Turk. J. Math. Comput. Sci.
17(2)(2025) 519–526
© MatDer
DOI : 10.47000/tjmcs.1699086

Fine-Tuning LLaMA2, LLaMA3, and Phi3 Models for Code Generation with Turkish Instructions

Emir Öztürk[1,*], Aydin Carus[1]

[1]*Department of Computer Engineering, Faculty of Engineering, Trakya University, 22030 Edirne, Türkiye.*

Received: 14-05-2025 • Accepted: 30-06-2025

Abstract. With the advancement of artificial intelligence and large language models, various models have begun to be utilized at every stage of software development processes, which has led to changes in coding habits. Instead of writing entire pieces of code themselves, developers now leave certain patterns to language models or use these models to query the issues they encounter. Similarly, code translation from one programming language to another is also carried out with the help of language models. While such studies have been conducted in English, a model designed specifically for generating code in response to queries posed in Turkish does not yet exist. In this study, the "python_code_instructions_18k_alpaca" dataset, which includes data for translation, code generation from scratch, and error querying, was translated into Turkish, and different language models have been trained on this dataset. The performance of the trained language models has been evaluated using the ROUGE, BLEU and METEOR metrics and models that can generate code are presented.

2020 AMS Classification: 68T07, 68T05, 68N15

Keywords: Code generation, Llama, Phi3, Python.

1. Introduction

Software development stages are processes that require considerable time, even when the methods and algorithms to be implemented are clearly defined. Initially, an analysis is conducted in the software development process. Information obtained from this analysis is then used to create a design, followed by planning of coding procedures based on the created design [17]. In the coding phase, development is performed using a programming language suitable for the target platform.

With the diversification of digital systems, various platforms have emerged, each offering a growing set of features. Considering that each platform has its unique properties and usage scenarios, software products have become more complex, making their understanding and development increasingly difficult [3, 18].

The increasing complexity of software development processes has also influenced software development practices. These applications may contain millions of lines of code, written by large teams consisting of numerous developers, often using various programming languages [6]. When considering other tasks like coding, documentation, debugging, and testing, it becomes evident that the time required for these processes is growing steadily.

Therefore, many previous studies have focused on translating natural language into code [2, 7]. Additionally, tools providing answers within development environments have been created to assist developers during coding [5].

*Corresponding Author
Email addresses: emirozturk@trakya.edu.tr (E. Öztürk), aydinc@trakya.edu.tr (A. Carus)

With the widespread use of artificial intelligence models, their integration into software development fields has been achieved. Recently, deep learning models have been included in software project development phases, significantly improving efficiency, software quality, and ease of maintenance, particularly in areas such as automatic testing, code reviews, and requirements analysis [13]. Deep learning models are used in diverse areas, including automotive software [8], software test automation [11, 15], and software cost estimation [19]. Moreover, AI models are frequently employed for code development and code generation from given algorithms.

While models such as RNN and LSTM have been developed for algorithm-to-code or code-to-algorithm translation and transformer-based models [9] for finding errors in code or translating natural language, large language models (LLMs) have become popular today. Large language models can generate code by providing only key parts without detailed algorithm steps, due to their capabilities in language understanding, translation, and wide context window support. These models can also successfully perform tasks such as writing code from specified method steps, detecting errors in existing code, or translating code from one programming language to another.

Although popular tools like ChatGPT, Gemini, and similar service-based products can be used for these tasks, they have certain disadvantages compared to open-source models in terms of data security and because they are closed products with specific limitations.

Numerous open-source models have been developed and are available for academic use. Among the most well-known models are language models such as LLaMA [10, 20], Mistral [12], and Phi 3 [1]. These models can be trained as desired and independently utilized in a local environment.

Open-source language models are generally trained in English, and even multilingual models show limited performance in Turkish. To address this issue, these models need to be trained specifically on Turkish-language data.

