# LLM4CodeSec: A Framework for Evaluating the Effectiveness of Large Language Models in Source Code Vulnerability Detection

K. I. Gladkikh, A. A. Zakharov, I. G. Zakharova

*Abstract*— **In the context of a sustained increase in software vulnerabilities and the growing adoption of large language models (LLMs) for source code analysis, objective evaluation of their effectiveness in vulnerability detection tasks becomes increasingly important. Despite a substantial body of research in this area, most existing studies focus on specific programming languages, limited datasets, or proprietary models, which hinders reproducibility and comparability.**

**This paper presents a LLM4CodeSec framework for the comprehensive evaluation of large language models in source code vulnerability detection tasks. The framework is implemented in Python and provides a unified infrastructure for reproducible experiments with various language models, datasets, prompting strategies, and evaluation metrics. Its architecture is based on object-oriented design principles and ensures extensibility without modifying the system core. The framework supports binary and multiclass classification, classification by Common Weakness Enumeration (CWE) types, and risk-related metrics, including false negative rate and inference time.**

**The functionality of the proposed solution is validated through experimental evaluation on several widely used source code vulnerability benchmarks. The obtained results demonstrate the applicability of the framework for both research and practical software security analysis tasks, including integration within Continuous Integration (CI) pipelines.**

**The source code of framework available at: https://github.com/vodkar/llm4codesec-framework.**

*Keywords*—**framework, information security, large language models, source code, vulnerability detection.**

## I. Introduction

Vulnerabilities in source code remain a critical problem. The number of zero-day vulnerabilities continues to grow each year. In particular, the number of vulnerabilities increased by 35% between 2020 and 2024 [1]. In 2025, the total number of discovered vulnerabilities reached 40,000 [2].

At the same time, large language models, as well as agent-based systems built on top of them, have gained significant popularity in automating a wide range of tasks related to natural language processing, including tasks involving source code written in various programming languages. For example, language models are capable of understanding the semantics of code [3], explaining it and generating documentation, fixing errors [4], and performing code reviews [5], [6]. This suggests that language models can also be applied to the analysis of source code for vulnerability detection.

A substantial body of research has already explored the use of language models for cybersecurity [7], as well as for static analysis of program source code. For instance, in [8], the authors conduct a comprehensive literature review and propose a research roadmap for this domain. The survey presented in [9] analyzes 236 studies related to language models and software vulnerabilities, highlighting the relevance and significance of this research area. As an example, the GPT-J language model (built on top of GPT-2) achieves a vulnerability detection accuracy of 97% on the VulDeePecker benchmark [10] for C/C++ code.

At the same time, the majority of existing studies focus on the C/C++ programming languages [11], [12], despite a clear global trend toward the widespread use of JavaScript, PHP, Java, and Python [13]. Another problem, is that current research focus on isolated code snippets, rather than big projects [14]. Beyond analyzing isolated code segments, future research should be structured around the key problems in real-world development [15].

To develop an AI agent that can be integrated into CI/CD development pipelines-similar to how automated code review is already applied-it is necessary to use smaller-scale models that can be deployed on hardware resources available within an organization. Otherwise, developers must either incur additional financial costs by relying on APIs of proprietary language models, which is economically inefficient, or deploy large language models (with more than 10 billion parameters) on their own infrastructure, which requires expensive computational resources [16]. Moreover, transmitting source code to external services poses a serious risk of data leakage, potentially leading to violations of information security requirements [17], [18].

It should also be noted that there is a lack of sufficient research on the impact of prompting strategies on vulnerability detection accuracy. For example, some authors rely on proprietary models [19] and do not provide source code that would allow reproducing their results using alternative language models.

Another direction of studies focuses on optimization and increasing a performance of LLM in vulnerability detection task. Recent studies focus on how to inject a relevant for vulnerability context [20], ensemble strategies applying [21], prompt strategies optimization [22], agentic system for robust code review [23]. This plethora of researches requires an easy to use, flexible, and reliable framework for evaluation of LLM in code security tasks, with wide set of metrics and controlled environment [15].

Table I. Comparison with other works

| Feature | Our Work | VulnLLM Eval [24] | CORRECT [25] | LLM4 Vuln [26] |
|---|---|---|---|---|
| **Quantization Support** | ✓ 4-bit, 8-bit | ✗ | ✗ | ✗ |
| **Production Infrastructure** | ✓ Docker + Compose | ✗ | ✗ | ✗ |
| **Extensible Architecture** | ✓ SOLID + Interfaces | ✗ | ✗ | ⚠ Partial |
| **Dataset Scale** | ✓ | ✗ | ✗ | ✗ |

Recent frameworks (VulnLLMEval [24], CORRECT [25], LLM4Vuln [26]) advance academic understanding but lack five critical features for practical deployment: quantization support, production infrastructure (Docker), extensible architecture (SOLID), security-aware metrics (FNR), and comprehensive scale (>270K samples). Our framework addresses these gaps, bridging academic research and practical deployment. Table I presents key futures of developed framework in comparison with other papers.

