal of CodeBERT is to form high-quality joint representations for text and code, which makes it possible to perform effectively tasks related to code search using natural-language queries, automatic documentation generation, and the study of links between natural language and code. The model showed high results in code-clone detection, defect and vulnerability identification, as well as in correcting simple errors in code. In general, CodeBERT is directed towards universal understanding and analysis of existing NL-PL artefacts. By contrast, CodeLlama, thanks to its generative decoder architecture and the significant volume of training data, was oriented towards a wide range of tasks for creating and modifying code. It was able to generate code fragments from a description, complete code in real time, and also perform the filling of missing parts of code between a given context. The model also helped with debugging, code explanation and the execution of natural-language instructions, which made it a powerful tool for interactive developer support. Of particular note is CodeLlama's ability to process very large contexts, which is a key advantage when working with large codebases (Ghaemi *et al.*, 2024).

Comparative analysis shows that CodeBERT was oriented towards tasks requiring a deep understanding of the relationship between natural language and code, in particular search, documentation and analytics. CodeLlama was more effective in generative tasks related to creating, supplementing and modifying code based on natural-language context. This difference followed from the basic architectural decisions and training approaches of the models: CodeBERT, as an encoder-bimodal model, focuses on understanding and representations, whereas CodeLlama, being a generative decoder, specialises in autoregressive generation, infilling and working with instructions. Therefore, the choice between these models should be based on the specifics of the particular tasks and the functional requirements. Below is a comparison of the main functional features of the CodeBERT and CodeLlama models, showing the differences in architecture, training approaches and task types (Table 4).

Table 4. Functional features of the training approaches of the CodeBERT and CodeLlama models

Characteristic	CodeBERT	CodeLlama
Primary focus	Understanding natural language and code	Generation and modification of code
Task type	Search, classification, analysis	Generation, autocompletion, instructions
Architecture	Encoder-bimodal	Generative decoder
Training type	Token masking (MLM), detection of substituted tokens (RTD)	Autoregressive training, infilling, instructions
Support for long contexts	Limited (up to 512 tokens)	High (up to 100 thousand tokens)
Application	Understanding NL-PL semantics, analytics	Code creation, interactivity, developer assistance

Source: compiled by the authors based on A. Tehrani *et al.* (2024)

Analysing the presented data, it can be noted that the main difference between the models lay in the orientation and architectural base. CodeBERT, as an encoder-bimodal model, was created for deep understanding and alignment of natural language with code, which makes it optimal for tasks related to search, classification, and defect detection in code. Its limited support for long contexts imposed certain constraints on working with large files or projects. By contrast, CodeLlama, based on a generative-decoder architecture and trained on huge volumes of data, specialised in creating and modifying code. It successfully coped with autocompletion tasks, the generation of new fragments, and also executed complex instructions, which made it an indispensable tool in the development process. Thus, the choice between CodeBERT and CodeLlama should be based on the user's specific needs: for understanding, analysis and search tasks – CodeBERT is better suited, and for generation and active development support – CodeLlama.

Performance results on standard benchmarks. Evaluation of the performance of the CodeBERT and CodeLlama models on standard benchmarks was a key stage for understanding the strengths and weaknesses, as well as for determining the sphere of the effective application. It is worth noting that significant changes in the evaluation landscape took place between the release moments of these models, which affected the comparison of the results. CodeBERT, released earlier, set new standards in several important NL-PL tasks. In particular, on the CodeSearchNet benchmark it achieved state-of-the-art (SOTA) performance in code search by natural-language queries (by the MRR metric) and documentation generation (BLEU-4) for six programming languages. Within CodeXGLUE the model showed high results in code-clone detection (F1 = 94.1, MAP = 82.67), as well as in defect classification (accuracy 62.08%). It also showed decent results in code translation between Java and C# (CodeBLEU 85.10/79.41) and confirmed superiority over the RoBERTa model in specialised NL-PL probing tasks. At the same time, CodeBERT is not oriented towards the generation of functional code, and to participate in such tasks it should be integrated into an encoder-decoder architecture.

By contrast, CodeLlama, thanks to its generative architecture and training scale, set new records among open models in code generation. On the HumanEval and MBPP benchmarks, the model demonstrated significantly better performance: pass@1 reached 67.8% in the 70B-Instruct version and 65.6% in the 70B-Python version. Even smaller variants, for example Python 7B, showed performance surpassing large models such as Llama 2 70B. CodeLlama also exhibited strong results in multilingual code generation on the MultiPL-E benchmark, outperforming models such as StarCoder and CodeGen-Multi. For description generation in CodeXGLUE the model demonstrated results close to the industry leaders (BLEU about 20-21), and also showed high scores in algorithmic programming tasks, code-security assessment and other complex tasks (Shi *et al.*, 2024). Direct comparison of the models confirmed the different specialisation: CodeLlama prevailed in generative tasks (HumanEval, MBPP), while CodeBERT performed better in understanding tasks such as code search or the detection of clones and defects. The results for code translation (CodeBLEU) demonstrated the advantage of CodeBERT, while in code search it also retained leadership (Table 5). The evaluation of CodeLlama in clone or defect detection tasks was still insufficiently represented in public sources.

