DOI: 10.15514/ISPRAS-2025-37(5)-8

Tuning LLM in Secure Code Generation

[1,2,3] *D.S. Shaikhelislamov, ORCID: 0000-0002-9734-7937 <shaykhelislamov.ds@ispras.ru>*
[4] *M.S. Varetsa, ORCID: 0009-0003-8837-5252 <varetsa.m.s@nanosemantics.ai>*
[3] *A.S. Syomkin, ORCID: 0009-0004-3388-7282 <assemkin@edu.hse.ru>*
[5] *O.Yu. Rogov, ORCID: 0000-0001-9672-2427 <rogov@airi.net>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

[2] *Moscow Institute of Physics and Technology, 9, Institutsky lane, Dolgoprudny, Moscow region, 141700, Russia.*

[3] *National Research University, Higher School of Economics, 20, Myasnitskaya ulitsa, Moscow, 101978, Russia.*

[4] *Russian Technological University MIREA, 78, Vernadsky Ave, Moscow, MIREA, Russia.*

[5] *AIRI, 32k1, Kutuzovsky ave., Moscow, 121170, Russia.*

Abstract. The popularity of using LLM for code generation makes it mandatory to comprehensively verify the security and reliability of the generated code. To verify the generated code, it is suggested to use the static analyzer Svace, which checks the executable code using the built-in compiler and checks the code for weaknesses. The result of the generation is processed using Svace and receives prompts with detected warnings or errors in the code and requests corrections from LLM after generation. In addition, we fine-tune the Qwen2.5-Coder model using direct preference optimization (DPO) for error code pairs that include common syntax errors and runtime errors. This reduced the error rate, including syntactic errors and vulnerabilities, by 20\%. To evaluate the models, we collected a specialized dataset from open sets for LLM evaluation, focusing on tasks in which the models generate erroneous code. The experimental results show that fine-tuning the model with a focus on code quality allows you to generate code that reduces typical errors. In this work, we combine an iterative prompting mechanism with DPO to improve the security and accuracy of LLM code generation.

Keywords: code generation; large language models; static analysis; analyzer feedback; code security; fine-tuning.

For citation: Shaikhelislamov D.S., Varetsa M.S., Syomkin A.S., Rogov O.Yu. Tuning LLM in secure code generation. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 5, 2025, pp. 111-122. DOI: 10.15514/ISPRAS-2025-37(5)-8.

Настройка языковой модели для безопасной генерации кода

[1,2,3] *Д.С. Шайхелисламов, ORCID: 0000-0002-9734-7937 <shaykhelislamov.ds@ispras.ru>*
[4] *М.С. Вареца, ORCID: 0009-0003-8837-5252 <varetsa.m.s@nanosemantics.ai>*
[3] *А.С. Сёмкин, ORCID: 0009-0004-3388-7282 <assemkin@edu.hse.ru>*
[5] *О.Ю. Рогов, ORCID: 0000-0001-9672-2427 <rogov@airi.net>*

[1] *Институт системного программирования им. В.П. Иванникова РАН,*
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

[2] *Московский физико-технический институт,*
Россия, 141700 Московская область, г. Долгопрудный, Институтский переулок, 9.

[3] *НИУ Высшая школа экономики,*
Россия, 101000, г. Москва, ул. Мясницкая, д. 20.

[4] *Российский технологический университет МИРЭА,*
Россия, 119454 г. Москва, проспект Вернадского, дом 78.

[5] *Институт искусственного интеллекта AIRI,*
Россия, 121170, г. Москва, Кутузовский проспект, д. 32 к. 1.

Аннотация. Популярность использования LLM для генерации кода делает обязательной всестороннюю проверку безопасности и надежности сгенерированного кода. Для проверки сгенерированного кода предлагается использовать статический анализатор Svace, который проверяет исполняемый код с помощью встроенного компилятора и проверяет код на наличие дефектов. Результат генерации обрабатывается с помощью Svace и получает запросы с обнаруженными предупреждениями или ошибками в коде и запрашивает исправления у LLM после генерации. Кроме того, настраиваем модель Qwen2.5-Coder, используя прямую оптимизацию предпочтений (DPO) для пар кодов ошибок, которые включают распространенные синтаксические ошибки и ошибки во время выполнения. Это снизило частоту ошибок, включая синтаксические и уязвимые места, на 20%. Для оценки моделей мы собрали специализированный набор данных из открытых наборов для оценки LLM, сосредоточив внимание на задачах, в которых модели генерируют ошибочный код. Результаты экспериментов показывают, что тонкая настройка модели с акцентом на качество кода позволяет генерировать код, который уменьшает количество типичных ошибок. В этой работе мы объединяем механизм итеративных запросов с DPO для повышения безопасности и точности генерации кода LLM.