In this work, we focus on the design and implementation of a comprehensive framework (LLM4CodeSec) for conducting experiments with different configurations. The framework supports multiple benchmarks using a unified experiment configuration file that includes the following parameters:

- language model name;
- language model parameters (maximum number of tokens, temperature, quantization, etc.);
- prompt used by the language model;
- output evaluation metrics;
- dataset.

The practical relevance of locally deployed language models is confirmed by the experimental results of an approbation study [27], which demonstrate that medium-sized models (3–8B parameters) are capable of delivering competitive vulnerability detection performance with acceptable inference time. This makes them suitable for integration into CI/CD pipelines without transmitting source code to external services or relying on proprietary APIs.

The proposed framework enables the following tasks:

- assessing the complexity of vulnerability datasets and comparing them with other datasets;
- comparing the effectiveness of language models using various metrics, such as accuracy, F1-score, false positive rate (FPR), and execution time;
- comparing the effectiveness of different prompting strategies using the same set of metrics.

The objective of this work is to develop a tool for evaluating the effectiveness of language models in vulnerability detection tasks. To achieve this objective, the following tasks are addressed:

- designing and implementing a framework for running experiments with language models in source code vulnerability detection tasks;
- implementing an experiment configuration scheme that supports binary classification (classification of a specific vulnerability type and classification based on the presence or absence of a vulnerability) as well as multiclass classification;
- validating the functionality of the developed framework.

## II. Methodology and Tools

During the development of the framework, we followed object-oriented programming (OOP) principles and the SOLID design principles to achieve flexible interaction with different datasets and language models [28]. This approach allows users to extend the framework for their own research purposes without modifying its core components.

### A. Tools Used

**Programming language and libraries.** The framework is implemented in the Python programming language. Python provides convenient tooling for interacting with language models and for conducting statistical analyses, making it well suited for experimental research in this domain.

The following auxiliary libraries were used:

- scikit-learn (sklearn) - a comprehensive machine learning toolkit for Python. It provides a wide range of algorithms for classification, regression, clustering, data preprocessing, and model evaluation, all implemented within a unified API [29].
- Hugging Face - a company and open-source community that provides tools, datasets, and a model repository (Model Hub) to facilitate the development and sharing of state-of-the-art natural language processing and multimodal models [30].
- Transformers - a Python library developed by Hugging Face that implements hundreds of pretrained transformer architectures (e.g., BERT, GPT) for tasks such as text generation, classification, sequence labeling, and others, with convenient support for fine-tuning and model interaction [30].

**Hardware and Software Environment (Testbed Configuration).** For framework validation, cloud servers with the following specifications were rented: Intel® Xeon® Silver 4214R @ 2.20 GHz CPU, 87 GB of RAM, and an NVIDIA A100 GPU with 40 GB of video memory.

A preconfigured Ubuntu 22.04 image was used, with Docker already installed and integrated with NVIDIA drivers. CUDA version 12.8 was employed.

### B. Framework Architecture

The framework consists of multiple components, including interfaces, their implementations, datasets, and configuration modules. To describe the architecture of the designed application, UML diagrams are used.

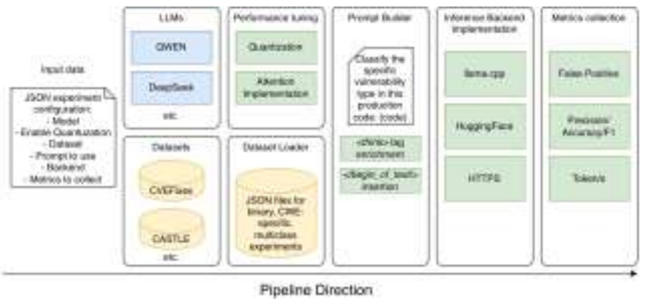
Input data
JSON experiment configuration: - Model - Enable Quantization - Dataset - Prompt to use - Backend - Metrics to collect
LLMs
QWEN
DeepSeek
etc.
Datasets
CVEFixes
CASTLE
etc.
Performance tuning
Quantization
Attention implementation
Dataset Loader
JSON files for binary, CWE-specific, multiclass experiments
Prompt Builder
Classify the specific vulnerability type in this production code: {code}
<think>-tag enrichment
<begin_of_text> insertion
Inference Backend implementation
llama.cpp
HuggingFace
HTTPS
Metrics collection
False Positive
Precision/ Accuracy/F1
Token/s
Pipeline Direction

Fig. 1. High level schema of framework pipeline

On Fig. 1 we present of how our framework is working and which key components are existed in it. In the following sections, each component of the framework is examined step by step.

The extensibility of the framework was validated in practice during an approbation study [27], in which the following were implemented within a unified architecture:

- support for four different vulnerability benchmarks with fundamentally different data structures;
- multiple families of language models (Qwen, DeepSeek, LLaMA, Gemma);
- various classification modes (binary, CWE-based, multiclass);
- separate prompting strategies, including reasoning-based modes.

