On the Compression of Language Models for Code: An Empirical Study on CodeBERT

Giordano d'Aloisio
DISIM Department
University of L'Aquila
L'Aquila, Italy
giordano.daloisio@univaq.it

Luca Traini
DISIM Department
University of L'Aquila
L'Aquila, Italy
luca.traini@univaq.it

Federica Sarro
Department of Computer Science
University College London
London, UK
f.sarro@ucl.ac.uk

Antinisca Di Marco
DISIM Department
University of L'Aquila
L'Aquila, Italy
antinisca.dimarco@univaq.it

***Abstract*—Language models have proven successful across a wide range of software engineering tasks, but their significant computational costs often hinder their practical adoption. To address this challenge, researchers have begun applying various compression strategies to improve the efficiency of language models for code. These strategies aim to optimize inference latency and memory usage, though often at the cost of reduced model effectiveness. However, there is still a significant gap in understanding how these strategies influence the efficiency and effectiveness of language models for code. Here, we empirically investigate the impact of three well-known compression strategies – knowledge distillation, quantization, and pruning – across three different classes of software engineering tasks: vulnerability detection, code summarization, and code search. Our findings reveal that the impact of these strategies varies greatly depending on the task and the specific compression method employed. Practitioners and researchers can use these insights to make informed decisions when selecting the most appropriate compression strategy, balancing both efficiency and effectiveness based on their specific needs.**

***Index Terms*—Language Models, Compression Strategies, Software Quality, Empirical Study**

I. Introduction

The growing adoption of transformer-based Language Models (LMs) has radically transformed the software engineering (SE) field in recent years. These models have been effectively applied to a wide array of SE tasks, including vulnerability detection [1], code summarization [2], and code search [3], consistently demonstrating superior performance over traditional approaches [4]. However, despite their impressive capabilities, the widespread adoption of these models is often hindered by practical challenges, particularly their high computational cost [5]. The deployment of LMs typically requires computations across millions or even billions of learned parameters, resulting in significant memory demands and high inference latency.

To address this issue, AI researchers have developed various strategies over the past decade to reduce the size and computational cost of LMs, including techniques such as knowledge distillation [6], quantization [7], and pruning [8]. These "*model compression*" strategies can reduce the memory demand of LMs and/or speed up their inference times, albeit often at the cost of reduced effectiveness (i.e., prediction correctness).

These strategies have recently begun to gain attention in the field of software engineering [9]–[11]. For instance, Shi et al. [9] applied knowledge distillation to drastically reduce the memory size of models of code. Wei et al. [11] have investigated the impact of quantization on code generation tasks regarding inference latency, memory consumption, and carbon footprint. However, despite these initial efforts, the broader impact of compression strategies on software engineering tasks remains largely unexplored. Most existing research has focused on specific compression strategies applied to individual SE tasks, making it difficult to determine whether specific strategies perform better than others on particular tasks. Additionally, it is unclear if the impact of different strategies varies by task or if they demonstrate similar behavior across different software engineering tasks.

In this study, we investigate the impact of different model compression strategies across three software engineering tasks: vulnerability detection (*code classification*), code summarization (*code-to-text generation*), and code search (*text-to-code recommendation*). We fine-tune a well-known language model for code, CodeBERT [12], on each of these tasks. Subsequently, we assess how three model compression strategies – namely, knowledge distillation, quantization, and pruning – affect (i) the effectiveness of the LM in performing the task, (ii) inference latency, and (iii) the model's memory size. Our results provide practitioners and researchers with guidelines on balancing the trade-offs between effectiveness and efficiency when selecting a model compression strategy.

In summary, the main contributions of this work are:

- An extensive empirical evaluation of the impact of three compression strategies, namely knowledge distillation, quantization, and pruning on inference time, model size and effectiveness of LMs fine-tuned on the vulnerability detection, code summarization and code search tasks;
- Insights for practitioners and researchers on the adoption of those compression strategies;
- A publicly available replication package of our empirical study [13].

II. Background and Related Work

A. Language Models in Software Engineering Tasks

Since its introduction by Vaswani et al. [14], the transformer architecture has become the de facto standard in language

modeling. Transformer-based language models have achieved state-of-the-art performance across numerous natural language processing tasks [15]–[17], and have recently gained traction in the software engineering field [4].

This trend has led to the development of several LMs specialized in code, such as CodeBERT [12], CodeT5 [18], and Codex [19]. These models are usually trained in a two-step process: (i) a pre-trained step using a self-supervised training objective aiming at providing the model with general knowledge about source code constructs and patterns, and (ii) a fine-tuning step aiming at tailoring the model for the software engineering task at hand.

Over the past few years, LMs for code have been fine-tuned to automate a wide range of SE tasks [4]. For example, CodeBERT – an encoder-only LM based on the BERT architecture [17] – has been widely and successfully applied to code summarization [20], vulnerability detection [21], and code search [12], among others [4].

B. Compression Strategies for Language Models

A major obstacle to the practical adoption of language models has always been their significant computational cost [4], [5]. To address this issue and increase their sustainability, the AI community has developed various strategies to reduce the memory demands and inference latency of these models. Nowadays, the three most popular model compression strategies are [9], [10]: *knowledge distillation* [6], *quantization* [7], and *pruning* [8].

1) Knowledge distillation: is a technique in which a smaller model (i.e., the "*student*") is trained to replicate the behavior of a larger, pre-trained language model (i.e., the "*teacher*"). This compression strategy results in a model that demands less memory and provides faster inference time. However, since student models usually have thinner and shallower neural networks, they often struggle to fully capture the knowledge embedded in the larger models. This limitation typically leads to a reduction in the LM capabilities compared to the teacher model [22].

2) Quantization: is a compression strategy that reduces the precision of the model's weights, converting them from full-precision (e.g., 32-bit floating point) to lower precision representations (e.g., 8-bit integer). Two broad categories of quantization strategies exist: *post-training quantization* and *quantization-aware training*. Post-training quantization generates a quantized model from an existing full-precision model without requiring additional training or fine-tuning. This strategy is a popular choice due to its low computational cost. However, it is more susceptible to quantization noise. In contrast, quantization-aware training involves training the model from scratch while incorporating simulated quantization operations to mitigate the quantization noise. Although this approach can produce a more effective model, its high training costs can make it often impractical.

3) Pruning: aims to make the language model more efficient by removing neural network weights considered less critical for the model's effectiveness [23]. Various pruning methodologies exist. For instance, *structured pruning* modifies the model's architecture by removing entire structures within the neural network, such as neurons, filters, or even layers [24]. Conversely, *unstructured pruning* targets individual weights [25], removing the less relevant ones (e.g., those close to zero). Pruning can be applied either to individual layers of the network (*layer-wise pruning*) [26], or across the entire model (*global pruning*) [27].

C. Compression Strategies in Software Engineering Tasks

Model compression strategies have recently gained relevance in the software engineering literature. Shi et al. [9] introduced *Compressor*, a knowledge distillation-based approach, evaluated on CodeBERT [12] and GraphCodeBERT [28] for two code classification tasks (i.e., vulnerability detection and clone detection). Their results demonstrate that *Compressor* can considerably accelerate inference time with minimal impact on model effectiveness. In a subsequent study [10], the authors expanded their approach to tackle energy consumption and carbon footprint concerns. Wei et al. [11] conducted an empirical evaluation of quantized models on code generation tasks, examining resource usage, carbon footprint, accuracy, and robustness. They found that quantization, under specific settings, can substantially enhance model efficiency with negligible accuracy or robustness trade-offs. Additionally, Sun et al. [2] explored dynamic inference as a method to speed up code completion. Despite these advancements, there remains a noticeable gap in comprehensive studies that systematically investigate the effects of different model compression strategies across various software engineering tasks.