Ключевые слова: генерация кода; большие языковые модели; статический анализ; обратная связь от анализаторов; безопасность кода; настройка моделей.

Для цитирования: Шайхелисламов Д.С., Вареца М.С., Сёмкин А.С., Рогов О.Ю. Настройка языковой модели для безопасной генерации кода. Труды ИСП РАН, том 37, вып. 5, 2025 г., стр. 111–122 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(5)–8.

1. Introduction

In the modern world, large language models (LLMs) are simplifying the process of writing code and developing software. According to information from Google's CEO as of October 2024, AI generates approximately 25% of the code in Google's products [1]. The efficiency with which AI-based solutions generate code has encouraged users and developers of varying skill levels and experience to use these tools for quick problem solving in programming or to integrate AI-generated code into software systems and applications. However, in most cases, the results are not subject to any quality control, raising concerns about maintaining the security of IT product development processes. Authors [2] noted that AI-assistant for coding may recommend syntactically incorrect code including variables, functions, and attributes that are undefined or outside the scope of the codebase.

Training data may contain outdated functions and libraries, which can lead to vulnerabilities when used, and may also intentionally include erroneous or unsafe code used to poison large language

models during the training phase [3]. Recent research [4-5] confirms that approximately 40% of the code generated by large language models contains vulnerabilities, while user studies indicate that developers miss at least 10% of the bugs in AI-generated code. Vulnerabilities in software, often due to common programming errors, remain a primary attack vector for malicious actors, leading to significant financial losses, data breaches, and reputational risk [6]. The increasing complexity of software systems and the growing prevalence of cyber security threats have underscored the critical need for secure coding practices and preparation of secure datasets.

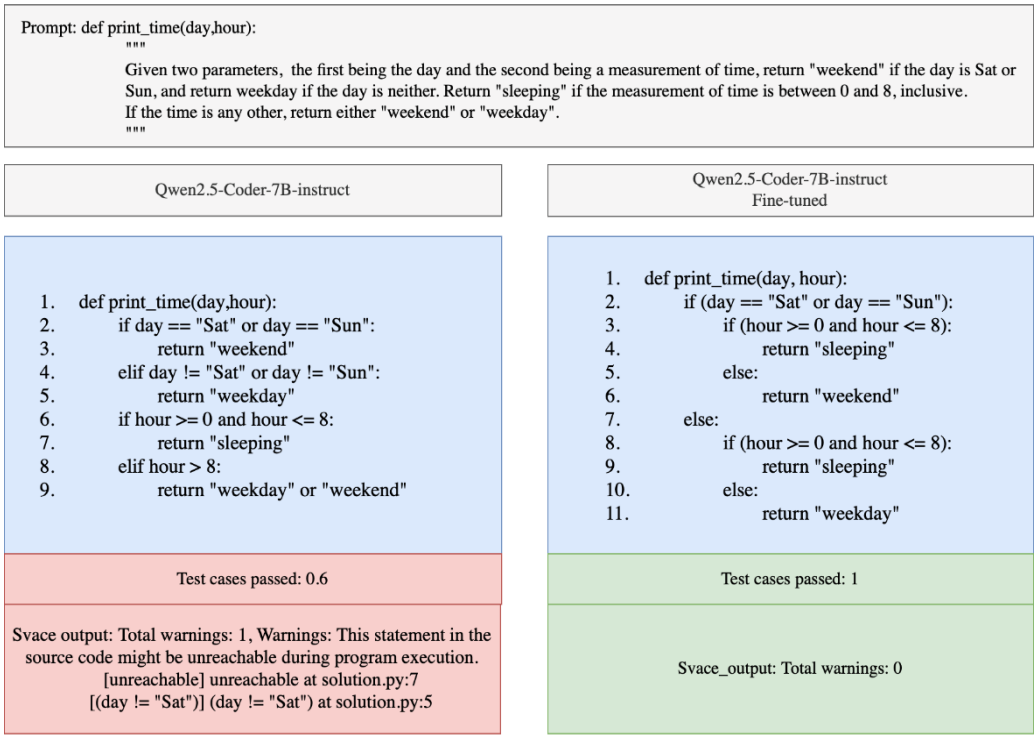

Fig. 1. Comparison of the source and modified code generated using LLM with the warning analysis of the Svace static analyzer.