These results confirm that adding new datasets, models, or evaluation metrics does not require modification of the framework core and can be achieved solely through the implementation of the corresponding interfaces.

**Dataset Loader.** The first component of the framework is the dataset loader used for conducting experiments. Since different datasets are provided in heterogeneous formats, it is necessary to implement a dedicated loading mechanism capable of handling such variability.

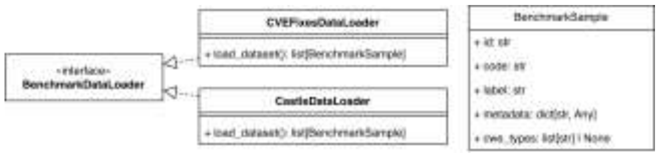
«interface»
BenchmarkDataLoader
CVEFixesDataLoader
+ load_dataset(): list[BenchmarkSample]
CastleDataLoader
+ load_dataset(): list[BenchmarkSample]
BenchmarkSample
+ id: str
+ code: str
+ label: str
+ metadata: dict[str, Any]
+ cwe_types: list[str] | None

Fig. 2. UML diagram of the dataset loader component.

As shown in Fig. 2, the dependency inversion principle is applied to allow the *BenchmarkRunner* class to interact with different datasets without requiring changes to the core framework code. To achieve this, the *BenchmarkDataLoader* interface was introduced with a *load_dataset* method. This design enables other users to implement their own dataset loaders without modifying the main project codebase.

Table II. Characteristics of the supported datasets.

| Dataset | Language | Samples count | CWEs count | Task Types |
|---|---|---|---|---|
| VulBench | C/C++ | ~250 | 10 | Binary, Multiclass |
| JitVul | C/C++ | ~1 758 | 91 | Binary, CWE |
| CASTLE | C/C++ | ~250 | 24 | Binary, CWE |
| CVEFixes | C/C++, Java, Python | >270 000 | 272 | Binary, Multiclass |

To unify the processing of heterogeneous datasets, a common JSON format is used. This format includes metadata (programming language, CWE identifier, task type) and an array of normalized examples. This approach has been previously validated and has demonstrated its suitability for comparative analysis of language models under different classification regimes. Table II represents the datasets used in this study.

**Experimental Data Protocol.** To ensure reproducibility and comparability of results, the framework employs a unified experimental protocol (contract) that was previously validated in the study [27], which focuses on the comparative analysis of language models for source code vulnerability classification.

All experiments are conducted in an inference-only mode, without model fine-tuning. This allows isolating the impact of model architecture, model size, and prompting strategy on the resulting evaluation metrics.

For each dataset, fixed-size subsets ranging from 50 to 200 samples are constructed, depending on the complexity of the task. In binary classification tasks targeting a specific vulnerability type, no more than 50 representative samples are used per experiment. Data samples do not overlap between experiments with different configurations.

Experiments are executed deterministically with fixed generation parameters, eliminating the influence of stochastic factors. For each model and prompt configuration, a single run is performed, as no training is involved.

Baseline reference points are provided by the results of a comparative analysis of modern LLMs (Qwen, DeepSeek, LLaMA, Gemma) obtained in the approbation study, which enables interpretation of the framework's output metrics in the context of the current state of the field.

**Prompt Generator for Language Models.** Different language models require different interaction patterns. For example, when working with models from the DeepSeek family, it is necessary to avoid using system prompts [31]. For Qwen3-family models, enabling reasoning behavior requires adding the *<think>* tag [32]. At the same time, it is desirable to use the same base prompt across experiments to maximize experimental rigor and comparability.

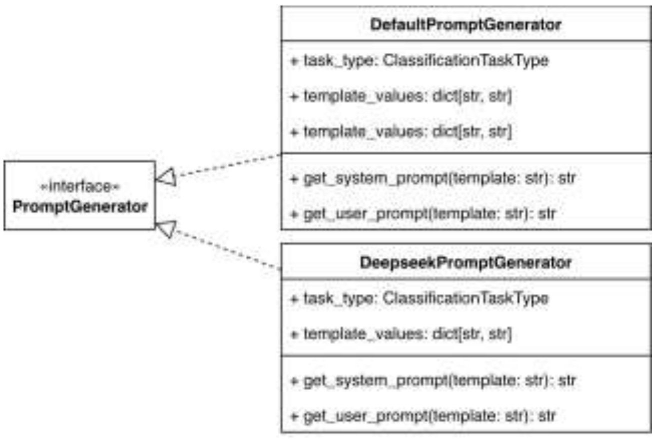
«interface»
PromptGenerator
DefaultPromptGenerator
+ task_type: ClassificationTaskType
+ template_values: dict[str, str]
+ template_values: dict[str, str]
+ get_system_prompt(template: str): str
+ get_user_prompt(template: str): str
DeepseekPromptGenerator
+ task_type: ClassificationTaskType
+ template_values: dict[str, str]
+ get_system_prompt(template: str): str
+ get_user_prompt(template: str): str

Fig. 3. UML class diagram of the prompt generation component.