In this study, selected models were trained to generate Python code from algorithm descriptions in Turkish, perform cross-language translation, and answer questions. The performance of these models was evaluated using ROUGE [14] and BLEU [16] metrics. Using the Turkish-adapted ”iamtarun/python_code_instructions_18k_alpaca” dataset, open-source LLaMA2, LLaMA3, and Phi3 language models were trained, and performance results were obtained. Additionally, to improve the responses to Turkish-language questions, Turkish LLaMA2, LLaMA3, and Phi3 models, initially trained on Turkish instruction datasets, were fine-tuned using the same dataset, and their performance results are also presented.

The second section of the study provides information about the language models used. The third section describes the metrics utilized. The fourth section presents the experimental results, and the final section discusses the obtained results and offers suggestions for future studies.

2. Models Used in the Study

2.1. **Llama2.**

Llama2 (Large Language Model Meta AI) is an open-source language model developed by Meta, based on the original Llama model. The Llama 2 family includes several versions ranging from 7 billion to 70 billion parameters, designed to fit different use cases and computational resources.

One key architectural improvement in Llama 2 is an extended context window—twice as long as in the original Llama—which lets the model handle longer input sequences more effectively. Llama 2-Chat is a fine-tuned variant of Llama 2, optimized specifically for dialogue applications. In addition, Llama 2 has been further refined with Reinforcement Learning from Human Feedback (RLHF). This iterative process gathers human preference data, trains reward models, and then updates the language model to better match those preferences. To boost performance in real-world conversational settings, techniques such as Proximal Policy Optimization (PPO) and rejection sampling fine-tuning are applied.

2.2. **Llama3.**

Llama 3 is the latest release in Meta's Llama series, offering an enhanced architecture and a larger parameter set than earlier versions. The main Llama 3 model has 150 billion parameters and supports multiple languages. In addition to this, the Llama 3 family includes smaller variants with 8 billion and 70 billion parameters to suit different application needs.

To improve runtime performance, Llama 3 incorporates optimizations in parallel processing and memory management. An integrated contentfiltering mechanism enforces policy requirements. The Llama 3 series is organized into three sub-releases: 3.1 for the largest models, 3.2 for the smaller models, and 3.3 for multimodal versions. In this work, we use the main 150 billion-parameter model from the 3.1 release.

2.3. **Phi3.** Phi3, developed by Microsoft, is designed to deliver high performance in a compact model, advancing natural language processing on resource-limited devices. The Phi3-4k version has 3.8 billion parameters and was trained on a dataset of 3.3 trillion tokens. Its primary goal is to run locally on devices with limited RAM and compute power—such as smartphones—thereby improving data security and response speed. Architecturally, Phi3-4k uses a Transformer decoder with a default context window of 4,000 tokens. A long-context variant, Phi3-128k, extends this window to 128,000 tokens, enabling the model to process much longer inputs without performance degradation. Phi3 shares the same tokenization scheme and block structure as Llama 2, ensuring compatibility with the wider Llama tool and application ecosystem.

In this study, we employ the Hugging Face models ”Llama2 Chat 7B,” “LLaMA3 Instruct 8B”, and ”Phi3 mini 4k Instruct”, along with three Turkish fine-tuned versions: ”Llama 2 7b chat hf Turkish”, ”llama3 Instruct Turkish” and ”Phi 3 mini 4k instruct Turkish”.

3. Metrics Used in Study

3.1. **BLEU.** BLEU is an automatic metric used to evaluate machine translation systems. It measures how well the machine output matches one or more human-produced reference translations. Introduced by Kishore Papineni et al. at IBM in 2002, BLEU was designed to offer a fast, inexpensive, and language-independent evaluation method in response to the limited availability and high cost of human judgments [16].

BLEU assesses translation quality by computing the overlap of n-grams between the candidate translation and the reference translations. Its success hinges on correlating this overlap closely with human preference judgments.