Table 5. Comparison of the performance of the CodeBERT and CodeLlama models on standard benchmarks

Benchmark	Task	Metric	CodeBERT (125M)	CodeLlama (variant/size)
HumanEval	Python-code generation	pass@1	data absent or the model was not officially tested	Up to 67.8% (70B-Instruct)
MBPP	Python-code generation	pass@1	data absent or the model was not officially tested	Up to 65.6% (70B-Python)
MultiPL-E	Multilingual code generation	pass@1	data absent or the model was not officially tested	SOTA among open models

Continued Table 5.

Benchmark	Task	Metric	CodeBERT (125M)	CodeLlama (variant/size)
CodeXGLUE (BigCloneBench)	Clone detection	F1	94.1	data absent or the model was not officially tested
CodeXGLUE (Devign)	Defect detection	Accuracy	62.08	data absent or the model was not officially tested
CodeXGLUE (CodeSearchNet)	Description generation (Python)	BLEU-4	SOTA at the time of release	~20.4 (7B), ~21.1 (13B)
CodeXGLUE (CodeTrans Java→C#)	Code translation	CodeBLEU	85.10	N/A (used in RAG studies)
CodeXGLUE (CodeSearchNet)	Code search NL→PL	MRR	~0.724	data absent or the model was not officially tested

Notes: RAG – retrieval-augmented generation
Source: compiled by the authors based on J. Shi *et al.* (2024), Z. Su *et al.* (2024)

The table shows that CodeBERT and CodeLlama differed in task spectrum and results. CodeBERT was a powerful model for code-understanding tasks, in particular search, defect and clone classification, as well as code translation, which corresponded to its encoder architecture and bimodal training. Its results on the corresponding benchmarks remained competitive even after the appearance of newer models. By contrast, CodeLlama significantly prevailed in generative tasks, especially in creating code from a description, autocompletion, and instruction following. Its outstanding scores on HumanEval and MBPP confirmed high efficiency in Python-code generation tasks, where CodeBERT is not applied. CodeLlama also demonstrated excellent results in multilingual code generation, which makes it a versatile tool for developers with different language stacks. The absence of public results for CodeLlama in classical clone and defect-detection tasks complicates a full comparison, but the existing data underline that these models are oriented towards different aspects of "code intelligence". CodeBERT specialises in analysis and understanding, whereas CodeLlama – in functional generation and working with long code contexts. In addition, the difference in the years of the models' release and the respective evaluations is important for interpreting the results: CodeBERT was evaluated on tasks relevant in 2020, while CodeLlama – on newer generative benchmarks that reflect current challenges and needs in development. It is worth noting that for CodeBERT (125 M) there are no official data on performance on the HumanEval, MBPP and MultiPL-E benchmarks, since the model was not tested on these tasks at the time of publication. This is explained by the fact that CodeBERT was originally developed mainly for code-understanding, search, classification and analytics tasks, rather than for Python-code generation or multilingual generation. Therefore, direct comparison with CodeLlama in these generative tasks is impossible, and the results show the specialisation of the models for different types of tasks: CodeBERT – for analysis and understanding of code, CodeLlama – for generation and working with long contexts.

Analysis of competitive advantages and challenges of model application. A deeper analysis of the CodeBERT and CodeLlama models makes it possible to identify the key advantages and disadvantages, which were determined by differences in architecture, scale, training data and period of creation. CodeBERT has clear advantages in specialised bimodal understanding, since this model was designed for the efficient modelling of the semantic link between natural language and programming language. Due to to bimodal training and the Replaced Token Detection (RTD) objective, it shows good results in tasks that require such matching, and also demonstrates efficiency in forming robust code representations (Zhang *et al.*, 2025). This makes it a powerful tool for code understanding tasks, for example in clone or defect detection. In addition, CodeBERT has become an important starting point for further research and the development of more complex models, such as GraphCodeBERT, which take code structure into account. An important advantage is the model's accessibility – it has 125 million parameters, is distributed under the Massachusetts Institute of Technology (MIT) licence and is relatively easy to deploy and fine-tune even on standard hardware. At the same time, CodeBERT has a number of limitations. The model is outdated by modern standards, as it was developed in 2020, and it has a rather small size of 125 million parameters, which limits its ability for complex reasoning or generation. As a RoBERTa-based encoder, it has limited code-generation capabilities and requires additional architectural solutions for extended generative functionality. The context window is limited to 512 tokens, which does not allow it to work effectively with long code files or large projects. The model also has limited language diversity, having been trained on only six programming languages, which is inferior to more large-scale modern models.