To address this, we introduced an additional abstraction: the *PromptGenerator* interface, which defines the *get_system_prompt* and *get_user_prompt* methods which is demonstrated in Fig. 3. We implemented a default prompt generator, as well as specialized generators for DeepSeek and Qwen models with reasoning support. Framework users can also provide custom implementations to support model-specific requirements or experiment-specific prompting strategies.

It is important to note that different classification modes require different prompt construction strategies. To formalize this, we implemented the *ClassificationTaskType* enum, which includes the following modes: binary vulnerability classification, binary classification of a specific vulnerability type, classification in the custom-task mode, and multiclass classification.

**Interaction with the Language Model.** Interaction with a language model consists of several stages. First, the language model itself is loaded. During the loading process, additional configuration is applied, including setting the temperature, limiting the maximum number of generated tokens, and allocating the model to the available computational resources (a single GPU, multiple GPUs, or the CPU).

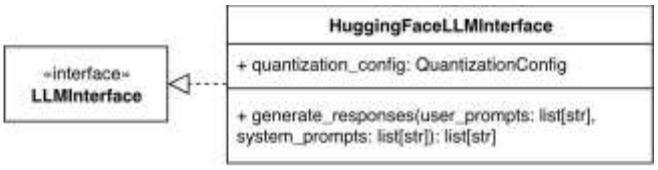
«interface»
LLMInterface
HuggingFaceLLMInterface
+ quantization_config: QuantizationConfig
+ generate_responses(user_prompts: list[str], system_prompts: list[str]): list[str]

Fig. 4. UML diagram of the central framework interface.

In our implementation, we use the Hugging Face library to interact with language models. This choice is motivated by the simplicity and flexibility of its APIs for loading, configuring, and running language models. The Fig. 4 shows which interface method is implemented.

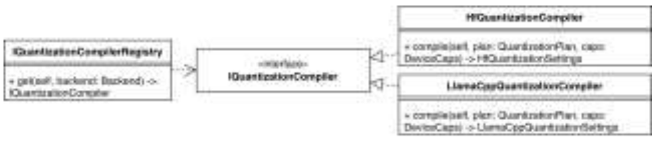
IQuantizationCompilerRegistry
+ get(self, backend: Backend) -> IQuantizationCompiler
«interface»
IQuantizationCompiler
HfQuantizationCompiler
+ compile(self, plan: QuantizationPlan, caps: DeviceCaps) -> HfQuantizationSettings
LlamaCppQuantizationCompiler
+ compile(self, plan: QuantizationPlan, caps: DeviceCaps) -> LlamaCppQuantizationSettings

Fig. 5. UML diagram of the quantizer factory for language models.

During model loading, quantization can also be applied. Quantization reduces the computational and memory requirements of a language model, at the cost of a potential decrease in accuracy. For example, on resource-constrained machines, 4-bit quantization can be used, or part of the computation can be offloaded to the CPU. Moreover, in future other LLM backends might be implemented. We take this in count, and created *IQuantizationCompilerRegistry* for different setups. Fig. 5 illustrates how flexibility is achieved.

Different model families require different quantization approaches. Moreover, quantization configuration is conceptually separate from the logic used to interact with the language model itself. For this reason, the logic for constructing quantization configurations was extracted into a dedicated factory component.

**Response Handler.** Different vulnerability datasets exhibit different response structures. For example, the VulDetectBench dataset focuses not only on vulnerability identification and classification [33], but also on locating specific lines of code that are directly related to a vulnerability. Other benchmarks investigate how language models can remediate vulnerabilities. All these scenarios may require different interpretations of the outputs generated by a language model.

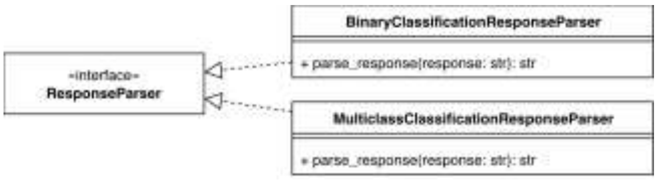
«interface»
ResponseParser
BinaryClassificationResponseParser
+ parse_response(response: str): str
MulticlassClassificationResponseParser
+ parse_response(response: str): str

Fig. 6. UML class diagram of the language model response handler.

To enable users to add custom benchmark scenarios, we introduced an additional abstraction, the *IResponseParser* interface on Fig. 6. Two primary response processing implementations were developed: one for binary classification (including classification by vulnerability type) and another for multiclass classification.

It is important to emphasize that, depending on the user-defined task, the output produced by a language model may vary significantly. Therefore, it is crucial to specify a clear and well-defined response format in the prompt that the model is expected to follow. Otherwise, ambiguous or loosely constrained outputs may lead to incorrect or unreliable results.