The BLEU score itself is calculated by counting the number of matching n-grams, regardless of their positions in the sentence (i.e., position-independent matches). The more matching n-grams found, the better the translation is considered. BLEU scores range from 0 to 1, with higher values indicating higher translation quality.

The BLEU score is computed using the following formula:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right),$$

where p_n is the modified precision for n-grams of size n, w_n is a weight (typically uniform), and BP is the brevity penalty:

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ \exp\left(1 - \frac{r}{c}\right) & \text{if } c \leq r \end{cases}.$$

Here, c is the length of the candidate translation and r is the length of the reference.

3.2. **ROUGE.** ROUGE is a set of metrics used to automatically evaluate the quality of summaries by comparing them to human-written reference summaries [14]. It was introduced by Chin-Yew Lin as a way to assess summarization systems efficiently and at scale. The main goal of ROUGE is to offer a measure that correlates well with human judgments of summary quality, while remaining fast, inexpensive, and repeatable. ROUGE includes several variants, each focusing on a different aspect of summary evaluation.

ROUGE-N measures the overlap of n-grams between a candidate summary and one or more reference summaries. It computes the recall of these n-grams, with ROUGE-1 (unigrams) and ROUGE-2 (bigrams) being the most commonly used forms. The recall-based ROUGE-N score is computed as:

$$\text{ROUGE-N} = \frac{\sum_{\text{gram}_n \in \text{Ref}} \text{Count}_{\text{match}}(\text{gram}_n)}{\sum_{\text{gram}_n \in \text{Ref}} \text{Count}(\text{gram}_n)}.$$

ROUGE-L is based on the longest common subsequence (LCS) between the candidate and reference summaries. The LCS is the longest sequence of words that appears in both texts in the same order, though not necessarily consecutively. This metric emphasizes the structural similarity between summaries. The ROUGE-L score is based on the F-measure:

$$\text{ROUGE-L} = \frac{(1+\beta^2) \cdot \text{R}_{\text{LCS}} \cdot \text{P}_{\text{LCS}}}{\text{R}_{\text{LCS}} + \beta^2 \cdot \text{P}_{\text{LCS}}},$$

where $\text{P}_{\text{LCS}} = \frac{\text{LCS}(X,Y)}{n}$, $\text{R}_{\text{LCS}} = \frac{\text{LCS}(X,Y)}{m}$, and β is usually set to 1.

ROUGE-W is a weighted variant of ROUGE-L that assigns higher scores to summaries preserving longer consecutive matches in the LCS. By giving more weight to contiguous word sequences, it ensures that key phrases and sentence structures have a greater impact on the score.

3.3. **METEOR.** METEOR is an automatic metric for evaluating machine translation quality, introduced by Banerjee and Lavie in 2005 [4]. It was designed to address some of BLEU's limitations by incorporating richer linguistic matching (such as stemming and synonymy) and by combining precision and recall into a single score that correlates better with human judgments.

METEOR works by first aligning the candidate translation to one or more reference translations at the word level. Matches are performed in the following order of priority: exact surface form, stemmed form, synonym matches (via WordNet), and paraphrase matches if available. Once matched, unigram precision (the fraction of matched words out of all candidate words) and unigram recall (the fraction of matched words out of all reference words) are computed and combined into an F-mean score, with recall weighted higher than precision by default.

To account for word order, METEOR applies a fragmentation penalty based on the number of chunks (i.e., groups of contiguous matches) relative to the total number of matches. The final METEOR score is given by:

$$\text{Score} = (1 - \text{Penalty}) \times F\text{-mean},$$

where

$$\text{Penalty} = \gamma \left(\frac{\text{chunks}}{\text{matches}} \right)^{\beta}$$

and γ and β are parameters tuned to maximize correlation with human judgments. Higher METEOR scores indicate closer agreement with reference translations across both content and structure.

4. Materials And Methods

To obtain the results, the six models presented in Section 2 were fine-tuned for 10,000 steps using the Turkish-translated "iamtarun/python_code_instructions_18k_alpaca" dataset. The fine-tuned models were then evaluated to produce ROUGE, BLEU and METEOR scores. The models' training parameters are shown in Table 1.