With this study, we aim to fill this gap by performing an extensive empirical evaluation of the impact of the three most adopted compression strategies – knowledge distillation, model quantization, and model pruning – on the inference time, model size, and prediction's effectiveness of an LM fine-tuned for three widely adopted SE tasks – vulnerability detection, code summarization, and code search.

III. Empirical Study Design

The *goal* of this study is to analyse the impact of compression strategies on the *efficiency* (i.e., in terms of inference time and model size) and *effectiveness* (i.e., in terms of prediction correctness) of language models for code. Specifically, we investigate the impact of three compression strategies – knowledge distillation, pruning, and quantization – on CodeBERT models fine-tuned for three SE tasks: vulnerability detection, code summarization, and code search. We selected these tasks due to their relevance in software engineering [1]–[3], [29], [30] and because they span diverse categories, namely code classification (vulnerability detection), code-to-text generation (code summarization), and text-to-code recommendation (code search). We use CodeBERT as the reference language model due to its popularity in the software engineering literature [4] and its versatility in handling classification, generation, and recommendation tasks [12].

Our research is driven by the following research questions:

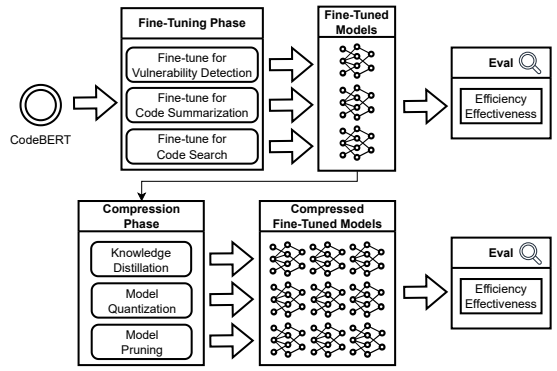


Fig. 1. Experimental Methodology

TABLE I
EVALUATION METRICS

	Task	Task Category	Metrics	
			Effectiveness	Efficiency
RQ$_1$	Vulnerability Detection	Code-Code Classification	Accuracy [31] F1 Score [32] MCC [33]	Inf. Time (sec.) Model Size (MB)
RQ$_2$	Code Summarization	Code-Text Generation	Bleu [34] BERTScore [35] SIDE [36]	
RQ$_3$	Code Search	Text-Code Search	MRR [37] MRR@1 [37] MRR@5 [37]	

RQ$_1$ *How do compression strategies impact the efficiency and effectiveness of models fine-tuned for vulnerability detection?*

RQ$_2$ *How do compression strategies impact the efficiency and effectiveness of models fine-tuned for code summarization?*

RQ$_3$ *How do compression strategies impact the efficiency and effectiveness of models fine-tuned for code search?*

To answer these RQs we carry out an empirical investigation based on the methodology described in the following. Figure 1 provides an overview of our experimental methodology. We first fine-tune CodeBERT for each SE task of interest and collect the corresponding effectiveness and efficiency metrics. Next, we apply each of the three compression strategies individually to produce compressed versions of the fine-tuned models. Finally, we compare the efficiency and effectiveness metrics of the compressed models with those of the original fine-tuned CodeBERT model to assess the impact of each compression strategy.

Table I shows the effectiveness and efficiency metrics used in our study. For effectiveness, we consider specific metrics depending on the software engineering task at hand. For instance, we use *Accuracy*, *F1 Score* and *MCC* to measure the effectiveness of the LM for the vulnerability detection (i.e., code classification) task. For efficiency, we focus on two aspects: inference time (in seconds) and model memory size (in MB). We collect inference time metrics for both CPU and GPU environments, as compression strategies are frequently used to adapt language models for constrained hardware setups, such as desktop computers without dedicated GPUs. The experimental process took around ten days of machine execution over a CentOS HPC cluster equipped with 32 Intel(R) Xeon(R) Gold 6140M CPUs and two Nvidia A100 and A30 GPUs.

TABLE II
DATASETS AND LM HYPER-PARAMETERS FOR EACH TASK

	Task	Dataset				Hyper Params.
		Name	Train	Val.	Test	
RQ$_1$	Vulnerability Detection	Devign [38]	21,854	2,732	2,732	Epochs: 5 Learning Rate: 2^{-5}
RQ$_2$	Code Summarization	CodeSearchNet (Java) [39]	164,923	5,183	10,955	Epochs: 10 Learning Rate: 5^{-5}
RQ$_3$	Code Search	CodeSearchNet (Python) [39]	251,820	9,604	19,210	Epochs: 2 Learning Rate: 2^{-5}

A. Software Engineering Tasks

1) Vulnerability Detection: In this task, the language model is prompted with a code function and asked to predict whether it contains a security vulnerability. The model produces a binary label indicating whether the code is vulnerable or not. We fine-tune and evaluate CodeBERT using the Devign dataset by Zhou et al. [38]. This dataset contains 27,318 C functions extracted from two popular open-source projects (QEMU and FFmpeg). Each function is accompanied by a label that denotes whether the code contains a security vulnerability or not.

2) Code Summarization: It is the task of automatically generating natural language summaries (namely comments) for code snippets. The language model is given a code function to produce code summary. We use a Sequence-to-Sequence (Seq2Seq) model, which includes CodeBERT as encoder layer and a six-layer Transformer as decoder layer. We fine-tune and evaluate CodeBERT on the CodeSearchNet dataset [39]. This dataset contains 2 million code-comment pairs extracted from open-source repositories written in different languages, such as Python, Javascript, Ruby, Go, Java, and PHP. For this task, we focus on the Java programming language for a total of 181,061 code-comment pairs.

3) Code Search: Given a natural language sentence (i.e., comment), this task aims to retrieve semantically relevant code snippets. This is performed by first producing the embeddings of the comment and the code snippets. Then the semantically similar code snippets are ranked using the embedding similarity of sentence and code (through inner dot product). We fine-tune and evaluate CodeBERT using the CodeSearchNet dataset for the Python programming language, which contains a total of 280,634 code-comment pairs [39].

Following previous works [9], [10], we reuse the pipeline provided by the CodeXGLUE benchmark [40] for all the aforementioned tasks. CodeXGLUE is a popular benchmark which provides data and code for fine-tuning and evaluating LMs on different code-related tasks. We extended the pipeline by adding the code required to compress the LMs using knowledge distillation, quantization, and pruning.

For evaluation, we use the train, validation, and test splits as well as the same model hyper-parameters provided by the CodeXGLUE benchmark. The size (in terms of number of items) of each dataset split and the model's hyper-parameters are reported in Table II.

B. Compression Strategies

1) Knowledge Distillation: Knowledge distillation is a computationally complex task, as it typically involves re-training the "student" model from scratch. Given the extensiveness of our experimental setup, we opted not to retrain a distilled model ourselves. Instead, we utilized a pre-trained distilled BERT model, namely DistilBERT [41], which we fine-tuned for the specific SE task of interest. For vulnerability detection and code search tasks, we directly fine-tuned DistilBERT. For code summarization, we used DistilBERT as the encoder layer of a Seq2Seq model, which we then fine-tuned using the same procedure.

2) Quantization: Model Quantization reduces a model's size by changing its weights' precision from the standard `float32` to less precise data types. We apply *post-training quantization* (see Section II) implemented by the Hugging Face's *optimum-quanto* library.[1] We chose this implementation because it requires minimum configuration and supports CPU and GPU. We tested three different applications of post-training quantization by reducing the weights to `int4`, `int8`, and `float8` (as, by the time we ran our experiments, the library allowed those three reductions). Following the library's documentation, we first quantized the fine-tuned model and then calibrated its activation functions using the validation set. Finally, following again the documentation, we *frozen* the quantized weights before storing the model.