**Experiment Plan Configuration.** In the proposed framework, running a specific experiment implies defining an execution plan. The experiment plan is stored in a structured json format and consists of several components:

- Dataset specification. This can be a file path or any other value that uniquely identifies a dataset. Client code may define custom logic for handling this value by implementing a compatible dataset loader interface.
- Language models and their configurations. An experiment plan may include multiple language models with different configurations. The only requirement is that the configuration specified in the plan is compatible with the corresponding inference interface implementation.
- Prompts for language model inference. One common research task is to evaluate the effectiveness of different prompting strategies, such as zero-shot, one-shot, chain-of-thought, and others. By specifying multiple prompts, a single experiment run can be used to compare their relative effectiveness. An important detail is that prompts defined in the plan may include template strings, which must be correctly processed by the corresponding prompt handler.
- Output metrics. The set of resulting metrics may vary depending on the task. For example, in non-classification scenarios where the model outputs code fragments, metrics such as recall or vector-based similarity between the reference code and the generated code may be required. Therefore, a specific set of metrics can be associated with each prompt, since the choice of metrics directly depends on the task posed to the language model. The list of metrics to be computed is defined by the user. All existing configuration examples are available for review and extension via the link provided in the abstract.

**Computation Optimization.** To enable faster and more efficient experimentation, several optimization techniques were applied. One of them, mentioned earlier, is language model quantization. Quantization is the process of mapping continuous model parameters (weights and activations), typically represented as high-precision floating-point numbers, into lower-precision discrete formats (e.g., 8-bit integers). By reducing numerical precision, quantization decreases model size and accelerates inference, often with only a minor loss in accuracy.

Another optimization integrated into the framework is the *FlashAttention* v3 library. *FlashAttention* accelerates large language model inference by fusing several stages of the standard attention computation-namely, query–key dot product calculation, softmax application, and multiplication by values-into a single GPU kernel operating on small input "tiles" [34]. This significantly reduces the number of reads and writes to global memory.

This tiling strategy also enables more efficient use of GPU shared memory and registers, reducing latency and mitigating memory bandwidth bottlenecks that typically slow down quadratic-time attention algorithms. As a result, *FlashAttention* provides a more compact and computation-local implementation that delivers substantial speedups compared to native attention kernels, especially for long sequences, making it a general-purpose accelerator for both inference and training of large transformer-based models.

**Framework Infrastructure.** During the framework implementation, we repeatedly encountered situations where the same code functioned correctly on Linux systems but caused troubles on Windows. For example, problems arose when installing certain critical dependencies on Windows.

Another challenge was the need to install a substantial number of auxiliary drivers and libraries. Moreover, system-level libraries must be compatible with Python packages. For instance, installing *PyTorch* requires a specific version of NVIDIA CUDA.

To achieve maximum cross-platform compatibility and simplify the installation of supporting dependencies, we adopted Docker.

We developed a custom Docker image that installs all required system and Python libraries and performs compilation of *FlashAttention*. This image is based on an openly available base image: *nvcr.io/nvidia/cuda:12.8.1-devel-ubuntu24.04*.

Once built, the image can be uploaded to a third-party Docker image registry and reused to quickly launch experiments without the need to install all associated software from scratch.

It is important to note that the host machine running the container must have minimal configuration to enable integration between Docker and NVIDIA drivers. For framework testing, we used cloud servers whose provider already supplied a preconfigured Ubuntu image with Docker and NVIDIA CUDA integration.

Additionally, we provided Docker Compose files to simplify experiment execution. Docker Compose is used to create persistent cache storage (to avoid re-downloading models when launching new containers), define required environment variables, and enable the necessary Docker capabilities.

## III. RESULTS

### A. Framework Validation

After configuring the testbed described above, we successfully executed the framework using test experiment configurations.

For a more thorough validation of the implemented framework, we conducted a comparative evaluation of language models across several key parameters:

- Classification metrics: accuracy, precision, recall, F1-score, and specificity (for both binary and multiclass classification), as well as false negative rate, which is used to assess the risk associated with applying language models in vulnerability detection tasks.
- Execution time of each language model.

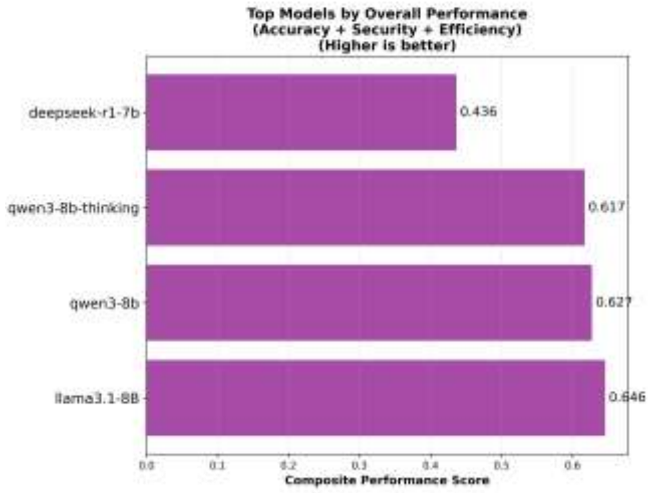

Fig. 7. Comparison of models based on overall performance metrics.