Table 1. Training parameters used for the models

Parameter	**Value**	**Description**
Steps	10000	Total training steps
Learning rate	2.5e-5	Step-wise learning rate
BF16	No	Brain Float 16 precision not used
FP16	Yes	16-bit floating point precision enabled
LoRA Dropout	0.1	Dropout rate for LoRA adaptation
Target modules	Q, K (LLaMA) / QKV, O (Phi3)	Targeted attention modules: Q=Query, K=Key, V=Value, O=Output

Table 1 summarizes the key training parameters used in the experiments. The total number of training steps was set to 10000, with a learning rate of 2.5e-5 applied in a step-wise manner. Regarding numerical precision, FP16 (16-bit floating point) training was enabled to improve memory efficiency, while BF16 (Brain Float 16) was not used. Low-Rank Adaptation (LoRA) was employed with a dropout rate of 0.1 to prevent overfitting during fine-tuning. The target modules specified for adaptation differ based on the model. For LLaMA, LoRA was applied to the query (Q) and key (K) projection layers, whereas for Phi-3, it was applied to the combined query-key-value (QKV) and output (O) layers. These modules correspond to components of the self-attention mechanism commonly used in transformer architectures.

During training, the dataset was split in an 80-10-10 ratio and used for the training, validation, and test phases, respectively. The loss values obtained during the models' training phases are shown in Figure 1.

As shown in Figure 1, the training and validation losses are close to each other, which means there is no sign of overfitting. In all cases, the models pre-trained on Turkish data reached lower loss values in both training and validation. Except for the smaller Phi3 model, all Turkish-tuned models also started with lower initial losses than their