3) Pruning: In our experiments, we analyse the *unstructured global pruning* implemented by the PyTorch library.[2] We have chosen this implementation because it does not change the internal structure of a network; hence, it does not require the re-training of the model after its application. Following the work of Gordon et al. on pruning BERT models [42], for each task, we prune the weights of all the linear layers of the network using the *L1 norm* as the selection strategy. The *L1 norm* strategy uses the sum of the absolute values of a vector's components to determine the importance of structures within a neural network [43]. We analyse three different configurations of pruning: one in which we prune the 20% of weights in the whole network (Prune 0.2), one in which we prune the 40% (Prune 0.4), and one in which we prune the 60% (Prune 0.6). As done for quantization, we first fine-tuned the models for each task and then pruned them. Finally, before storing the model, we removed the internal copy of the original weights created by the PyTorch library after the application of pruning.

C. Efficiency Metrics

For efficiency, we indicate the performance of a model in terms of the time required to give a prediction (i.e., inference time) and the memory size of a model.

1) Inference time: We measure the inference time for each batch of the testing set. Following the CodeXGLUE benchmark, we consider a batch size of 64 test instances for each task. The inference time is measured both on CPU and GPU (CUDA). For CPU, we use the `time` Python function, while for GPU, we use the `Event` class provided by PyTorch. Moreover, before computing the inference time on the GPU, we perform a series of warm-up iterations to avoid inconsistencies in the results. In addition, we apply GPU synchronization after each inference iteration.

To assess the impact of compression strategies, we compare the inference time measurements of each compressed model with those of the original fine-tuned CodeBERT model. To ensure rigor, we follow performance engineering best practices [44]–[47], specifically the approach proposed by Kalibera and Jones [48], to build confidence intervals for the relative change in measurements statistics. Specifically, we construct the confidence interval for the median relative change in inference time using bootstrapping with 10,000 iterations, involving random resampling with replacement [49]. The main advantage of this technique, compared to others such as the Wilcoxon test [50], is that it provides a clear and rigorous account of the inference time change and the associated uncertainty [48], [49], [51]. For example, this method allows us to state that a compressed model is faster than the original CodeBERT model by -30%$\pm$2% with 95% confidence. We consider a difference to be statistically significant if the confidence interval is not greater than the percentage change.

2) Model size: To measure the model size, we save the model state in memory using the `save` function provided by PyTorch and then calculate its size using the `getsize` Python function. The value the function returns is converted to megabytes (MB). Although we performed this process on both CPU and GPU, as expected, the models' size remained unchanged between the two environments. Therefore, we do not differentiate between CPU and GPU when reporting the models' size in Section IV.

D. Effectiveness Metrics

For effectiveness, we assess how good the predictions of a model are for a specific task. Given the heterogeneity of tasks involved in our evaluation, we used different metrics depending on the SE task being analyzed (see Table I).

1) Vulnerability Detection: We use the *Matthews Correlation Coefficient (MCC)* as it considers all quadrants of the classification matrix (i.e., it gives a comprehensive overview of the model's performance). It has been shown to be a reliable measure when handling imbalanced data, as is often the case for vulnerability prediction [33], [52]–[54]. MCC is defined as a correlation factor between the true and predicted labels. It ranges from -1 to 1, where -1 means the model gives opposite predictions, 0 means random predictions, and 1 means perfect predictions. In our study, we also report on *F1 Score* [32] and *Accuracy* [31] for compatibility with respect to previous work. The F1 Score is defined as the harmonic mean between

[1]https://github.com/huggingface/optimum-quanto

[2]https://pytorch.org/tutorials/intermediate/pruning_tutorial.html

Precision and Recall. Accuracy is defined as the number of correct predictions over the whole predictions of a model. Both F1 Score and Accuracy values range from 0 to 1, where 1 is the best value. Although the use of Accuracy is deprecated for problems suffering from data imbalance [54], we include this measure in our analysis for completeness as it is the metric employed by the CodeXGLUE benchmark. Still, we discourage its use in practice as done in previous work [54].

2) Code Summarization: We employ three metrics that assess different aspects of the quality of a generated text [55]. Bleu is the metric employed in the CodeXGLUE benchmark and is a standard metric adopted in natural language translation and, generally, text generation tasks [34]. It computes how similar a generated text is with respect to a reference baseline by comparing overlapping n-grams (contiguous sequences of n words) between the generated and reference texts. It is a metric of *summary-summary text similarity*, but has been criticised for not considering the semantic similarity between two texts [11], [55]. For this reason, we extended the evaluation by including two additional metrics. BERTScore evaluates the quality of a generated text by comparing the similarity between the BERT embeddings of the generated text and the reference baseline [35]. It is a metric of *summary-summary semantic similarity* [55]. Finally, SIDE is a metric based on contrastive learning that measures how good a generated explanation is for a given code snippet without considering a reference baseline [36]. It is specific for code summarization tasks and is a metric of *summary-code semantic similarity* [55]. All metrics range between 0 and 1, where 1 is the optimum value.

3) Code Search: We employ three versions of the Mean Reciprocal Rank (MRR) score [37]. We adopt this metric because it is used in the CodeXGLUE benchmark and is by far the most commonly adopted metric in code search [56]. MRR measures the average of the reciprocal ranks of the correct results for a set of code comments. The reciprocal rank for a single code comment is defined as the inverse of the rank position where the correct corresponding code appears in the list of retrieved results. In addition, we include two variations of MRR: MRR@1, which measures the proportion of code comments where the correct code appears in the first position, and MRR@5, which calculates the mean reciprocal rank based on the top five results. These metrics all range from 0 to 1, with 1 representing the optimal score.

IV. Empirical Study Results

In this section, we report the result of our empirical evaluation. For each RQ, we discuss the impact of each compression strategy relative to the model's efficiency and effectiveness, as well as the trade-off between these two aspects.

Table III shows the results for each RQ. On each sub-table, the first row reports the results of the plain CodeBERT model (i.e., without compression), while the remaining rows show the percentage variations provided by each compression strategy.[3] For inference time, we also report the confidence interval for the change using the Kalibera and Jones approach [49] (see Section III-C for details). Non-statistically significant changes are marked with an asterisk (*). For each considered efficiency or effectiveness metric, the best values are highlighted in **bold**, while the worst values are underlined. We also present scatter plots showing the trade-off between effectiveness (y-axis) and efficiency (x-axis) for each RQ in Figure 2.

A. RQ_1 Results - Vulnerability Detection

We analyse the impact of compression strategies from various aspects: inference time, model size, and vulnerability detection effectiveness. Additionally, we investigate the trade-offs involved among these metrics.

1) Inference Time: As reported in Table IIIa, all compression strategies change the inference time of the vulnerability detection model with statistical significance. However, the impact varies widely depending on the specific compression strategy and the hardware environment. On the CPU, for example, the most aggressive configuration of pruning (0.6) leads to the highest speed-up across all the strategies, with an inference time reduction of -67.9% compared to the plain CodeBERT model. Interestingly, less aggressive forms of pruning (0.2 and 0.4) lead to the opposite outcome, with an inference slow-down of $+15.5\%$ and $+18.8\%$, respectively. All pruning configurations also negatively affect inference time in GPU environments, with slow-downs of up to $+9.1\%$. A possible explanation for this behavior could be identified in the lower ability of GPU to handle sparse-matrix multiplications [26]. These results of model pruning suggests that this strategy requires specific hardware and configurations to improve the inference time of vulnerability detection models; however, when these conditions are met, the benefits can be substantial.