To compare the overall effectiveness of the models, we combined accuracy-related metrics, the number of false negatives, and model execution time across all evaluated datasets. The outcome of such comparison is demonstrated on Fig. 7.

Table III. Models' evaluation comparison table

| Model with size | Dataset | Experiment type | Precision | Recall | F1 | False negatives |
|---|---|---|---|---|---|---|
| Deep Seek R1 7B | CASTLE | binary | 0.68 | 0.65 | 0.66 | 43 |
| | | cwe | 0.10 | 0.60 | 0.17 | 16 |
| | | multi | - | - | - | - |
| | CVEFixes | binary | 0.50 | 0.21 | 0.29 | 87 |
| | | cwe | 0.01 | 0.17 | 0.02 | 1 |
| | | multi | - | - | - | - |
| | JitVul | cwe | **0.53** | **0.62** | **0.57** | **78** |
| | | multi | - | - | - | - |
| | VulBench | binary | 0.27 | 0.32 | 0.28 | 35 |
| | | multi | - | - | - | - |
| Llama 3.1 8B | CASTLE | binary | 0.66 | **0.93** | **0.77** | **9** |
| | | cwe | 0.16 | **0.78** | 0.27 | **9** |
| | | multi | - | - | - | - |
| | CVEFixes | binary | 0.50 | **0.43** | **0.46** | **22** |
| | | cwe | 0.02 | 0.50 | 0.03 | 0 |
| | | multi | - | - | - | - |
| | JitVul | cwe | 0.56 | 0.85 | 0.68 | 16 |
| | | multi | - | - | - | - |
| | VulBench | binary | 0.28 | **0.58** | **0.36** | **4** |
| | | multi | - | - | - | - |
| Qwen 3 8B | CASTLE | binary | **0.75** | 0.62 | 0.68 | 45 |
| | | cwe | 0.23 | 0.43 | 0.30 | 23 |
| | | multi | - | - | - | - |
| | CVEFixes | binary | 0.50 | 0.11 | 0.18 | 117 |
| | | cwe | 0.00 | 0.00 | 0.00 | 0 |
| | | multi | - | - | - | - |
| | JitVul | cwe | 0.32 | 0.06 | 0.09 | 170 |
| | | multi | - | - | - | - |
| | VulBench | binary | **0.37** | 0.30 | 0.31 | 43 |
| | | multi | - | - | - | - |
| Qwen 3 8B with thinking | CASTLE | binary | 0.74 | 0.62 | 0.67 | 45 |
| | | cwe | **0.25** | 0.45 | **0.32** | 22 |
| | | multi | - | - | - | - |
| | CVEFixes | binary | 0.50 | 0.10 | 0.16 | 121 |
| | | cwe | 0.00 | 0.00 | 0.00 | 0 |
| | | multi | - | - | - | - |
| | JitVul | cwe | 0.19 | 0.05 | 0.07 | 196 |
| | | multi | - | - | - | - |
| | VulBench | binary | 0.33 | 0.29 | 0.29 | 40 |
| | | multi | - | - | - | - |

Also, we provide a full list of results in Table III, which reflects of how each model behave. The bold values are top values for this experiment type in each dataset.

To construct the plots, we interpreted and processed the framework's output using Jupyter Notebook. The code used for this analysis is publicly available at: https://github.com/vodkar/llm4codesec-llm-benchmark/blob/main/LLM_Vulnerability_Detection_Analysis.ipynb

### B. *Metric Interpretation and Aggregation*

The framework is designed to aggregate results from a large number of heterogeneous experiments. Therefore, during the validation phase, metric values are averaged across datasets and task types (binary classification, CWE-based classification, and multiclass classification).

The approach to metric aggregation and visual analysis (including heatmaps and summary tables) follows the methodology applied in the approbation study [27], in which models were compared in terms of accuracy, F1-score, false negative rate (FNR), and inference time.

The objective of the present work is not to perform statistical hypothesis testing to establish the superiority of specific models, but rather to demonstrate that the framework reliably reproduces stable performance differences between models that were previously identified in an independent comparative study.

## IV. Conclusion

This work presents the design and implementation of a software framework LLM4CodeSec for evaluating the effectiveness of large language models in source code vulnerability detection tasks. The proposed framework provides a unified and reproducible infrastructure for conducting experimental studies with various language models, datasets, prompting strategies, and evaluation metrics.

The scientific novelty of this work lies in the development of an extensible tool tailored to evaluation of large language models in vulnerability detection tasks, which enables consistent and comparable analysis of results across different experimental configurations without modifying the system core. Unlike existing approaches, the framework supports multiple classification modes (binary classification, vulnerability-type classification, and multiclass classification) and accounts for both accuracy-related metrics and risk-oriented measures, including the false negative rate.