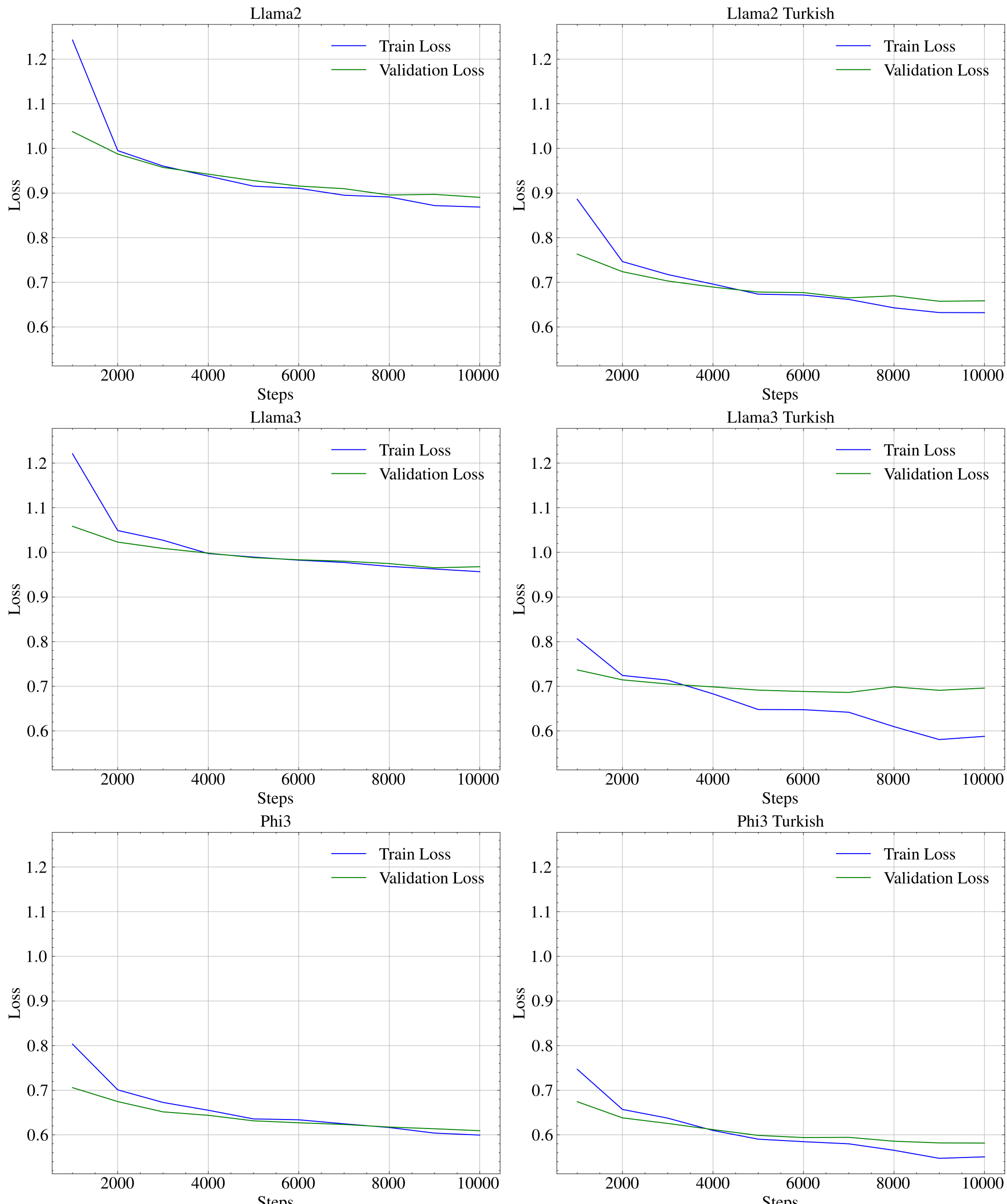

Figure 1. Loss values obtained during the training phases of the models.

base versions. This shows the advantage of using a Turkish dataset for pre-training. In addition to the lower losses, all models show smooth and steady improvement during training, which suggests that the training process was stable. The biggest improvement from Turkish pre-training is seen in the LLaMA3 Turkish model, which starts and finishes with much better losses than the base model. Although the Phi3 Turkish model begins with a similar loss to its base version, it performs better over time. This suggests that the positive effects of Turkish pre-training may become more noticeable in later training stages, especially for smaller models. The results show that using data in the target language during pre-training helps models learn better from the beginning and leads to stronger performance at the end. The final loss values at step 10,000 for all models are given in Table 2.

TABLE 2. Final training and validation losses at step 10,000

Model	Training Loss	Validation Loss
LLaMA2-7B	0.8686	0.8903
LLaMA2-7B Turkish	0.6321	0.6586
LLaMA3-8B	0.9567	0.9678
LLaMA3-8B Turkish	0.5879	0.6961
Phi-3	0.5996	0.6093
Phi-3 Turkish	0.5508	0.5817

The loss values show that models pre-trained on Turkish command data consistently reach lower losses, reflecting improved Turkish understanding through pre-processing. The BLEU, ROUGE-1, ROUGE-2, and ROUGE-L scores on the test set for all fine-tuned models are listed in Table 3.

TABLE 3. Test set BLEU, ROUGE and METEOR scores for each model

Model	BLEU	ROUGE-1	ROUGE-2	ROUGE-L	METEOR
LLaMA2-7B	0.0238	0.2894	0.1031	0.2148	0.1867
LLaMA2-7B-Chat-hf-Turkish	0.0268	0.2831	0.1115	0.2107	0.1917
LLaMA3-8B	0.0673	0.3817	0.1972	0.3096	0.2895
LLaMA3-Instruct-Turkish	0.0789	0.3949	0.2228	0.3277	0.3170
Phi3-4k	0.0299	0.1677	0.0810	0.1523	0.2141
Phi-3-mini-4k-Instruct-Turkish	0.0317	0.1373	0.0731	0.1259	0.2216

Phi does better than LLaMA 2 because it was pre-trained and tuned mostly on code data, so it learns code patterns and terms better even with fewer parameters. LLaMA 3 is also a general-purpose model but has three main improvements: it has more parameters (8 billion vs. 7 billion), its pretraining included more code examples, and its instruction-tuning was improved. These changes let LLaMA 3 outperform both LLaMA 2 and the smaller Phi model on code generation.