From Table IIIa, we also observe that quantization negatively impacts inference time in all configurations, resulting in the highest slowdowns across all compression strategies in both CPU and GPU environments. `int4` quantization, in particular, causes the most significant slowdowns, with increases of $+133.5\%$ on the CPU and $+201.6\%$ on the GPU. These negative results could be explained by the fact that CPUs and GPUs are heavily optimized for full-precision floating-point operations (e.g., 32-bit), hence quantized models may not fully take advantage of these optimizations and experience an inference slowdown [9], [57]–[59].

Knowledge Distillation, on the other hand, consistently and significantly improves inference time across both CPU and GPU environments. It ranks as the second-best strategy (after Pruning 0.6) on the CPU, with a reduction of -39.8% in inference time, and as the best strategy on the GPU, with a speed-up of -47.7%.

[3]For inference time, the first row reports the median of measurements across batches of the plain CodeBERT model. The remaining rows show the percentage variations in the median inference time provided by each compression strategy, computed using the Kalibera and Jones approach [49].

TABLE III
RQs 1-3: EFFICIENCY AND EFFECTIVENESS OF ORIGINAL AND COMPRESSED CODE MODELS FOR EACH OF THE SE TASKS.

Compression Method	Efficiency			Effectiveness		
	CPU Inf. Time	GPU Inf. Time	Model Size	Accuracy	F1	MCC
None	15.902 (sec)	0.011 (sec)	499 (MB)	0.630	0.541	0.247
Know. Distil.	-39.8% ± 2.7%	**-47.7% ± 0.6%**	-48.8%	-2.2%	**+3.1%**	-10.1%
Pruning (0.2)	+15.5% ± 6.0%	+9.1% ± 2.0%	0.0%	-4.4%	-41.0%	-11.3%
Pruning (0.4)	+18.8% ± 7.2%	+6.2% ± 1.7%	0.0%	-7.3%	-61.0%	-20.6%
Pruning (0.6)	**-67.9% ± 1.4%**	+7.3% ± 1.6%	0.0%	-5.9%	-49.2%	-18.2%
Quantization (float8)	+41.9% ± 16.3%	+98.3% ± 3.2%	-51.4%	**0.0%**	-1.1%	**+0.4%**
Quantization (int8)	+102.2% ± 14.4%	+107.4% ± 5.1%	-51.4%	-0.5%	-0.9%	-2.4%
Quantization (int4)	+133.5% ± 26.3%	+201.6% ± 4.8%	**-59.3%**	-1.6%	-4.4%	-8.5%

(a) RQ_1: Vulnerability Detection

Compression Method	Efficiency			Effectiveness		
	CPU Inf. Time	GPU Inf. Time	Model Size	Bleu	BERTScore	SIDE
None	157.369 (sec)	23.692 (sec)	707 (MB)	18.791	0.888	0.871
Know. Distil.	-39.8% ± 7.8%	**-2.2% ± 4.9%***	-33.0%	-42.3%	-6.1%	-70.6%
Pruning (0.2)	**-45.3% ± 4.9%**	+9.8% ± 1.9%	0.0%	-4.3%	-0.1%	**+0.4%**
Pruning (0.4)	+24.7% ± 19.8%	+121.9% ± 13.1%	0.0%	-66.6%	-17.7%	-42.4%
Pruning (0.6)	+183.5% ± 41.9%	+419.3% ± 29.5%	0.0%	-93.4%	-58.4%	-93.0%
Quantization (float8)	-20.3% ± 7.9%	+14.0% ± 2.2%	-42.0%	**+0.4%**	**0.0%**	+0.1%
Quantization (int8)	-27.2% ± 6.7%	+6.2% ± 2.3%	-42.0%	-0.3%	**0.0%**	+0.0%
Quantization (int4)	-17.9% ± 7.0%	+29.1% ± 2.5%	**-51.9%**	-2.0%	-0.1%	+0.2%

(b) RQ_2: Code Summarization

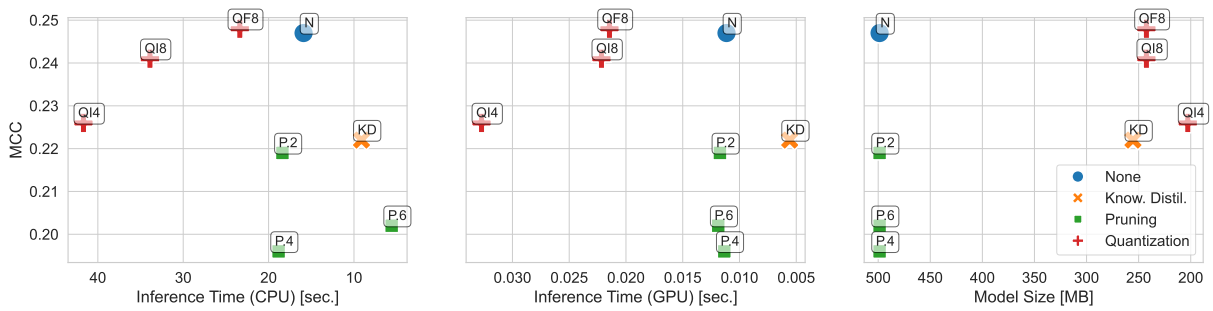
Compression Method	Efficiency			Effectiveness		
	CPU Inf. Time	GPU Inf. Time	Model Size	MRR	MRR@1	MRR@5
None	6.047 (sec)	0.010 (sec)	499 (MB)	0.329	0.242	0.310
Know. Distil.	**-84.7% ± 0.7%**	**-29.2% ± 0.5%**	-48.7%	-52.4%	-57.7%	-54.3%
Pruning (0.2)	+5.5% ± 1.0%	+11.1% ± 0.7%	0.0%	-3.2%	-3.4%	-3.4%
Pruning (0.4)	+16.1% ± 2.2%	+6.4% ± 1.4%	0.0%	-52.1%	-57.3%	-54.3%
Pruning (0.6)	+3.1% ± 3.5%*	+19.8% ± 3.3%	0.0%	-99.6%	-99.9%	-99.8%
Quantization (float8)	+34.2% ± 2.9%	+113.2% ± 1.8%	-51.4%	-0.2%	**0.0%**	-0.3%
Quantization (int8)	+42.2% ± 3.8%	+105.9% ± 1.1%	-51.4%	**0.0%**	-0.1%	**+0.1%**
Quantization (int4)	+61.8% ± 4.8%	+209.6% ± 2.0%	**-59.3%**	-6.3%	-7.6%	-6.7%

(c) RQ_3: Code Search

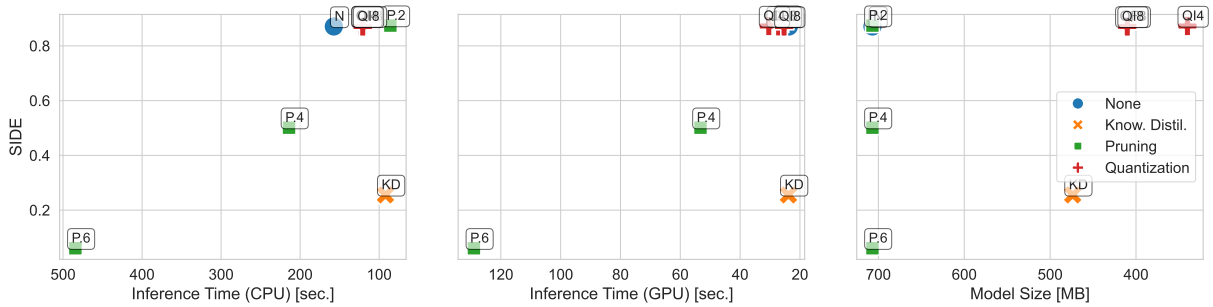
2) Model Size: As can be observed from Table IIIa, all the compression strategies, with the exception of pruning, reduce the size the vulnerability detection model. `int4` quantization provides the highest model's size reduction (-59.3%), followed by the other two quantization configurations (-51.4%). Those results align with the expected quantization behaviour, where a lower bits' precision implies a lower model size. Knowledge Distillation reduces the original model size by almost half (-48.8%). This result aligns with the smaller architecture of the distilled model compared with the original fine-tuned CodeBERT model. In contrast, pruning does not affect the model size. This behavior can be explained by the unstructured nature of the pruning strategy employed, where pruning impacts only the weight values by setting them to zero without modifying the network structure itself [9].