By contrast, CodeLlama is distinguished by high code-generation quality and is a leader among open models on the HumanEval and MBPP benchmarks. The model is specially optimised for code autocompletion, enabling it to add fragments intelligently into an existing context, which is particularly useful when IDEs. CodeLlama is presented in several scales – from 7 to 70 billion parameters – allowing users to balance performance and computational cost. It supports the processing of very long contexts up to 100,000 tokens, which expands the scope of application from working with large codebases to complex analysis and debugging. Instruction-tuned variants provide an intuitive interface through natural language, enabling zero-shot programming and making deployment safer. Special attention is paid to specialisation in Python, where the model shows even better performance. Importantly, CodeLlama is an open model, and its code and weights are available under a community licence, which promotes academic research and commercial use (Yong *et al.*, 2025). However, CodeLlama also has its drawbacks. Like all large LLMs, it may generate inaccurate, biased or undesirable content, which requires careful monitoring during use. Large variants of the model require significant hardware resources, including complex distributed computing for inference and fine-tuning. The Llama 2 Community licence provides restrictions for organisations with more than 700 million active users, which creates barriers to mass commercial use. The model is also less oriented towards bimodal tasks, therefore its effectiveness in semantically matching natural language with code is inferior to CodeBERT, although larger scale can partially compensate for this difference.

Therefore, the analysis of these models indicates the presence of a trade-off between specialisation and versatility. CodeBERT is a highly specialised model, optimised for deep understanding of the relationship between natural language and code, which provides high performance in narrowly specialised tasks. At the same time, its limited scale and architectural features significantly narrow the scope of application. CodeLlama, in turn, thanks to powerful scaling, support for autocompletion and long contexts, as well as a variety of variants, is a more versatile tool for generating and analysing code in many programming languages. The factor of time and technological progress played a key role: CodeLlama is a newer, technically more advanced model with functionality that was unavailable at the time CodeBERT appeared. At the same time, the drawbacks of CodeLlama related to hardware requirements, safety and licensing are characteristic of modern large models. At the time of its release, CodeLlama was the state-of-the-art open generative code model, whereas CodeBERT retained its value as an accessible and efficient solution for specialised code-understanding tasks (Ghaemi *et al.*, 2024).

The different strengths and weaknesses of the CodeBERT and CodeLlama models determine the optimal areas of application within the software development life cycle. CodeBERT is particularly effective in tasks that require a deep understanding of the relationship between natural language and code, such as semantic code search, where it helps to create systems that allow developers to find relevant examples or functions using natural-language queries, thereby improving code search and reuse. It is also useful for automated code documentation, generating docstrings, comments or summaries, which improves the readability and maintainability of projects. CodeBERT is effective in detecting code clones – duplicated or similar blocks – which helps with refactoring and maintaining consistency, and it is also used for defect and vulnerability detection, becoming a key component of tools that analyse code for potential security issues, especially after additional fine-tuning on relevant datasets. In addition, it provides reliable code representations that underpin program analysis and various static-analysis tasks that require an understanding of code semantics.

In contrast, CodeLlama stands out for powerful generative capabilities and extended features, which makes it useful for code generation and completion. Acting as an AI programming partner, it generates boilerplate code, implements functions based on descriptions and provides autocompletion hints in development environments. CodeLlama supports code infilling – seamless insertion of fragments into already existing files – which is useful for completing function bodies or filling templates. Instruction-tuned variants allow interaction with developers through natural-language commands to perform tasks such as refactoring, optimisation, unit-test generation or code explanation. The model also assists in debugging, helping to detect and eliminate errors, explain complex sections and suggest fixes. CodeLlama is useful as an educational tool, helping programming newcomers by generating examples, explanations and help with exercises. It is effective when working with large codebases, using the long context window for analysis, refactoring and code generation that requires understanding of relationships between large parts of a project. It is particularly worth highlighting the specialised CodeLlama-Python versions, which are optimised for high performance and accuracy in Python projects (Shi *et al.*, 2024).

Thus, CodeBERT and CodeLlama complement each other rather than being complete substitutes. CodeBERT is most often useful before writing code or after it has been created, performing functions of searching for existing solutions, documentation and analysis, focusing on the understanding and analysis of already existing artefacts. CodeLlama, meanwhile, is a valuable assistant during active coding – it helps to generate, supplement, modify code and debug it, acting as an interactive assistant similar to GitHub Copilot. An ideal AI software-development environment could include both types of models: CodeBERT for search and static analysis, and CodeLlama for interactive generation and code support.

Infrastructure and licensing conditions for model deployment. Against the backdrop of rapidly developing AI, model availability is one of the key factors that determine the practical value, uptake in academic and industrial environments, and the pace of integration into applied solutions. This aspect covers not only the physical availability of models in open repositories, but also the licensing conditions, technical requirements for the deployment, and the barriers that may arise due to legal or infrastructural constraints. In this context, the CodeBERT and CodeLlama models demonstrate two different approaches to openness, usage regulation and technical accessibility, which leads to significant differences in the application.