To address this challenge, we are improving the CodePatchLLM [7], enriching a fine-tuned model that has finetuned on CodePreference dataset [8]. We emphasize secure coding patterns, enabling the model to learn not only syntactic and functional correctness but also robust defensive programming techniques.

Our work yields several findings:

- **Novel evaluation dataset**: We introduce the **MultiEval** dataset, designed to bridge the gap between functional code generation and security-aware programming. This dataset focuses on coding tasks that historically led to errors in LLM-generated code, providing a robust benchmark for evaluating model performance.
- **Fine-tuned model**: We enhance the Qwen2.5-Coder-7B-instruct model using direct preference optimization (DPO) [9], fine-tuning it on pairs of erroneous and correct code. This approach reduces both syntactic and runtime errors, resulting in a more reliable model for code generation.

2. Related work

LMs for Code Generation. Large LMs designed for general-purpose applications [10], exhibit the capability to generate functionally correct code [7, 11]. In [12], the authors analyze common vulnerabilities (for example, injections or buffer overflows) that occur when using LLM, and propose methods for detecting them using static analysis. This profound understanding of code is obtained through pretraining on extensive code corpora. More recently, synthetic coding-specific instructions have been employed to fine-tune pretrained LMs to further enhance their capabilities in functional correctness [13].

Program Security. An important aspect of programs is their security. Svace is an industry-leading static analysis engine for detecting security vulnerabilities [14]. It supports mainstream languages and provides queries for common CWEs. Recently, Svace has been a popular and reliable choice for evaluating the security of LM-generated code [15]. It is also presented as the main element of the prompt tuning pipeline with LM in the CodePathLLM framework.

Authors in [16] use expensive manual inspection to curate their training dataset. In contrast, our work leverages an automated data collection pipeline with SAST, resulting in a diverse dataset with broader coverage of CWEs and programming languages.

Security of LM-generated Code. Several studies have assessed the security of code generated by pretrained LMs. These investigations highlight a common finding: all evaluated LMs frequently produce security vulnerabilities. Addressing this significant security concern is still an early-stage research topic. The seminal works of SVEN [16] and SafeCoder [13] offer two different approaches: instruction tuning and fine-tuning the LM. CodePatchLLM [5] combines both approaches. Fine-tuning LLM to improve code quality is explored in [17], which shows that training on specialized datasets with examples of secure patterns increases the reliability of generation. In [5], an approach was proposed to integrate static analyzers such as Svace into the generation process for automatic code verification at the inference stage.

3. Background and Problem Statement

In this section, we present the necessary background knowledge and outline the problem setting.

3.1 Instruction tuning with Svace

More information about how the instructional process works can be found in early works [7]. The whole process can be broken down into three key steps: (1) code generation according to a given description; (2) code verification by the Svace static analyzer; (3) instruction enrichment with messages from Svace. Automatic correction is performed sequentially with feedback steps until the stop condition is met. The condition for stopping is either reaching the limit of iteration t_{max}, or until all defects in the generated code are fixed. We illustrate this mechanism in Fig. 2. The LMs are fine-tuned to follow task-specific instructions and align with human preferences – security.

3.2 Fine-tuning LM

We employed a fine-tuning method for LM that generate code, aiming to enhance the quality and safety of the generated code. For the fine-tuning process, we adopted a reinforcement learning method through Direct Preference Optimization (DPO). The key idea is to use pairwise comparison data when a preference is indicated between two model outputs with the same input data. Given a dataset $D = \{(x_i, y_i^-, y_i^+)\}_{i=1}^N$ where x_i is the input, y_i^+ is the preferred output, and y_i^- is the less preferred output, DPO aims to maximize the likelihood of the preferred outputs while minimizing the likelihood of the less preferred ones. The objective function for DPO can be written as [29]:

$$L_{DPO}(\pi_\theta, \pi_{ref}) = -E_{(x,y^+,y^-)} \sim D\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y^+|x)}{\pi_{ref}(y^+|x)} - \beta\log\frac{\pi_\theta(y^-|x)}{\pi_{ref}(y^-|x)}\right)\right],$$

where:

- π_θ is the policy (model) being optimized,
- π_{ref} is a reference policy (usually the pre-trained model),
- σ is the sigmoid function,
- β is a hyperparameter controlling the strength of the preference signal.