3) Effectiveness: Notably, we find that quantization has very limited impact on the model's effectiveness (see Table IIIa). `float8` quantization marginally changes the effectiveness, in terms of Accuracy, F1-score, and MCC (0.0%, -1.1%, and $+0.4\%$, respectively). Similar results hold for `int8` quantization, while `int4` quantization provides a slightly higher degradation, especially for F1 (-4.4%) and MCC (-8.5%).

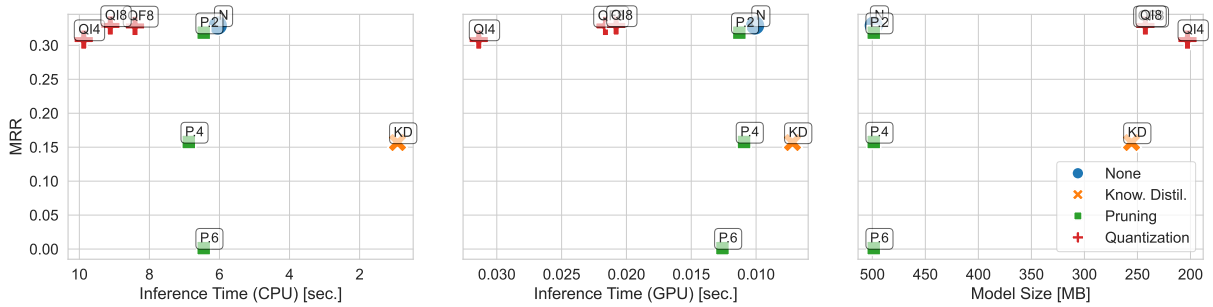
Knowledge distillation performs slightly worse than quantization, particularly in terms of MCC (-10.1%) and Accuracy (-2.2%). However, we observe an improvement in the F1-score ($+3.1\%$). This means that the distilled model has a higher tendency to predict vulnerabilities compared to the original fine-tuned CodeBERT model, which can decrease the number of true negative instances (i.e., increase the number of false positive instances). This leads to an improvement in recall, while only marginally affecting precision, which overall



(a) RQ_1: Vulnerability Detection



(b) RQ_2: Code Summarization



(c) RQ_3: Code Search

Fig. 2. Trade-off between effectiveness (y-axis) and efficiency (x-axis) metrics for each of the SE tasks.

positively impacts the F1-score. Nonetheless, the reduction in MCC suggests a lower correlation between the predicted vulnerabilities and the actual ones.

Pruning has the most significant detrimental impact on vulnerability detection effectiveness. All the pruning configurations considerably reduce model effectiveness, with MCC changes ranging from -11.3% to -20.6%. Among all compression strategies, Pruning 0.4 performs the worst across all three metrics, showing effectiveness losses of -7.3%, -61%, and -20.6% for Accuracy, F1-score, and MCC, respectively. Interestingly, Pruning 0.6 performs slightly better than Pruning 0.4, despite prior research suggesting more severe effectiveness degradation with higher pruning levels [42].

4) Trade-offs: Figure 2a shows the effectiveness-efficiency trade-off provided by each compression strategy. For each subplot, the upper-right solutions are the best. We use MCC as the effectiveness metric since, as explained in Section III, it is the most comprehensive metric for classification [54].

From the far-right subplot, we observe that quantization achieves the best model size reduction while maintaining effectiveness levels comparable to the baseline. However, as shown in the first two subplots, this gain comes at the cost of increased inference time on both CPU and GPU. From Figure 2a, we observe that pruning is consistently outperformed by other strategies across all efficiency and effectiveness metrics. The only exception is Pruning 0.6, which offers the fastest inference time but at the expense of a comparatively low effectiveness. Knowledge distillation is, on the other hand, the only compression strategy that improves efficiency across all dimensions (i.e., CPU and GPU inference time, and model size). At the same time, this strategy results in a comparatively moderate effectiveness degradation–greater than quantization but less than pruning. Overall, this makes knowledge distillation the strategy that offers the most balanced trade-off between efficiency gains and effectiveness degradation.

Answer to RQ$_1$: Quantization is the most effective strategy for reducing memory size in vulnerability detection models (up to -59.3%), with minimal impact on model effectiveness (-8.5% MCC in the worst case). However, it can significantly increase inference time, by as much as $+201.6\%$. Pruning, on the other hand, shows no efficiency gains, with the exception of the 0.6 configuration that results in notable speed-up on CPU (up to -67.9%), but at the cost of a -18.2% effectiveness degradation in MCC. Finally, Knowledge Distillation improves both inference time (up to -47.7%) and model size (up to -48.8%), while moderately impacting vulnerability detection effectiveness, with a reduction of -10.1% in MCC.

B. RQ$_2$ Results - Code Summarization

1) Inference Time: As shown in Table IIIb, Pruning 0.2 is the most effective approach for reducing inference time on the CPU, achieving a -45.3% reduction compared to the plain CodeBERT model. Counter intuitively, we observe a negative correlation between the percentage of weights pruned and the inference time reduction. For example, Pruning 0.4 performs worse than Pruning 0.2, increasing inference time by $+24.7\%$, while Pruning 0.6 shows the worst behavior, increasing inference time by $+183.5\%$ on the CPU. On the GPU, we observe an even more pronounced worsening trend, as inference times increase by $+9.8\%$, $+121\%$, and $+419.3\%$ for Pruning 0.2, 0.4, and 0.6, respectively. A possible explanation for this behaviour could be the lower hardware's ability to handle sparse matrix multiplications [60]. Since generation tasks require multiple network forward steps, a higher sparsity of the weights could increase the overall inference time.

Quantization strategies can all reduce the inference time on the CPU, with `int8` quantization being the best configuration (-27.2%). This behaviour differs from what we observed for vulnerability detection and could be explained by the higher complexity of the underlying task, which may benefit more from reduced weight precision [57]. On the other hand, on the GPU, quantized models are overall slower than the plain CodeBERT model, with inference time slow-downs ranging from $+6.2\%$ to $+29.1\%$.

Knowledge Distillation is the only strategy capable of reducing inference time of code summarization models across both CPU and GPU environments. On the CPU, it is the second-best compression strategy for inference time reduction (-39.8%). On the GPU, it is the only strategy to achieve an inference time reduction, with a decrease of (-2.2%). However, this reduction is not statistically significant, as the confidence interval for the relative change includes zero (see Section III-C for details).

2) Model Size: From Table IIIb, we observe how, like in vulnerability detection, `int4` quantization is the best strategy for reducing the model's size (-51.9%), followed by the other two quantization configurations. Again, these results align with the expected quantization behaviour, where the lower the precision, the lower the size. Knowledge Distillation can also reduce the model size (by -33%), while the adopted pruning strategies confirms that they do not lead any change.