Although these values are low in absolute terms (given that the maximum possible score is 1), this reflects the nature of programming-language output: valid code can be written in many different ways or with different variable names, which may cause a perfectly correct result to be scored poorly by text-based metrics. An example of this phenomenon—showing a ground truth and a generated output from the top-performing Llama 3 model—is presented in Table 4.

As shown in Table 4, although the Llama 3 model actually generates code that produces the same result as the ground truth, it scores poorly on metrics like ROUGE, BLEU or METEOR due to the variable names it chooses and the extra statements it includes—yet still produces correct, executable code.

5. CONCLUSIONS

As software development processes grow more complex, changes to applications during the various stages of development become inevitable. Accordingly, a range of tools and new methods are provided to support developers throughout these stages.

The adoption of artificial intelligence in every field has also made its mark on software development, integrating AI into these processes. Research has explored AI support at every phase—from error detection and testing to code generation and software cost estimation.

In particular, developers leverage AI assistance during the coding phase to automatically generate repetitive "boilerplate" code segments and to receive help when they encounter errors. Likewise, novice programmers with limited experience use these tools to gain hands-on practice.

Although many models exist—especially in English—that translate natural language to code or answer programming questions in natural language, there is currently no open-source model that handles queries posed in Turkish. The proprietary models that do support Turkish incur cost and security concerns.

Table 4. Ground truth vs. model output for the quadratic-intercept example (LLaMA 3).

Question: Create a Python program to compute the x-intercepts of $ax^2 + bx + c = 0$. Input: $a = 1,\ b = 4,\ c = -5$.

Ground Truth:

```
import math

def quadratic_x_intercept(a, b, c):
    discriminant = b**2 - 4*a*c
    if discriminant >= 0:
        x1 = (-b + math.sqrt(discriminant)
            ) / (2*a)
        x2 = (-b - math.sqrt(discriminant)
            ) / (2*a)
        return x1, x2
    else:
        return "No real solution!"

a = 1
b = 4
c = -5
intercepts = quadratic_x_intercept(a, b, c
    )
print(intercepts)
```

Model Output:

```
import math

def quadratic_x_intersection(a, b, c):
    d = b**2 - 4*a*c
    if d > 0:
        x1 = (-b + math.sqrt(d)) / (2 * a)
        x2 = (-b - math.sqrt(d)) / (2 * a)
        return x1, x2
    else:
        print("The equation has no real 
            solution.")

a = 1
b = 4
c = -5
x1, x2 = quadratic_x_intersection(a, b, c)
print("The x-intercepts are:", x1, "and",
    x2)
```

In this study, three different open-source models—and their variants previously fine-tuned on Turkish instruction datasets—were trained to respond to problems stated in Turkish natural language, and their performance was evaluated. The results show that models fine-tuned on Turkish data achieve notably better outcomes. As expected, Llama3 models outperformed other ones, attaining higher scores. Also, Phi3's better results compared to LLaMA 2 make it a more usable alternative with lower resource requirements. Moreover, the fact that these systems produce correct, runnable code—despite using different variable names or alternative implementation approaches—demonstrates that they are not merely memorizing examples. Trained models and the trained models and source code are available at: https://github.com/emirozturk/Turkish-Text-2-Code

In future work, to enhance the models' generalization, datasets in various programming languages will be collected and translated into Turkish, and the models will be further trained on these multilingual, Turkish-translated corpora. The ultimate goal is to develop a code-assistant model capable of answering questions asked in Turkish.