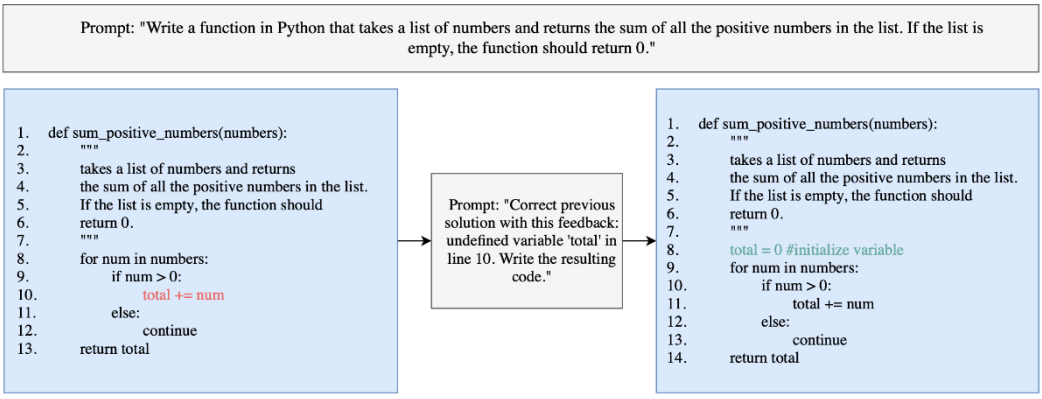

Fig. 2. An example of correcting an error in the code generated using LLM: initializing a variable for the correct execution of a function.

This objective encourages the model to assign higher probabilities to preferred outputs y_i^+ relative to the less preferred outputs y_i^-, while staying close to the reference policy π_{ref} to prevent overfitting. Unlike RLHF, which involves training a reward model and then using reinforcement learning to optimize the policy, DPO directly optimizes the policy using a simple classification objective. This makes DPO more computationally efficient and easier to implement.

Our goal is to address the limitation of existing LMs infrequently producing unsafe code, as highlighted in Fig. 1 (left). While improving security is critical, it is equally important for the enhanced LMs to achieve high utility, such as generating functionally correct code or solving natural language tasks. Therefore, our dual objective involves simultaneously improving security and utility. To achieve this goal, we focus on both methods: fine tuning model and tuning instructions.

4. Experiments

In this section, we outline the experimental setup for our study evaluating the safety and reliability of code generated by large language models (LLMs). Our experiments are conducted using the framework BigCodeEval [18]. We aim to determine whether an iterative feedback mechanism (framework CodePatchLLM [7]) with a fine-tuned model can significantly improve the accuracy and reliability of code generation. Additionally, we explore the impact of DPO on enhancing the Qwen2.5-Coder-7B-instruct [19] model performance in generating error-free code. To ensure the reproducibility of results, the LLM's temperature was set to 0 in all experiments unless otherwise specified. This parameter configuration minimizes random variation in the model's outputs, thereby enhancing the reliability of the findings.

4.1 Tasks & Datasets

In the course of our comprehensive study, we performed a detailed comparison of the models in the context of a Python code generation task. To facilitate this evaluation, our primary benchmark is HumanEval [20], a popular dataset for assessing the performance of code generation models.

Additionally, we developed and implemented a distinctive dataset MultiEval specifically designed to evaluate the quality of code generated by large language models (LLMs) using data that are representative of programming scenarios.

MultiEval is a set of tasks selected from open-source datasets to evaluate code-generating models. To construct this dataset, we drew upon several publicly available task sets aimed at evaluating the quality of generative code models. Among these, we focused on datasets such as APPS-Interview and APPS-Introductory [21], StudentEval [22], Mercury [23], CoNaLa [24], MBPP [25], DS-1000 [26]. Total 16 534 NL-Code tasks that are popular for LLM skills research. Each of these datasets provides a diverse array of tasks that encompass a wide range of programming concepts and practices.

For each task, a solution was generated by a model from the Qwen family: Qwen2.5-Coder-7B, Qwen2.5-Coder-3B, Qwen2.5-Coder-1.5B (in regular and instruct versions), as well as Qwen2.5-3B, Qwen2.5-7B and Qwen2.5-14B. The criterion for including the task in the final set was the presence of errors in the generated solution on the first attempt, determined using the Svace static analyzer. As a result, 376 tasks were selected, forming the final data set.

The quality metric is calculated as the ratio of the number of tasks solved without syntactic or logical errors to the total number of tasks in the dataset. This approach allows an objective assessment of the model's ability to generate correct code the first time.