CodeBERT, developed by researchers at Microsoft Research, is an example of high accessibility from both a technical and legal perspective. The model, together with its pre-trained weights (including the versions microsoft/codebert-base and microsoft/codebert-base-mlm), is freely hosted on the GitHub and Hugging Face Hub platforms, which greatly simplifies downloading, use and integration into various systems. Moreover, openness extends to subsequent developments, such as GraphCodeBERT, which demonstrates consistency in the open-access policy. CodeBERT is distributed under the MIT licence – one of the most liberal open licences – which allows any use, including commercial, as well as modification and redistribution, provided that the relevant copyright notices are preserved. This creates a favourable legal environment for its deployment in both academic research and business products, imposing virtually no restrictions on the end user (Yong *et al.*, 2025). From a technical point of view, CodeBERT is built on the RoBERTa architecture with around 125 million parameters, which makes it relatively compact among modern transformer models. This ensures efficient use of computing resources and allows both inference and fine-tuning even on standard consumer hardware, including a single mid-range GPU. Compatibility with popular libraries such as Hugging Face Transformers further simplifies integration into existing development environments and facilitates rapid deployment across infrastructures of varying scale.

By contrast, CodeLlama, created by Meta, although positioned as an open tool, in practice has more complex legal and technical conditions of use. The model is distributed under the Llama 2 Community Licence, which, although allowing research and commercial use, includes a number of significant restrictions. In particular, companies with a large user base (more than 700 million monthly active users) must enter into a separate commercial agreement with Meta, which significantly complicates its use in large-scale industrial products. In addition, the licence contains an acceptable-use policy that imposes additional regulatory requirements on user conduct, as well as restrictions on the use of the model or its outputs to train or improve other large language models, apart from the Llama 2 family. Thus, Meta's licensing model entails a certain legal complexity that requires careful analysis by companies, especially when implementing long-term or large-scale solutions.

From the perspective of technical accessibility, CodeLlama is notable for high scale and corresponding resource requirements. Models in this series, from 7B to 70B parameters, are available for download, including via the Hugging Face Hub, but the effective deployment varies according to size. The smallest model (7B) can still be run on a single modern GPU, but larger modifications – especially 34B and 70B – require serious computing infrastructure, including multi-GPU configurations, model parallelism, and in some cases distributed computing. Such technical complexity not only increases implementation costs but also reduces the model's accessibility for a wide range of researchers and companies that do not have the appropriate hardware base. Although the official inference code is also hosted on GitHub and supports integration with standard libraries, practical deployment remains significantly more complex compared with CodeBERT (Bhandari *et al.*, 2025).

Overall, the existing differences in the accessibility of CodeBERT and CodeLlama indicate two contrasting strategies for AI openness. The former model demonstrates an example of maximum flexibility, simplicity and legal neutrality, which promotes wide engagement of users from different environments. The latter model, although providing exceptionally powerful capabilities, simultaneously requires a high level of responsibility from the user, resource provision and readiness to comply with regulatory frameworks. This difference is important when choosing a model, as it directly affects its applicability in a particular research or commercial context.

DISCUSSION

Analysis of the latest scientific publications shows active development of methods for applying LLMs to program analysis tasks, in particular vulnerability detection, automated assessment, code summarisation and transformation of code structures. Such works demonstrate a shift of emphasis from classical heuristic approaches to the use of transformers, adapters, federated learning and hybrid context analysis. This underlines the relevance and value of the present study, which also evaluates the performance of modern LLMs, in particular CodeBERT and CodeLlama, in a code context, including architectural and applied aspects.

For example, in the article by Z. Su *et al.* (2024), the Codeart technique was proposed, which improves the performance of code models under conditions of missing semantic symbols such as variable names or comments. The authors used an attention regularisation mechanism that helps the model to identify structural patterns better, even in simplified or obfuscated code. Compared with the present study, this work is likewise aimed at increasing model reliability in a

"non-standard" environment; however, it focuses less on automated vulnerability detection or patch generation and instead emphasises improvements in model training. In the study by Y. Zhang *et al.* (2025), the MMF-Detect method was developed, based on a multimodal combination of features for detecting WebShell scripts that evade traditional detection methods. The model combines different types of features – from structural to semantic – and uses deep architectures to create fused vector representations. Compared with the present study, this work demonstrates an alternative approach to threat detection that can be integrated into LLM systems, which was also proposed as a promising direction – combining LLMs with detectors based on fusion representations to increase accuracy. In the publication by C. Yong *et al.* (2025), a smart-contract generation model is considered that combines code annotations with AST trees and a modified LSTM architecture. The authors emphasise the importance of using semi-structured description and syntactic parsing to improve the safety of automatically generated code. Compared with the present study, this work integrates syntactic structures more deeply into the generation process, indicating potential synergy between AST mechanisms and LLMs, which is considered promising for improving the accuracy of safe code generation in vulnerability-fixing tasks.