The study also shows that different metrics are needed to assess the correctness of the generated code outputs. Although some correlation can be observed in the results, obtaining low scores on a benchmark with values between 0 and 1 can lead to contentious conclusions. Therefore, it appears that there is a gap in alternative metrics specifically designed for code output, and exploring these alternatives could be the subject of future work.

Conflicts of Interest

The author/authors declare that there are no conflicts of interest regarding the publication of this article.

Authors Contribution Statement

All authors jointly worked on the results and they have read and agreed to the published version of the manuscript.

References

[1] Abdin, M., Aneja, J., Behl, H., Bubeck, S., Eldan, R. et al., *Phi-4 technical report*, arXiv preprint arXiv:2412.08905, (2024).

[2] Allamanis, M., Tarlow, D., Gordon, A., Wei, Y., *Bimodal modelling of source code and natural language*, International Conference on Machine Learning, PMLR, (2015), 2123–2132.

[3] Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C., *A survey of machine learning for big code and naturalness*, ACM Computing Surveys, **51**(2018), 1–37.

[4] Banerjee, S., Lavie, A., *METEOR: An automatic metric for MT evaluation with improved correlation with human judgments*, Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, (2005), 65–72.

[5] Campbell, B.A., Treude, C., *NLP2Code: Code snippet content assist via natural language tasks*, Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), (2017), 628–632.

[6] Dehaerne, E., Dey, B., Halder, S., De Gendt, S., Meert, W., *Code generation using machine learning: A systematic review*, IEEE Access, **10**(2022), 82434–82455.

[7] Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A. et al., *Program synthesis using natural language*, Proceedings of the 38th International Conference on Software Engineering, (2016), 345–356.

[8] Falcini, F., Lami, G., Costanza, A. M., *Deep learning in automotive software*, IEEE Software, **34**(2017), 56–63.

[9] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X. et al., *CodeBERT: A pre-trained model for programming and natural languages*, arXiv preprint arXiv:2002.08155, (2020).

[10] Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A. et al., *The LLaMA 3 herd of models*, arXiv preprint arXiv:2407.21783, (2024).

[11] Guo, Q., Xie, X., Li, Y., Zhang, X., Liu, Y. et al., *Audee: Automated testing for deep learning frameworks*, Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, **2020**, 486–498.

[12] Jiang, D., Liu, Y., Liu, S., Zhao, J., Zhang, H. et al., *From CLIP to DINO: Visual encoders shout in multi-modal large language models*, arXiv preprint arXiv:2310.08825, (2023).

[13] Li, K., Zhu, A., Zhao, P., Song, J., Liu, J., *Utilizing deep learning to optimize software development processes*, arXiv preprint arXiv:2404.13630, (2024).

[14] Lin, C.-Y., *ROUGE: A package for automatic evaluation of summaries*, Text Summarization Branches Out, **2004**, 74–81.

[15] Narayanan, A., Wang, J., Shi, L., Wei, M., Wang, S., *Automatic unit test generation for deep learning frameworks based on API knowledge*, arXiv preprint arXiv:2307.00404, (2023).

[16] Papineni, K., Roukos, S., Ward, T., Zhu, W.-J., *BLEU: A method for automatic evaluation of machine translation*, Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, (2002), 311–318.

[17] Royce, W.W., *Managing the development of large software systems: concepts and techniques*, Proceedings of the 9th International Conference on Software Engineering, (1987) , 328–336.

[18] Shin, J., Nam, J., *A survey of automatic code generation from natural language*, Journal of Information Processing Systems, **17**(2021), 537–555.

[19] Sreekanth, N., Rama Devi, J., Shukla, K.A., Mohanty, D.K., Srinivas, A. et al., *Evaluation of estimation in software development using deep learning-modified neural network*, Applied Nanoscience, **13**(2023), 2405–2417.

[20] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A. et al., *LLaMA 2: Open foundation and fine-tuned chat models*, arXiv preprint arXiv:2307.09288, (2023).