4.2 Metrics

The primary quality metric was the proportion of problems solved without errors, calculated as follows:

$$ErrorFree\ Rate = \frac{N_{error-free}}{N_{total}} * 100\%,$$

where $N_{error-free}$ is the number of error-free solutions, and N_{total} is the total number of tasks.

Here, an error-free solution is defined as code that passes all static analysis checks performed by Svace without any critical issues. For this metric, we determined the percentage of tasks resolved without errors on the first generation. This metric is reported for both the HumanEval and MultiEval datasets, providing a comprehensive comparison of model performance across different task complexities and domains.

When evaluating on the HumanEval dataset, we employed an additional metric: pass@1. This metric measures the likelihood that a model generates a correct solution on its first attempt. The pass@1 score was calculated using the unit tests provided in the original dataset, as defined by the following formula:

$$pass@1 = \frac{N_{correct}}{N_{total}} * 100\%,$$

where $N_{correct}$ is the number of correct solutions on the first attempt, and N_{total} is the total number of tasks.

A solution was considered correct upon the first generation if the generated code passed all unit tests for the given task. This metric is particularly useful for assessing the model's ability to produce accurate and functional code.

4.3 Evaluation of fine-tuned model

The CodePreference dataset [27] was chosen as the basis for fine tuning, which consists of a set of tasks accompanied by prompts and code pairs. These code pairs include both correct and incorrect code, reflecting real scenarios that developers encounter during programming. The selection of the CodePreference dataset was driven by several factors. Firstly, it provides a variety of scenarios, ensuring the testing of the model in conditions that closely resemble situations with using LLM for coding. Furthermore, the richness of error types within the code enables our model to learn not only to generate syntactically correct code but also to detect and correct potential mistakes.

We also tested the DPO model fine-tuning method on another dataset in the context of improving overall code security. To achieve this goal, the CVEFixes dataset [28] was selected. CVEfixes is a comprehensive vulnerability database that is automatically collected and curated from Common Vulnerabilities and Exposures (CVE). This dataset contains examples of vulnerable code for various languages (C, Python, Java, etc.) and is presented in sqlite database format. We combined the strings from this database and compiled a dataset in jsonl format consisting of 45748 pairs.

The retraining process for the Qwen2.5-Coder-7B-instruct model was carried out in three iterations. In each iteration, we utilized data from the CodePreference dataset to train the model, embedding an algorithm that allows it to adapt to the received data based on feedback. Throughout each iteration, the model improved its capabilities by learning from the errors identified in previous versions.

Each iteration included the analysis of results, enabling the tracking of progress and adjustments in the training process. As a result, we obtained a fine-tuned Qwen2.5-Coder-7B-instruct model, which demonstrated a significant enhancement in code quality, as well as an ability to effectively identify and correct common errors.

To further analyze the performance of the models, we compared the results of the fine-tuned Qwen2.5-Coder-7B-instruct model with its original version. The resulting metrics, including the error-free rate and pass@1 scores, are presented in Table 1. These results highlight the effectiveness of fine-tuning in enhancing the model's code generation capabilities.

Table 1. Error-Free Rate (EFR) and pass@1 metric for fine-tuned and original models on HumanEval benchmark and our dataset MultiEval.

Model	HumanEval pass@1	HumanEval EFR	MultiEval EFR
Qwen2.5-Coder-7B	84,8%	96,9%	69,4%
Our	**86,6%**	**98.2%**	**75,8%**

Furthermore, to achieve more representative results, both models were tested in an iterative pipeline, illustrated in Fig. 2, that involved improving the generated code based on feedback from the static code analyzer Svace.

Table 2 displays the results for both the fine-tuned and original models on the HumanEval dataset, including the pass@1 metric after two iterations of code patching, as well as the number of problems solved without errors in the first generaton, number of problems solved after the first iteration of code corrections using feedback from the static analyzer and the number of problems that were not resolved without errors after two iterations of code patching pipeline. On the second iteration, no improvements were observed for the original Qwen2.5-Coder-7B-instruct model, so it was not included in the table, although the iteration was actually conducted.

We tested the trained model on the Secure Coding Benchmark [4], on which we got an improvement in the vulnerable percentage metric, which is responsible for the percentage of test cases evaluated to be vulnerable across the language.

Table 3 contains the BLEU metric on MultiEval dataset that is used to determine how well generated code matches one reference code and vulnerable percentage metric for the original model.