3) Effectiveness: From Table IIIc we observe that all quantization strategies provide only marginal change in the model's effectiveness, with `float8` and `int8` quantization having almost comparable metrics to the baseline, and `int4` quantization showing limited degradations of -2%, -0.1%, -0.2% for Bleu, BERTScore and SIDE, respectively.

For pruning, we observe that the impact grows as pruning becomes more aggressive. Specifically, Pruning 0.2 shows marginal impact, with a slight degradation of -4.3% in summary-summary text similarity (BLEU). On the other hand, Pruning 0.4 and 0.6 substantially reduce the model's effectiveness across all dimensions. For instance, in terms of summary-code semantic similarity (SIDE), Pruning 0.4 and 0.6 lead to effectiveness decreases of -42.4% and -93%, respectively. This behaviour is in line with previous research showing how the effectiveness of a BERT model starts to decrease if the amount of pruned weights is $\geq 40\%$ [42].

Knowledge Distillation also shows significant impact, particularly in terms of summary-summary text similarity (-42.3% in Bleu) and summary-code semantic similarity (-70.6% in SIDE). This behavior is consistent with previous studies, which highlight that distilled models are less effective for tasks beyond classification [22].

4) Trade-offs: Figure 2b illustrates the trade-off between effectiveness and efficiency. We use SIDE as the effectiveness metric, as it is specifically designed for code summarization tasks [36]. We observe that Knowledge Distillation overall improves the inference time and size of the model, but it significantly degrades effectiveness. In contrast, quantization generally performs well across all efficiency metrics, with only a slight slow-down in inference time on the GPU. Moreover, this strategy has only a marginal impact on the model's effectiveness, with values comparable with the baseline. As such, quantization appears to offer the best balance between efficiency and effectiveness as a compression strategy. Pruned models are generally outperformed by other compressed models in terms of both efficiency and effectiveness. The only exception is Pruning 0.2, which provides the best trade-off between inference time and effectiveness on the CPU. However, using pruning does not improve model size.

Answer to RQ$_2$: Quantization strategies offer the best efficiency-effectiveness trade-off among compression techniques, with inference time speed-up of up to -27.2%, model size reduction of up to -51.9%, and minimal effectiveness degradation of up to -0.2% in SIDE. Pruning generally underperforms compared to other strategies, but under specific configurations, such as Pruning 0.2, it achieves the best inference time improvements on the CPU (-45.3%). While Knowledge Distillation improves all efficiency metrics (with up to -39.8% in inference time and -33% in model size), it causes significant losses in effectiveness, with reductions of -70.6% in SIDE.

C. RQ$_3$ Results - Code Search

1) Inference Time: Table IIIc reports how Knowledge Distillation is the strategy that better reduces the inference time on both CPU and GPU, with improvements of -84.7% and -29.2%, respectively. We observe that all other compression strategies have a negative impact on inference time. Pruning slows down inference on both CPU and GPU, with an increase

in inference time ranging from $+3.1\%$ to $+19.8\%$. A possible explanation for this behavior with pruning could be the complexity of the code search task, which typically requires multiple comparisons between a code comment and the code snippets. In that, the sparsity of matrices could increase inference time, especially if the hardware is not well-optimized for handling sparse matrices [26]. Similarly to vulnerability detection, we observe that quantization consistently increases the inference time of code search on both CPU and GPU. The magnitude of the increase is lower on CPU, ranging from $+34.2\%$ to $+61.8\%$, and higher on GPU, with variations from $+113.2\%$ to $+209.6\%$. The negative behavior of quantization strategies could be (once again) explained by the lack of optimization in CPU and GPU kernels for low-precision bit operations [9], [57]–[59].

2) Model Size: Since the model used for this task is the same as the one used for vulnerability detection (i.e., CodeBERT), the results regarding the model size are identical. Hence, `int4` quantization emerges as the best strategy, while the unstructured nature of pruning does not imply any change.

3) Effectiveness: Table IIIc shows how `int8` and `float8` quantization provide no significant change compared with the baseline. A slightly higher degradation is instead observed with `int4` quantization (-6.3% in MRR). All pruning strategies provide a degradation in the model's effectiveness, with reductions in MRR ranging from -3.2% to -99.6%. We observe a correlation between the increase in effectiveness loss and the percentage of pruned weights. Pruning 0.6 strategy proves to be the worst compression strategy, with an MRR loss of -99.6%. Finally, we also observe a significant degradation in effectiveness for Knowledge Distillation, with a -52.1% loss in MRR. This result is consistent with previous research, which has shown that distilled models often exhibit lower effectiveness on tasks different from classification [41].

4) Trade-offs: Figure 2c shows the effectiveness-efficiency trade-off. We adopt the general MRR score as the effectiveness metric. We observe that quantization strategies are preferred to reduce the model size while not impacting its effectiveness; however, they negatively influence the inference time. Knowledge Distillation achieves significant reductions in both inference time and model size, but it also drastically reduces the model's effectiveness, with a loss of -52.1% in MRR. Finally, pruning strategies do not achieve positive results in any of the efficiency and effectiveness metrics analysed.

Answer to RQ$_3$: Quantization strategies drastically reduce the size of code search models (up to -59.3%) with only a marginal impact on effectiveness (up to -6.3% in MRR). However, they can significantly slow down inference time, with an increase of up to $+61.8\%$ on CPU and $+209.6\%$ on GPU. Knowledge Distillation reduces both model size (-48.7%) and inference time (-84.7% on CPU and -29.2% on GPU), but it comes at the cost of a considerable loss in effectiveness (-52.4% in MRR). Pruning, however, does not show improvements in any of the efficiency metrics analysed.

V. Discussion

In the following, we discuss the overall behaviour of the analysed compression strategies and report insights for practitioners and researchers derived from our empirical evaluation.

A. Performance of LM Compression Strategies

1) Knowledge Distillation: We found that Knowledge Distillation is the only compression strategy capable of improving both inference time and model size across all the software engineering tasks we considered. However, this strategy can lead to a significant loss in effectiveness. This loss in effectiveness is particularly pronounced in tasks like code search and summarization, while it is milder in vulnerability detection. These findings align with previous research, which suggests that distilled models tend to be less effective for tasks other than classification [41]. Indeed, to date, knowledge distillation has predominantly been applied to code classification tasks, such as vulnerability detection and clone detection [9], [10].

2) Model Quantization: We found that quantization drastically reduces the size of models across all tasks while maintaining relatively high effectiveness. However, this improvement often comes at the cost of increased inference time. On GPU environments, quantization can double or even triple the inference time for tasks like vulnerability detection and code search. In contrast, the slowdown in inference time is much less pronounced for the code summarization task. Interestingly, in CPU environments, quantization can even improve the inference time for code summarization. However, for vulnerability detection and code search, it still introduces notable slowdowns. Based on these findings, we hypothesize that quantization performs better in terms of efficiency when the task is complex—requiring multiple forward passes through the model—such as in code generation tasks. These results align with previous studies on the application of quantization for code generation tasks [11].

3) Model Pruning: We found that while pruning offers limited overall benefits in terms of efficiency, it can lead to significant speed-ups in inference time under specific combinations of configurations and environments. For instance, Pruning 0.2 provides the highest inference speed-up for code summarization on CPU among all compression strategies, with only a marginal reduction in effectiveness. Similarly, in the vulnerability detection task, Pruning 0.6 proves to be the most effective strategy for improving inference time on CPU, though it comes with a moderate loss in effectiveness. These findings suggest that, when properly configured, pruning can deliver substantial inference speed-ups, particularly in CPU environments.