In the work of K. Mohamed *et al.* (2024), the effectiveness of using LLMs for automated assessment of programming tasks is considered. The authors focused on the practical application of models in learning systems, finding high accuracy and flexibility of such models. The approach was based on comparing manual and automated assessment, which made it possible to identify limitations in the generalisability of solutions. By contrast, the present study is focused not on educational systems but on the technical features of code generation and understanding in engineering tasks. At the same time, the study by W. Luo *et al.* (2025) concentrates on privacy issues in training LLMs for software code fixing, in particular through federated learning approaches. The authors found that local training preserves data privacy without a significant loss of model quality. In the context of the present study, these results are important as an example of a trade-off between performance and security – an aspect that was partially considered in the current comparison of models.

Z. Zhou *et al.* (2024) proposed an approach to generating structured textual descriptions of code based on hybrid context. The method was based on integrating lexical, syntactic and semantic signals, which made it possible to create more accurate explanations of functions. This significantly improves the model's capabilities in documentation tasks. In the present study, models capable of similar tasks are considered, but the main attention is paid to the overall architecture and multifunctionality. I. Saberi *et al.* (2025) presented the AdvFusion method, which uses adapters for transfer learning in code summarisation tasks. The authors achieved improvements without the need for full model retraining, reducing computational costs. This approach is relevant to the current analysis of adaptive LLMs, particularly in comparison with CodeBERT. The present study also recorded the advantages of architectures that allow modular adaptation to specific tasks. In turn, Z. Qin *et al.* (2025) developed the CLNX model for detecting vulnerable commits in C/C++ code, combining code analysis and natural language. The approach demonstrates an effective combination of structural analysis with textual representations. This is consistent with the present observation that models that take into account context and meta-information show higher accuracy in security tasks. However, the analysis conducted here was broader and covered model architectures as a whole.

S.M. Taghavi Far & F. Feyzi (2025) carried out a thorough review of models for detecting software vulnerabilities, classifying existing techniques, datasets, and metrics. The researchers pointed to the limitations of existing LLMs in the field of explainability and overall consistency of results. This confirms some results of this work regarding the limitations of CodeLlama in high-precision vulnerability-analysis tasks. However, in this case, a deeper comparison of models on practical tests was made. S. Shimmi *et al.* (2024) presented the VulSim method, based on multidimensional vector similarities, for detecting vulnerabilities in code. The authors used embeddings of neighbouring code elements as the basis for similarity, which is innovative for semantic analysis. This approach shows a step towards interpretable analysis, but requires powerful computational resources. In the present study, similar architectures were evaluated in the context of efficiency on standard benchmarks. B. Xiang & Y. Shao (2024) investigated the effectiveness of SUMLLAMA models in generating summaries from bug reports using contrastive learning. The results indicate high accuracy in summarising bug content thanks to specialised adapters. In comparison, the present study focuses on the universal capabilities of models such as CodeBERT for multiple tasks. F. Panebianco *et al.* (2025) critically assessed the ability of LLMs to detect vulnerabilities, describing LLMs as "not yet ready" for productive use in security scenarios. The authors pointed to a high level of "guessing" and instability of responses. These results are consistent with current observations regarding insufficient determinism in CodeLlama's outputs, especially in zero-shot scenarios. This highlights the need for additional training or post-processing in security systems. The study by D. de-Fitero-Dominguez *et al.* (2024) is devoted to improving automatic debugging of vulnerable code using large language models. The authors presented new fine-tuning methods and adaptive mechanisms that increase the accuracy of patch generation. Compared with the present work, which investigates the performance of various LLMs in vulnerability detection and classification tasks, this

paper offers a more practical aspect of application – automatic fixing – thereby complementing the conclusions of the present study and showing directions for further development.

In the publication by M. Zhong *et al.* (2024), the universal ComBack dataset was presented, designed to improve the development efficiency of back-end compilers. The authors emphasised the complexity and diversity of tasks that the set can cover, allowing different aspects of compilation optimisation to be modelled. Compared with the present study, this approach underscores the importance of high-quality datasets as a foundation for training LLMs and creating reliable tools for code analysis and repair. The work of A. Singhal *et al.* (2025) is devoted to using LLMs in a zero-shot mode to extract code characteristics in JSON format to support RAG systems. The authors showed that large models can effectively extract structured features without additional training. This coincides with the present approach, which evaluates the potential of LLMs for rapid and accurate code analysis without lengthy fine-tuning. In the article by O. Çaylı (2024) the use of generative AI for preventive measures and protection against cyber threats, particularly in the field of security vulnerabilities, is considered. The author emphasises the promise of applying generative models for active attack prevention and automated response. Compared with the present study, this work broadens the context of LLM use with a focus on cybersecurity, confirming the relevance and practical significance of implementing such technologies.