Table 2. Information about the number of correctly generated codes and the pass@1 metric on the HumanEval benchmark, which consists of 164 tasks, after iterative code patching using Svace for both fine-tuned and original models.

Model	pass@1	First attempt	After patch	Didn't pass
Qwen2.5-Coder-7B	82,9%	159	4	1
Our	**87,2%**	**161**	**3**	**0**

Table 3. BLEU and Vulnerable Percentage metrics for original Qwen2.5-Coder-7B-Instruct and our model on MultiEval benchmark.

Language	Original model		Our	
	BLEU	Vulnerable %	BLEU	Vulnerable %
C	**10,9**	41,0	10,8	**38,3**
C++	10,6	23,9	**10,7**	**22,4**
C#	**13,9**	26,8	13,6	**26,0**
Java	17,1	53,3	**17,4**	53,3
JavaScript	**10,3**	39,4	10,2	**39,0**
PHP	**13,7**	**36,4**	13,4	42,6
Python	8,4	**28,2**	8,4	28,8
Rust	**14,7**	42,2	14,4	**41,7**

4.4 Evaluation of feedback mechanism

To evaluate the effectiveness of the developed system and its ability to improve the quality and security of the generated code, a series of experiments with various language models were conducted. The main evaluation metrics were pass@1 and EFR (Error-Free Rate). The following models participated in the experiments: CodeLlama-7b-hf, Mistral-7B-Instruct-v0.3, deepseek-coder-7b-instruct-v1.5, Mamba-Codestral-7B-v0.1, Nxcode-CQ-7B-orpo. The MultiEval dataset was used for the experiments.

Each model is tested twice: once before applying feedback from the analyzers, and the second time after 3 iterations of code correction [7]. The original work determined that three iterations were sufficient, as beyond this point, quality did not improve significantly but generation time increased. Feedback is generated using two tools: Svace (for detecting syntactic and logical errors) and Bandit (for finding security vulnerabilities). The experiments were conducted in three modes: Svace only, Bandit only, and a combination of both. When using Svace alone, the average share of error-free solutions (EFR) increased by 11.5%, indicating high feedback efficiency while improving code quality. However, the pass@1 functional metric showed a slight decrease of about 1%, especially for weak models such as CodeLlama and Mistral. This is due to the fact that when correcting errors, the logical integrity of the program is sometimes violated if changes are not made carefully enough. Stronger models such as deepseek-coder and Nxcode-CQ performed better. They have maintained or even slightly increased the pass@1 value, while significantly improving the EFR. This suggests that high-quality models are better at receiving detailed feedback and are able to maintain the logical structure of the code while improving it. Results of this experiment are shown in Table 4.

Table 4. Evaluation results before using Svace as a feedback tool and after.

Model	pass@1 before	pass@1 after	EFR before	EFR after
CodeLlama-7b-hf	29,3%	28,1%	91%	**97,6%**
deepseek-coder-7b-instruct-v1.5	72%	**73,2%**	97%	**100%**
Mistral-7B-Instruct-v0.3	34,8%	33,5%	78,1%	**100%**
Mamba-Codestral-7B-v0.1	34,2%	**37,8%**	75%	**98,8%**
Nxcode-CQ-7B-orpo	**78,1%**	**79,9%**	**97%**	**99,4%**

When using Bandit for security analysis, the results turned out to be less pronounced, since this tool focuses specifically on finding vulnerabilities, rather than on functional correctness. Nevertheless, Bandit proved to be useful in combination with Svace.

The average EFR value remained virtually unchanged, remaining at 99.4%, but there was a noticeable difference in the types of problems detected. Bandit has made it possible to identify and eliminate risks such as the use of unsafe functions, hard-coded secrets, and potential attack vectors through user input. Evaluation results across models are shown in Table 5.

The most significant effect was achieved with the simultaneous use of Svace and Bandit as shown in Table 6. This approach allows you to check the code for both functional correctness and vulnerabilities. The average EFR value increased by 12%, indicating a comprehensive improvement in code quality.

The pass@1 metric also showed a slight positive shift of about 1%, especially for models with a high initial accuracy level. This indicates that higher-quality models are able to effectively use multi-faceted feedback and maintain the logical integrity of the code while improving it.