B. Insights

1) Insights for Practitioners: Our results indicate that the impact of different compression strategies can vary significantly depending on the SE task and the underlying execution environment, often involving important efficiency-effectiveness trade-offs. Hence, practitioners should carefully

select the compression strategy based on their requirements and the underlying task as follows:

- If the practitioner's priority is to improve both inference time and model size, regardless of the task or underlying execution environment, Knowledge Distillation is the preferred choice. Indeed, it is the only compression strategy that delivers improvements across all efficiency aspects in both CPU and GPU environments. However, practitioners should be aware of potential negative impacts on effectiveness, particularly in tasks other than code classification (e.g., code summarization and code search). Additionally, it is important to note that Knowledge Distillation is the only compression strategy that requires re-training a model from scratch, which may be undesirable if practitioners lack sufficient computational resources.
- If the priority is to reduce model size without significantly degrading effectiveness, quantization is the natural choice, regardless of the SE task. Nevertheless, practitioners should be mindful of the potential side effects on inference time, which can vary greatly depending on the task and environment, and often result in significant slowdowns. In some specific cases, however, quantization can even improve inference time, such as code summarization on CPU.
- If the practitioner's goal is to reduce inference time in GPU environments, the only viable choice is Knowledge Distillation. However, as previously mentioned, this approach can drastically affect model effectiveness, particularly for tasks like code summarization and code search.
- If the priority is to improve inference time on CPU, the practitioner could once again consider Knowledge Distillation. They may also consider pruning, especially for tasks such as vulnerability detection and code summarization. However, pruning must be carefully configured to provide benefits. For instance, a less aggressive form of pruning (0.2) proved beneficial in reducing inference time for vulnerability detection, while a more aggressive form (0.6) resulted in the highest inference speed-up for code summarization. For code search, Knowledge Distillation remains the only viable option for reducing inference time on CPU, though it drastically reduces the model's effectiveness. In fact, we did not find any compression strategy that improves inference time for code search without significant losses in effectiveness.

2) Insights for Researchers: Our results show that the behaviour of compression strategies greatly varies depending on the context. However, when these strategies are carefully selected for the underlying task and environment, they can significantly enhance efficiency aspects with a marginal impact on effectiveness. For instance, we found that specific quantization configurations can enhance both CPU inference time and model size without influencing the effectiveness of code summarization. These findings highlight the potential for developing approaches that automatically select the optimal compression strategy based on the underlying task, execution environment, and efficiency/effectiveness requirements. Moreover, we observed that the efficacy of compression strategies can highly depend on their specific configuration. This behavior is particularly evident in the pruning strategy, where the amount of pruned weights significantly influence its impact on inference time. We encourage future research aimed at developing approaches to automatically identify the optimal compression configuration (e.g., amount of weights to prune), based on the SE task and execution environment.

VI. Threats to Validity

Internal Validity: Execution time measurements are typically subject to variability, which can hinder rigorous evaluation [61]. To address this issue, we followed best practices from performance engineering to increase the reliability of our evaluation [44]–[49] (see Section III-C). Furthermore, the outcomes may be influenced by potential errors in the implementation of the baseline models and compression strategies. To address this concern, we utilized a widely adopted benchmark (CodeXGLUE) for training and testing the baseline models and employed compression strategies implementations from widely used and maintained libraries.

Construct Validity: As explained in Section III-D, we employed multiple metrics for each task to assess a model's effectiveness. This has been done to account for possible threats concerning adopting specific metrics (like Accuracy [54] or Bleu [55]). Concerning efficiency metrics, we relied on standard approaches and statistics to measure inference time and model size.

External Validity: The major threats of our work concern its generalizability. The results obtained are limited to the models and tasks analysed and the environment in which the experiments were run. In addition, the results concerning Knowledge Distillation are specific to the DistilBERT LM and may not hold for other distilled models. Future work can extend our analysis with other LMs and code-related tasks and by analysing other Knowledge Distillation techniques. To this end we strove to describe the methodology we followed as clearly as possible and have made our code and scripts publicly available [13].

VII. Conclusion and Future Work

In this paper, we performed an empirical evaluation of the impact that using three LM compression strategies (i.e., Knowledge Distillation, Quantization, and Pruning) could have on the inference time, memory size, and effectiveness of models fine-tuned for three widely adopted SE tasks – vulnerability prediction, code summarization, and code search. Results show how each strategy provides some trade-offs among the three analysed dimensions and how the proper compression method should be chosen based on the underlying task and practitioner's priorities. Future work includes extending our analysis to other SE tasks, language models (like CodeT5 or GraphCodeBERT), compression strategies and energy metrics. Another interesting avenue would be the analysis of how compression strategies perform in devices with very limited computational resources.

ACKNOWLEDGMENT

This work is supported by European Union - NextGenerationEU - National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza, PNRR) - National Centre for HPC, Big Data and Quantum Computing (CN_00000013 – CUP: E13C22001000006) and by EMELIOT national research project, which has been funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY) and by Territori Aperti, a project funded by Fondo Territori Lavoro e Conoscenza CGIL CISL UIL. Part of the numerical simulations have been realized on the Linux HPC cluster Caliban of the High-Performance Computing Laboratory of the Department of Information Engineering, Computer Science and Mathematics (DISIM) at the University of L'Aquila.

REFERENCES

[1] Y. Ding, Y. Fu, O. Ibrahim *et al.*, "Vulnerability detection with code language models: How far are we?" 2024. [Online]. Available: https://arxiv.org/abs/2403.18624

[2] Z. Sun, X. Du, F. Song *et al.*, "When neural code completion models size up the situation: Attaining cheaper and faster completion through dynamic model inference," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639120

[3] J. Chen, X. Hu, Z. Li *et al.*, "Code search is all you need? improving code suggestions with code search," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639085

[4] X. Hou, Y. Zhao, Y. Liu *et al.*, "Large language models for software engineering: A systematic literature review," 2024. [Online]. Available: https://arxiv.org/abs/2308.10620

[5] R. Schwartz, J. Dodge, N. A. Smith *et al.*, "Green ai," 2019. [Online]. Available: https://arxiv.org/abs/1907.10597

[6] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," in *NIPS Deep Learning and Representation Learning Workshop*, 2015. [Online]. Available: http://arxiv.org/abs/1503.02531

[7] O. Zafrir, G. Boudoukh, P. Izsak *et al.*, "Q8bert: Quantized 8bit bert," in *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*. Los Alamitos, CA, USA: IEEE Computer Society, dec 2019, pp. 36–39. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/EMC2-NIPS53020.2019.00016

[8] V. Sanh, T. Wolf, and A. M. Rush, "Movement pruning: adaptive sparsity by fine-tuning," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[9] J. Shi, Z. Yang, B. Xu *et al.*, "Compressing pre-trained models of code into 3 mb," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556964

[10] J. Shi, Z. Yang, H. J. Kang *et al.*, "Greening large language models of code," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society*, ser. ICSE-SEIS'24. New York, NY, USA: Association for Computing Machinery, 2024, p. 142–153. [Online]. Available: https://doi.org/10.1145/3639475.3640097

[11] X. Wei, S. K. Gonugondla, S. Wang *et al.*, "Towards greener yet powerful code generation via quantization: An empirical study," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 224–236. [Online]. Available: https://doi.org/10.1145/3611643.3616302

[12] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[13] G. d'Aloisio, L. Traini, F. Sarro, and A. Di Marco, "On the compression of language models for code: An empirical study on codebert," Dec. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.14357478

[14] A. Vaswani, N. Shazeer, N. Parmar *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[15] A. Radford, J. Wu, R. Child *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[16] C. Raffel, N. Shazeer, A. Roberts *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer," 2023. [Online]. Available: https://arxiv.org/abs/1910.10683

[17] OpenAI, "Openai codex," 2019. [Online]. Available: https://arxiv.org/abs/1810.04805

[18] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, "CodeT5+: Open code large language models for code understanding and generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 1069–1088. [Online]. Available: https://aclanthology.org/2023.emnlp-main.68

[19] M. Chen, J. Tworek, H. Jun *et al.*, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[20] J. Gu, P. Salza, and H. C. Gall, "Assemble foundation models for automatic code summarization," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2022, pp. 935–946. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER53432.2022.00112

[21] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codebert," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 425–436.