In summary, previous studies were mostly applied or review-oriented and focused on individual aspects of using LLMs with code: generation, debugging or security. The aforementioned studies provide valuable insights into the capabilities of LLMs in specific scenarios, but do not offer a holistic comparison of models of different architectural types across a range of criteria, including architecture, data, metrics and licences among others. Thus, the present study fills this gap by providing a systematised analytical overview of CodeBERT and CodeLlama as representatives of different approaches to processing code using LLMs.

CONCLUSIONS

In the course of the comparative analysis of the CodeBERT and CodeLlama models, it was found that each represents different generations and concepts of language models for code. CodeBERT, as a model built on the RoBERTa transformer, is oriented mainly towards code understanding tasks such as classification, search, query-code matching and other kinds of static analysis. Its performance is stable in a range of standard tests, and it provides fast training and inference even on limited resources. Its MIT licence creates maximally open conditions for use in any environment, including commercial purposes. By contrast, CodeLlama is an example of modern large language models built according to the principles of the LLaMA 2 architecture. It provides high-quality code generation, supports long contexts, and shows better results in tasks of auto-completion, instruction-based generation and multimodal scenarios. However, the model requires significant hardware resources, especially in the 34B and 70B configurations, and has a number of restrictions defined by the Llama 2 Community Licence, particularly for large companies. This limits its large-scale deployment in high-load products without additional legal interaction with Meta.

To summarise, both models have the unique advantages and are suitable for different purposes: CodeBERT is suitable for code analysis, research purposes and educational tasks in resource-constrained environments, whereas CodeLlama is more oriented towards generation and complex software tasks at industrial scale. The choice between these models should be based on the nature of the tasks, available infrastructure capabilities, licensing requirements, and requirements for accuracy or performance. A limitation of this study is the focus exclusively on the open models CodeBERT and CodeLlama, without an in-depth analysis of closed or commercial solutions such as Copilot (GitHub), Gemini Code Assist (Google) or Amazon CodeWhisperer. Also, the work did not consider empirical testing of the models in real development environments, which limits the practical assessment of performance. Future research in the field of code generation using large language models should focus on improving contextual understanding, interpretability, the ability for multistep reasoning, integration into the full software development life cycle, as well as support for multimodal inputs and domain adaptation. Topical issues remain the ethical use, security, quality control of generated code and licensing constraints. Further development of models like CodeLlama and CodeBERT presupposes not only an increase in power but also the creation of more transparent, adaptive and responsible solutions for the professional software development environment.

ACKNOWLEDGEMENTS

None.

FUNDING

None.

CONFLICT OF INTEREST

None.

REFERENCES

[1] 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.