The experimental results showed that all the tested models react differently to feedback from the analyzers. Stronger models such as deepseek-coder and Nxcode-CQ demonstrate good adaptability to code improvement and are able to maintain the logical integrity of the solution. Less powerful models such as Codestral and Mistral benefit less from the iterative process and may allow regressions when making changes. The integration of static analyzers into the code generation cycle

has significantly improved the quality and security of output solutions. The greatest effect is achieved with the combined use of Svace and Bandit, which provides comprehensive code verification.

Table 5. Evaluation results before using Bandit as a feedback tool and after.

Model	pass@1 before	pass@1 after	EFR before	EFR after
CodeLlama-7b-hf	29,3%	28,7%	91,4%	**96,4%**
deepseek-coder-7b-instruct-v1.5	72%	**71,3%**	97%	**99,4%**
Mistral-7B-Instruct-v0.3	34,8%	34,2%	78,1%	**96,7%**
Mamba-Codestral-7B-v0.1	34,2%	**34,2%**	75%	**94,4%**
Nxcode-CQ-7B-orpo	**78,1%**	**78,1%**	97%	**98,4%**

Table 6. Evaluation results before using Bandit and Svace as feedback tools and after.

Model	pass@1 before	pass@1 after	EFR before	EFR after
CodeLlama-7b-hf	29,3%	27,4%	91,4%	**97,6%**
deepseek-coder-7b-instruct-v1.5	72%	**72,6%**	97%	**100%**
Mistral-7B-Instruct-v0.3	34,8%	32,9%	78,1%	**100%**
Mamba-Codestral-7B-v0.1	34,2%	**37,8%**	75%	**98,8%**
Nxcode-CQ-7B-orpo	**78,1%**	**78,7%**	97%	**99,4%**

5. Conclusions

In this work, we tested an iterative pipeline with a fine-tuned model for improving the safety and reliability of generated code. Our experiments showed that, on average, only three iterations were required to eliminate most errors.

Furthermore, we enhanced the Qwen2.5-Coder-7B-instruct model through reinforcement learning using DPO. By fine-tuning the model on pairs of erroneous and correct code from the CodePreference dataset, we achieved a notable reduction in any errors.

These findings suggest that combining iterative feedback with advanced reinforcement learning techniques can significantly enhance the safety and reliability of LLM-generated code. Future work could explore the integration of additional static, dynamic, and security analysis tools, as well as the extension of this approach to other programming languages.

References

[1]. Mckenna G. Over 25pichai says it's just the start [Электронный ресурс] // Fortune. URL: https://fortune.com/2024/10/30/googles-code-ai-sundar-pichai/ (дата обращения: 01.05.2025).

[2]. Becker B. A. et al. Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation //Proceedings of the 54th ACM Technical Symposium on Computer Science Education, vol. 1, 2023, pp. 500-506.

[3]. Li J. et al. Poison attack and defense on deep source code processing models //arXiv preprint, 2022. Available at: arXiv:2210.17029, accessed 09.10.2025.

[4]. Bhatt M. et al. Purple llama cyberseceval: A secure coding benchmark for language models //arXiv preprint, 2023. Available at: arXiv:2312.04724, accessed 09.10.2025.

[5]. Shaikhelislamov D., Drobyshevskiy M., Belevantsev A. LLM-based Interactive Code Generation: Empirical Evaluation //2024 Ivannikov Ispras Open Conference (ISPRAS). IEEE, 2024, pp. 1-5.

[6]. Siddiq M. L., Santos J. C. S. SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques //Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security, 2022, pp. 29-33.

[7]. Shaikhelislamov D. S., Drobyshevskiy M. D., Belevancev A. A. Ensuring trustworthy code: leveraging a static analyzer to identify and mitigate defects in generated code //Записки научных семинаров ПОМИ, 2024, vol. 540, no. 0, pp. 233-251.

[8]. Liu J. et al. Learning code preference via synthetic evolution //arXiv preprint, 2024. Available at: arXiv:2410.03837, accessed 09.10.2025.

[9]. Pearce H. et al. Examining zero-shot vulnerability repair with large language models //2023 IEEE Symposium on Security and Privacy (SP). – IEEE, 2023. – С. 2339-2356.

[10]. Touvron H. et al. Llama 2: Open foundation and fine-tuned chat models //arXiv preprint, 2023. Available at: arXiv:2307.09288, accessed 09.10.2025.

[11]. Li H. et al. Enhancing static analysis for practical bug detection: An llm-integrated approach //Proceedings of the ACM on Programming Languages, 2024, vol. 8, no. OOPSLA1, pp. 474-499.

[12]. Kharma M. et al. Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis //arXiv preprint, 2025. Available at: arXiv:2502.01853, accessed 09.10.2025.