The practical significance of the obtained results is demonstrated by the applicability of the developed framework to real-world software development processes, including integration scenarios within CI/CD pipelines. The use of locally deployed language models for inference reduces the risk of source code leakage and lowers costs associated with relying on external proprietary services.

The framework's functionality has been validated through experimental evaluation on several widely used source code vulnerability benchmarks. The obtained results are consistent with previously published studies and demonstrate that the framework accurately reflects differences between language models in terms of detection accuracy, inference speed, and reliability of vulnerability identification.

Thus, the developed framework can be regarded as a general-purpose tool for both scientific research and applied experimentation in the field of source code security analysis using large language models, and it provides a solid foundation for the further development of intelligent static analysis methods.

### A. *Directions for Future Work*

The low accuracy observed in multiclass classification tasks is consistent with the results of the approbation study, which showed that when the number of classes is large (e.g., hundreds of CWE categories in the CVEFixes dataset), even state-of-the-art LLMs exhibit a substantial degradation in performance. At the same time, for datasets with a limited number of classes (e.g., VulBench), significantly higher accuracy can be achieved. This highlights the potential of hierarchical CWE-based classification and indicates the need to incorporate additional domain knowledge about vulnerability semantics, such as through fine-tuning or retrieval-augmented generation (RAG), which are planned directions for future research.

Inference with some language models required considerable execution time, even when using FlashAttention and carefully tuned configurations. That inference performance can be further improved through additional optimization techniques. For example, the Hugging Face ecosystem recommends using the Dataset abstraction to enable more efficient interaction with models during inference.

In addition, it is important to extend the framework to support alternative backends for running language models, such as Llama.cpp, which provides a convenient and efficient approach for interacting with large language models, particularly in resource-constrained environments.

## References

[1] A. A. Zakharov and K. I. Gladkikh, Characteristics and trends of zero-day vulnerabilities in open-source code. International Russian Automation Conference, 2024, pp. 498–502. doi: 10.1109/rusautocon61949.2024.10694228.

[2] The MITRE Corporation, CVE, "Metrics" [Online]. Available: https://www.cve.org/about/Metrics.

[3] J. Leinonen et al., "Comparing code explanations created by students and large language models," ITiCSE 2023: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, pp. 124–130, Jun. 2023, doi: 10.1145/3587102.3588785.

[4] B. Berabi, A. Gronskiy, V. Raychev, G. Sivanrupan, V. Chibotaru, and M. Vechev, "DeepCode AI Fix: fixing security vulnerabilities with large language models," *arXiv.org*, Feb. 19, 2024. https://arxiv.org/abs/2402.13291

[5] H. Y. Lin, C. Liu, H. Gao, P. Thongtanunam, and C. Treude, "CodeReviewQA: the code review comprehension assessment for large language models," *arXiv.org*, Mar. 20, 2025. https://arxiv.org/abs/2503.16167

[6] Y. Yu et al., "Fine-tuning large language models to improve accuracy and comprehensibility of automated code review," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 1, pp. 1–26, Sep. 2024, doi: 10.1145/3695993.

[7] D. Namiot, ""What LLM knows about cybersecurity." International Journal of Open Information Technologies 13.7 (2025): 37-46. (in Russian).

[8] H. Xu *et al.*, "Large language models for cybersecurity: a systematic literature review," *ACM Transactions on Software Engineering and Methodology*, Sep. 2025, doi: 10.1145/3769676.

[9] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, *Transformer-based language models for software vulnerability detection*. 2022, pp. 481–496. doi: 10.1145/3564625.3567985.

[10] Z. Li *et al.*, "VulDeePecker: a deep learning-based system for vulnerability detection," *Internet Society*, Jan. 2018, doi: 10.14722/ndss.2018.23158.

[11] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection," *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC '22)*, pp. 481–496, Dec. 2022, doi: 10.1145/3564625.3567985.

[12] N. Ziems and S. Wu, "Security vulnerability detection using deep learning natural language processing," *IEEE Conference on Computer Communications Workshops*, pp. 1–6, May 2021, doi: 10.1109/infocomwkshps51825.2021.9484500.

[13] K. Gladkikh and A. A. Zakharov, *Approach to forming vulnerability datasets for fine-tuning AI agents*. 2025 International Russian Smart Industry Conference (SmartIndustryCon), 2025, pp. 771–776. doi: 10.1109/smartindustrycon65166.2025.10986048.

[14] Z. Gao, H. Wang, Y. Zhou, W. Zhu, and C. Zhang, "How far have we gone in vulnerability detection using large language models," arXiv.org, Nov. 21, 2023. https://arxiv.org/abs/2311.12420

[15] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, "LLMs in Software Security: A survey of Vulnerability Detection Techniques and Insights," ACM Computing Surveys, vol. 58, no. 5, pp. 1–35, Sep. 2025, doi: 10.1145/3769082.