[2] Bhandari, G., Gavric, N., & Shalaginov, A. (2025). Generating vulnerability security fixes with code language models. *Information and Software Technology*, 185, article number 107786. doi: 10.1016/j.infsof.2025.107786.
[3] Budzynskyi, O.V. (2025). Method of detecting vulnerabilities and automated response in corporate database protection systems. *Modern Information Security*, 2(62), 180-186. doi: 10.31673/2409-7292.2025.029259.
[4] Çaylı, O. (2024). AI-enhanced cybersecurity vulnerability-based prevention, defense, and mitigation using generative AI. *Orclever Proceedings of Research and Development*, 5(1), 655-667. doi: 10.56038/oprd.v5i1.616.
[5] d'Aloisio, G., Traini, L., Sarro, F., & Di Marco, A. (2025). On the compression of language models for code: An empirical study on codeBERT. In *2025 IEEE international conference on software analysis, evolution and reengineering (SANER)* (pp. 12-23). Montreal: IEEE. doi: 10.1109/SANER64311.2025.00010.
[6] de-Fitero-Dominguez, D., Garcia-Lopez, E., Garcia-Cabot, A., & Martinez-Herraiz, J.-J. (2024). Enhanced automated code vulnerability repair using large language models. *Engineering Applications of Artificial Intelligence*, 138(A), article number 109291. doi: 10.1016/j.engappai.2024.109291.
[7] Deineha, O., Donets, V., & Zholtkevych, G. (2024). The approach development of data extraction from lambda terms. *Eastern-European Journal of Enterprise Technologies*, 3(2(129)), 42-54. doi: 10.15587/1729-4061.2024.298991.
[8] Gain, B., Bandyopadhyay, D., Mukherjee, S., Sahoo, A., Dana, S., Kodeswaran, P., Sen, S., Ekbal, A., & Garg, D. (2025). Transforming code understanding: Clustering-based retrieval for improved summarization in domain-specific languages. In O. Rambow, L. Wanner, M. Apidianaki, H. Al-Khalifa, B. Di Eugenio, S. Schockaert, K. Darwish & A. Agarwal (Eds.), *Proceedings of the 31st international conference on computational linguistics: Industry track* (pp. 546-560). Abu Dhabi: Association for Computational Linguistics.
[9] Gao, Z., Wang, H., Wang, Y., & Zhang, C. (2024). Virtual compiler is all you need for assembly code search. In *Proceedings of the 62nd annual meeting of the association for computational linguistics* (pp. 3040-3051). Bangkok: Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.167.
[10] Ghaemi, H., Alizadehsani, Z., Shahraki, A., & Corchado, J.M. (2024). Transformers in source code generation: A comprehensive survey. *Journal of Systems Architecture*, 153, article number 103193. doi: 10.1016/j.sysarc.2024.103193.
[11] Gurjar, A., Camp, L.J., Ringenberg, T., Ma, X., & Chaora, A. (2023). Can large language models detect PII in code? *SSRN*. doi: 10.2139/ssrn.4619112.
[12] Hodovychenko, M.A., & Kurinko, D.D. (2025). Analysis of existing approaches to automated refactoring of object-oriented software systems. *Herald of Advanced Information Technology*, 8(2), 179-196. doi: 10.15276/hait.08.2025.11.
[13] Huang, K., Zhang, J., Bao, X., Wang, X., & Liu, Y. (2025). Comprehensive fine-tuning large language models of code for automated program repair. *IEEE Transactions on Software Engineering*, 51(4), 904-928. doi: 10.1109/tse.2025.3532759.
[14] Li, J., Tao, C., Li, J., Li, G., Jin, Z., Zhang, H., Fang, Z., & Liu, F. (2023). Large language model-aware in-context learning for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(7), article number 190. doi: 10.1145/3715908.
[15] Luo, W., Keung, J., Yang, B., Ye, H., Goues, C.L., Bissyandé, T.F., Tian, H., & Le, X.B.D. (2025). When fine-tuning LLMs meets data privacy: An empirical study of federated learning in LLM-based program repair. *ACM Transactions on Software Engineering and Methodology*. doi: 10.1145/3733599.
[16] 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.
[17] Panebianco, F., Isgro, A., Longari, S., Zanero, S., & Carminati, M. (2025). Guessing as a service: Large language models are not yet ready for vulnerability detection. In *Proceedings of the joint national conference on cybersecurity (ITASEC & SERICS 2025)* (pp. 1-17). Bologna: Security and Rights in CyberSpace Foundation.
[18] Qin, Z., Wu, Y., & Han, L. (2025). CLNX: Bridging code and natural language for C/C++ vulnerability-contributing commits identification. In *Proceedings of the AAAI conference on artificial intelligence* (pp. 25047-25055). Washington: AAAI Press. doi: 10.1609/aaai.v39i23.34689.
[19] Raihan, N., Newman, C., & Zampieri, M. (2024). Code LLMs: A taxonomy-based survey. In *2024 IEEE international conference on Big Data (BigData)* (pp. 5402-5411). Washington: IEEE. doi: 10.1109/BigData62323.2024.10826108.
[20] Saberi, I., Esmaeili, A., Fard, F., & Chen, F. (2025). AdvFusion: Adapter-based knowledge transfer for code summarization on code language models. In *2025 IEEE international conference on software analysis, evolution and reengineering (SANER)* (pp. 563-574). Montreal: IEEE. doi: 10.1109/SANER64311.2025.00059.
[21] Shi, J., Yang, Z., Kang, H.J., Xu, B., He, J., & Lo, D. (2024). Greening large language models of code. In *ICSE-SEIS'24: Proceedings of the 46th international conference on software engineering: Software engineering in society* (pp. 142-153). New York: Association for Computing Machinery. doi: 10.1145/3639475.3640097.