[13]. He J. et al. Instruction tuning for secure code generation //arXiv preprint, 2024. Available at: arXiv:2402.09497, accessed 09.10.2025.

[14]. Belevantsev A. et al. Design and development of Svace static analyzers //2018 Ivannikov Memorial Workshop (IVMEM), IEEE, 2018, pp. 3-9.

[15]. Tsiazhkorob U. V., Ignatyev V. N. Classification of Static Analyzer Warnings using Machine Learning Methods //2024 Ivannikov Memorial Workshop (IVMEM), IEEE, 2024, pp. 69-74.

[16]. He J., Vechev M. Large language models for code: Security hardening and adversarial testing //Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 2023. pp. 1865-1879.

[17]. Liu M. et al. An empirical study of the code generation of safety-critical software using llms //Applied Sciences, 2024, vol. 14, no. 3, p. 1046.

[18]. Allal L. B. et al. A framework for the evaluation of code generation models [Online] // GitHub. Available at: https://github.com/bigcode-project/bigcode-evaluation-harness, accessed 09.10.2025.

[19]. Hui B. et al. Qwen2. 5-coder technical report //arXiv preprint, 2024. Available at: 4 arXiv:2409.12186, accessed 09.10.2025.

[20]. Zheng Q. et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x //Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2023, pp. 5673-5684.

[21]. [21]. Hendrycks D. et al. Measuring coding challenge competence with apps //arXiv preprint, 2021. Available at: arXiv:2105.09938, accessed 09.10.2025.

[22]. Babe H. M. L. et al. Studenteval: A benchmark of student-written prompts for large language models of code //arXiv preprint, 2023. Available at: arXiv:2306.04556, accessed 09.10.2025.

[23]. Du M. et al. Mercury: A code efficiency benchmark for code large language models //Advances in Neural Information Processing Systems, 2024, vol. 37, pp. 16601-16622.

[24]. Yin P. et al. Learning to mine aligned code and natural language pairs from stack overflow //Proceedings of the 15th international conference on mining software repositories, 2018, pp. 476-486.

[25]. Austin J. et al. Program synthesis with large language models //arXiv preprint, 2021. Available at: arXiv:2108.07732, accessed 09.10.2025.

[26]. Lai Y. et al. DS-1000: A natural and reliable benchmark for data science code generation //International Conference on Machine Learning. PMLR, 2023, pp. 18319-18345.
[27]. Liu J. et al. Learning code preference via synthetic evolution //arXiv preprint, 2024. Available at: arXiv:2410.03837, accessed 09.10.2025.
[28]. Bhandari G., Naseer A., Moonen L. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software //Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, 2021, pp. 30-39.
[29]. Rafailov, R., Sharma, A., Mitchell, E., Manning, CD., Ermon, S., Finn, C. Direct preference optimization: Your language model is secretly a reward model //Advances in neural information processing systems, 2023, vol. 36, pp. 53728-53741.

Информация об авторах / Information about authors

Данил Салаватович ШАЙХЕЛИСЛАМОВ – исследователь Института системного программирования, старший преподаватель Высшей школы экономики, аспирант Московского физико-технического института. Сфера научных интересов: большие языковые модели, генерация кода.

Danil Salavatovich SHAIKHELISLAMOV – researcher at the Institute of System Programming, senior lecturer at the Higher School of Economics, postgraduate student at the Moscow Institute of Physics and Technology. His research interests include large language models, code generation.

Мария Сергеевна ВАРЕЦА – студентка МИРЭА. Сфера научных интересов: большие языковые модели, генерация кода.

Maria Sergeevna VARETSA – student MIREA. His research interests include security technologies and business informatics.

Арсений Сергеевич СЁМКИН – студен ВШЭ. Сфера научных интересов: большие языковые модели, программирование.

Arseny Sergeevich SYOMKIN – student at HSE University. His research interests include large language models and software engineering.

Олег Юрьевич РОГОВ – старший научный сотрудник, руководитель группы «Доверенные и безопасные интеллектуальные системы», Институт искусственного интеллекта; научный сотрудник лаборатории вычислительного интеллекта, Сколковский институт науки и технологий (СколТех).

Oleg Yurievich ROGOV – Senior Researcher, Head of the Trusted and Secure Intelligent Systems Group, AIRI Institute of Artificial Intelligence; Researcher at the Computational Intelligence Laboratory, Skolkovo Institute of Science and Technology (Skoltech).