[22] V. Sanh, L. Debut, J. Chaumond *et al.*, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," 2020. [Online]. Available: https://arxiv.org/abs/1910.01108

[23] Y. LeCun, J. Denker, and S. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems*, D. Touretzky, Ed., vol. 2. Morgan-Kaufmann, 1989. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1989/file/6c9882bbac1c7093bd25041881277658-Paper.pdf

[24] H. Li, A. Kadav, I. Durdanovic *et al.*, "Pruning filters for efficient convnets," in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=rJqFGTslg

[25] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," 2016. [Online]. Available: https://arxiv.org/abs/1608.04493

[26] P. Molchanov, S. Tyree, T. Karras *et al.*, "Pruning convolutional neural networks for resource efficient inference," 2017. [Online]. Available: https://arxiv.org/abs/1611.06440

[27] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," 2019. [Online]. Available: https://arxiv.org/abs/1902.09574

[28] D. Guo, S. Ren, S. Lu *et al.*, "Graphcodebert: Pre-training code representations with data flow," 2021. [Online]. Available: https://arxiv.org/abs/2009.08366

[29] Q. Zhang and B. Wu, "Software defect prediction via transformer," in *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, vol. 1. IEEE, 2020, pp. 874–879.

[30] W. Wang, Y. Zhang, Z. Zeng *et al.*, "Trans^ 3: A transformer-based framework for unifying code summarization and code search," *arXiv preprint arXiv:2003.03238*, 2020.

[31] G. Rosenfield and K. Fitzpatrick-Lins, "A coefficient of agreement as a measure of thematic classification accuracy." *Photogrammetric Engineering and Remote Sensing*, vol. 52, no. 2, pp. 223–227, 1986. [Online]. Available: http://pubs.er.usgs.gov/publication/70014667

[32] A. A. Taha and A. Hanbury, "Metrics for evaluating 3D medical image segmentation: analysis, selection, and tool," *BMC Medical Imaging*, vol. 15, no. 1, p. 29, Aug. 2015. [Online]. Available: https://doi.org/10.1186/s12880-015-0068-x

[33] D. Chicco and G. Jurman, "The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation," *BMC genomics*, vol. 21, pp. 1–13, 2020.

[34] K. Papineni, S. Roukos, T. Ward *et al.*, "Bleu: a Method for Automatic Evaluation of Machine Translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, P. Isabelle,

E. Charniak, and D. Lin, Eds. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040

[35] T. Zhang, V. Kishore, F. Wu *et al.*, "Bertscore: Evaluating text generation with bert," *arXiv preprint arXiv:1904.09675*, 2019.

[36] A. Mastropaolo, M. Ciniselli, M. Di Penta *et al.*, "Evaluating Code Summarization Techniques: A New Metric and an Empirical Characterization," Dec. 2023. [Online]. Available: https://arxiv.org/abs/2312.15475v1

[37] O. Chapelle, D. Metlzer, Y. Zhang *et al.*, "Expected reciprocal rank for graded relevance," in *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, ser. CIKM '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 621–630. [Online]. Available: https://doi.org/10.1145/1645953.1646033

[38] Y. Zhou, S. Liu, J. Siow *et al.*, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 10 197–10 207.

[39] H. Husain, H.-H. Wu, T. Gazit *et al.*, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[40] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[41] V. Sanh, L. Debut, J. Chaumond *et al.*, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," 2020. [Online]. Available: https://arxiv.org/abs/1910.01108

[42] M. A. Gordon, K. Duh, and N. Andrews, "Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning," May 2020, arXiv:2002.08307 [cs]. [Online]. Available: http://arxiv.org/abs/2002.08307

[43] A. Kumar, A. M. Shaikh, Y. Li *et al.*, "Pruning filters with l1-norm and capped l1-norm for cnn compression," *Applied Intelligence*, vol. 51, pp. 1152–1160, 2021.

[44] L. Traini, F. Di Menna, and V. Cortellessa, "Ai-driven java performance testing: Balancing result quality with testing time," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 443–454. [Online]. Available: https://doi.org/10.1145/3691620.3695017

[45] M. Jangali, Y. Tang, N. Alexandersson *et al.*, "Automated generation and evaluation of jmh microbenchmark suites from unit tests," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1704–1725, 2023.

[46] Z. Zhang, Z. Xing, X. Xia *et al.*, "Faster or slower? performance mystery of python idioms unveiled with empirical evidence," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 1495–1507. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00130

[47] C. Laaber, S. Würsten, H. C. Gall *et al.*, "Dynamically reconfiguring software microbenchmarks: Reducing execution time without sacrificing result quality," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 989–1001. [Online]. Available: https://doi.org/10.1145/3368089.3409683

[48] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proceedings of the 2013 International Symposium on Memory Management*, ser. ISMM '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 63–74. [Online]. Available: https://doi.org/10.1145/2464157.2464160

[49] ——, "Quantifying performance changes with effect size confidence intervals," University of Kent, Technical Report 4–12, June 2012. [Online]. Available: http://www.cs.kent.ac.uk/pubs/2012/3233

[50] R. F. Woolson, "Wilcoxon signed-rank test," *Encyclopedia of Biostatistics*, vol. 8, 2005.

[51] L. Traini, D. Di Pompeo, M. Tucci *et al.*, "How software refactoring impacts execution time," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, dec 2021. [Online]. Available: https://doi.org/10.1145/3485136

[52] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, April 2010, most Influential Paper Award at ICST 2020 MIP Practical.

[53] M. Jimenez, R. Rwemalika, M. Papadakis *et al.*, "The importance of accounting for real-world labelling when predicting software vulnerabilities," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 695–705. [Online]. Available: https://doi.org/10.1145/3338906.3338941

[54] R. Moussa and F. Sarro, "On the use of evaluation measures for defect prediction studies," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3533767.3534405

[55] W. Sun, Y. Miao, Y. Li *et al.*, "Source Code Summarization in the Era of Large Language Models," Jul. 2024, arXiv:2407.07959 [cs]. [Online]. Available: http://arxiv.org/abs/2407.07959

[56] Y. Xie, J. Lin, H. Dong *et al.*, "Survey of code search based on deep learning," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, Dec. 2023. [Online]. Available: https://doi.org/10.1145/3628161

[57] A. Gholami, S. Kim, Z. Dong *et al.*, "A survey of quantization methods for efficient neural network inference," in *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326.

[58] M. Nagel, R. A. Amjad, M. van Baalen *et al.*, "Up or down? adaptive rounding for post-training quantization," 2020. [Online]. Available: https://arxiv.org/abs/2004.10568

[59] P. Ganesh, Y. Chen, X. Lou *et al.*, "Compressing large-scale transformer-based models: A case study on bert," *Transactions of the Association for Computational Linguistics*, vol. 9, pp. 1061–1080, 2021.

[60] Z. Liu, M. Sun, T. Zhou *et al.*, "Rethinking the value of network pruning," 2019. [Online]. Available: https://arxiv.org/abs/1810.05270

[61] A. Maricq, D. Duplyakin, I. Jimenez *et al.*, "Taming performance variability," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 409–425. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/maricq