[22] Shimmi, S., Rahman, A., Gadde, M., Okhravi, H., & Rahimi, M. (2024). VulSim: Leveraging similarity of multi-dimensional neighbor embeddings for vulnerability detection. In *33rd USENIX security symposium (USENIX Security 24)* (pp. 1777-1794). Philadelphia: Curran Associates, Inc.
[23] Siavvas, M., Kalouptsoglou, I., Gelenbe, E., Kehagias, D., & Tzovaras, D. (2024). Transforming the field of vulnerability prediction: Are large language models the key? In *2024 32nd international conference on modeling, analysis and simulation of computer and telecommunication systems (MASCOTS)* (pp. 1-6). Krakow: IEEE. doi: 10.1109/MASCOTS64422.2024.10786575.
[24] Singhal, A., Ghosh, R., Mundra, R., Dadlani, H., & Dutta, D. (2025). Code2JSON: Can a zero-shot LLM extract code features for code RAG? In *ICLR 2025 third workshop on deep learning for code* (pp. 1-23). Singapore: International Conference on Learning Representations.
[25] Su, Z., Xu, X., Huang, Z., Zhang, Z., Ye, Y., Huang, J., & Zhang, X. (2024). Codeart: Better code models by attention regularization when symbols are lacking. *Proceedings of the ACM on Software Engineering*, 1, 562-585. doi: 10.1145/3643752.
[26] Taghavi Far, S.M., & Feyzi, F. (2025). Large language models for software vulnerability detection: A guide for researchers on models, methods, techniques, datasets, and metrics. *International Journal of Information Security*, 24, article number 78. doi: 10.1007/s10207-025-00992-7.
[27] Tehrani, A., Bhattacharjee, A., Chen, L., Ahmed, N.K., Yazdanbakhsh, A., & Jannesari, A. (2024). CodeRosetta: Pushing the boundaries of unsupervised code translation for parallel programming. In *38th conference on neural information processing systems* (pp. 100965-100999). Vancouver: Neural Information Processing Systems Foundation, Inc.
[28] Weyssow, M. (2024). *Aligning language models to code: Exploring efficient, temporal, and preference alignment for code generation*. (PhD dissertation, University of Montreal, Montreal, Canada).
[29] Xiang, B., & Shao, Y. (2024). SUMLLAMA: Efficient contrastive representations and fine-tuned adapters for bug report summarization. *IEEE Access*, 12, 78562-78571. doi: 10.1109/access.2024.3397326.
[30] Yong, C., Defeng, H., Chao, X., Nannan, C., & Jianbo, L. (2025). Smart contract generation model based on code annotation and AST-LSTM tuning. *Journal of Supercomputing*, 81, article number 731. doi: 10.1007/s11227-025-07186-x.
[31] Zhang, Y., Kang, H., & Wang, Q. (2025). MMFDetect: Webshell evasion detect method based on multimodal feature fusion. *Electronics*, 14(3), article number 416. doi: 10.3390/electronics14030416.
[32] Zheng, Z., Ning, K., Wang, Y., Zhang, J., Zheng, D., Ye, M., & Chen, J. (2023). A survey of large language models for code: Evolution, benchmarking, and future trends. *ArXiv*. doi: 10.48550/arXiv.2311.10372.
[33] Zhong, M., Lyu, F., Wang, L., Geng, H., Qiu, L., Cui, H., & Feng, X. (2024). ComBack: A versatile dataset for enhancing compiler backend development efficiency. In *38th conference on neural information processing systems* (pp. 112310-112328). Vancouver: Neural Information Processing Systems Foundation, Inc.
[34] Zhou, Z., Li, M., Yu, H., Fan, G., Yang, P., & Huang, Z. (2024). Learning to generate structured code summaries from hybrid code context. *IEEE Transactions on Software Engineering*, 50(10), 2512-2528. doi: 10.1109/tse.2024.3439562.

Порівняльний аналіз моделей CodeBERT та CodeLlama: архітектура, функціональність та застосування в задачах програмного кодування

Олександр Дейнега

Доктор філософії з комп'ютерних наук, викладач
Харківський національний університет імені В.Н. Каразіна
61022, пл. 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, пл. 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, пл. Свободи, 4, м. Харків, Україна
https://orcid.org/0000-0003-0693-4122

Анотація. Актуальність дослідження зумовлена потребою порівняти великі мовні моделі CodeBERT і CodeLlama, які активно використовують для автоматизації генерації та аналізу коду з метою підвищення ефективності й якості програмного забезпечення. Метою дослідження було всебічне зіставлення архітектурних, функціональних характеристик обраних мовних моделей CodeBERT і CodeLlama. Використано інтерпретативний, порівняльний, системний та структурно-категоріальний аналізи для вивчення архітектур, завдань та релевантності моделей. Здійснено всебічний порівняльний аналіз моделей CodeBERT і CodeLlama за ключовими параметрами: архітектура моделей (архітектура енкодер RoBERTa у CodeBERT проти декодерної архітектури Llama 2 у CodeLlama), масштаб і джерела навчальних даних, спектр підтримуваних завдань, продуктивність на еталонних бенчмарках, переваги та обмеження, типові сфери застосування та умови доступності й ліцензування. Результати показали, що різниця в архітектурі та навчальних даних суттєво впливає на ефективність моделей у різних типах завдань, а також визначає їх практичні можливості й обмеження. Особливу увагу приділено питанням впровадження моделей у практичні сценарії, з урахуванням апаратних ресурсів і ліцензійної політики. Результати показали, що CodeLlama потребує значно більших обчислювальних ресурсів для ефективної роботи, тоді як CodeBERT є більш легким у впровадженні на стандартному обладнанні. Також було встановлено, що ліцензійні умови CodeLlama є більш обмежувальними, що може ускладнити його використання у комерційних проєктах, на відміну від CodeBERT із відкритою ліцензією. Зроблено висновок, що ці моделі виконують переважно взаємодоповнювальні функції: CodeBERT є ефективним інструментом для задач розуміння коду, тоді як CodeLlama демонструє високі результати в задачах генерації. У висновках окреслено виклики й перспективи розвитку моделей нового покоління з мультизадачністю та мультимодальністю. Практична цінність – допомога розробникам і дослідникам у виборі оптимального інструменту з урахуванням технічних і ліцензійних аспектів

Ключові слова: великі мовні моделі; ентрансформерна архітектура; декодерна архітектура; системний та функціональний аналіз; оптимізаційна модель; статистичний аналіз; обробка природної мови