[16] T. Chen, Challenges and opportunities in integrating LLMs into continuous integration/continuous deployment (CI/CD) pipelines. 5th International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT), 2024, pp. 364–367. doi: 10.1109/ainit61980.2024.10581784.

[17] W. Cheng, K. Sun, X. Zhang, and W. Wang, "Security attacks on LLM-based code completion tools," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 22, pp. 23669–23677, Apr. 2025, doi: 10.1609/aaai.v39i22.34537.

[18] S. Jenko, N. Mündler, J. He, M. Vero, and M. Vechev, "Black-box adversarial attacks on LLM-based code completion," *arXiv (Cornell University)*, Aug. 2024, doi: 10.48550/arxiv.2408.02509.

[19] D. Noever, "Can large language models find and fix vulnerable software?," *arXiv.org*, Aug. 20, 2023. https://arxiv.org/abs/2308.10345

[20] A. Lekssays, H. Mouhcine, K. Tran, T. Yu, and I. Khalil, "LLMXCPG: Context-Aware vulnerability detection through Code Property Graph-Guided Large Language Models," arXiv.org, Jul. 22, 2025. https://arxiv.org/abs/2507.16585

[21] Z. Sun et al., "Ensembling large language models for code vulnerability detection: an Empirical evaluation," arXiv.org, Sep. 16, 2025. https://arxiv.org/abs/2509.12629

[22] M. A. Hannan, R. Ni, C. Zhang, L. Jia, R. Mangal, and C. S. Pasareanu, "On the Difficulty of Selecting Few-Shot Examples for Effective LLM-based Vulnerability Detection," arXiv.org, Oct. 2025, doi: 10.14722/last-x.2026.23025.

[23] W. Charoenwet, K. Tantithamthavorn, P. Thongtanunam, H. Y. Lin, M. Jeong, and M. Wu, "AgenticSCR: an autonomous agentic secure code review for immature vulnerabilities detection," arXiv.org, Jan. 27, 2026. https://arxiv.org/abs/2601.19138

[24] A. Zibaeirad and M. Vieira, "VULNLLMEVAL: A framework for evaluating large language models in software vulnerability detection and patching," *arXiv (Cornell University)*, Sep. 2024, doi: 10.48550/arxiv.2409.10756.

[25] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, *Transformer-Based Language Models for Software Vulnerability Detection*. Asia-Pacific Computer Systems Architecture Conference, 2022, pp. 481–496. doi: 10.1145/3564625.3567985.

[26] Y. Li *et al.*, "Everything you wanted to know about LLM-based vulnerability detection but were afraid to ask," *arXiv (Cornell University)*, Apr. 2025, doi: 10.48550/arxiv.2504.13474.

[27] K. I. Gladkikh and A. A. Zakharov, *Comparison of language models for source code vulnerability classification*. 2025 International Russian Automation Conference (RusAutoCon), 2025, pp. 779–784. doi: 10.1109/rusautocon65989.2025.11177346.

[28] A. Neyer, F. F. Wu, and K. Imhof, "Object-oriented programming for flexible software: example of a load flow," IEEE Transactions on Power Systems, vol. 5, no. 3, pp. 689–696, Jan. 1990, doi: 10.1109/59.65895.

[29] PedregosaFabian et al., "SciKit-Learn: Machine Learning in Python," Journal of Machine Learning Research, pp. 2825–2830, Nov. 2011, doi: 10.5555/1953048.2078195.

[30] T. Wolf *et al.*, "HuggingFace's transformers: state-of-the-art natural language processing," *arXiv.org*, Oct. 09, 2019. https://arxiv.org/abs/1910.03771

[31] D. Guo *et al.*, "DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning," *Nature*, vol. 645, no. 8081, pp. 633–638, Sep. 2025, doi: 10.1038/s41586-025-09422-z.

[32] *A. Yang et al., "QWEN3 technical report," arXiv.org, May 14, 2025. https://arxiv.org/abs/2505.09388*

[33] *Y. Liu et al., "VulDetectBench: evaluating the deep capability of vulnerability detection with large language models," arXiv.org, Jun. 11, 2024. https://arxiv.org/abs/2406.07595*

[34] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "FlashAttention-3: fast and accurate attention with asynchrony and low-precision," *arXiv.org*, Jul. 11, 2024. https://arxiv.org/abs/2407.08608

Gladkikh K. I., post graduate student of Information security methods and systems course at the Institute of mathematics and computer science at Tyumen State University, School of Computer Science, Tyumen State University. 625003, Tyumen, Volodarskogo street, 6. E-mail: boombarah@gmail.com

Zakharov A. A., doctor of technical sciences, professor, School of Computer Science, Tyumen State University. 625003, Tyumen, Volodarskogo street, 6. E-mail: a.a.zakharov@utmn.ru

Zakharova I. A., Candidate of Physical and Mathematical Sciences, Doctor of Pedagogical Sciences, Professor, School of Computer Science, Tyumen State University. 625003, Tyumen, Volodarskogo street, 6. E-mail: i.g.zakharova@utmn.ru