Small Language Models for specific objectives on resource constrained devices

Rajat Kumar Thakur
202211070@diu.iiitvadodara.ac.in
Indian Institute of Information Technology Vadodara
Gujarat, India

Anirban Dasgupta
anirbandg@iitgn.ac.in
Indian Institute of Technology Gandhinagar
Gujarat, India

Abstract

The exponential growth of IoT devices and edge computing has made efficient AI models that can be run on low-resource hardware more sought after. This research delves into the design of Small Language Models for the generation of Python code, with real-time use cases like running and evaluating machine logs on edge devices. Two lightweight models, TinyLlama and StarCoder, were trained on a combination of varied datasets like SciDocs, Astronomy Stack Text, Wikipedia Field of Science, and The Stack for code generation for user's natural language queries. Pruning, weight sharing, and early exit strategies were employed to achieve maximum performance. Results indicate that pruning had a drastic impact on latency for TinyLlama, making it more viable for real-time applications on the cost of code quality. StarCoder was less accurate in producing correct executable code for scientific data even after optimization, indicating potential avenues for improvement. This research demonstrates the viability of small language models for domain specific use cases on low-resource hardware while highlighting difficulties in achieving the equilibrium between efficiency and functionality, paving the way for future research in lightweight deployment of AI.

Keywords: Small Language Models, Code Generation, Model Optimization, Resource Constrained Devices, Edge Computing

1 Introduction

The fast pace of edge computing combined with the proliferation of Internet of Things (IoT) devices has brought about an immediate need for artificial intelligence (AI) models that are capable of working optimally under the constraints of limited hardware resources. Platforms like IoT sensors, smartphones, and embedded systems tend to be constrained by memory space, computational power, and power supply, making it difficult to deploy large-scale, computationally expensive models. While Large Language Models (LLMs) are extremely capable, they tend to be too resource-intensive for such environments, causing increased latency, excessive energy usage, and unrealistic memory consumption. This has prompted the creation of Small Language Models, which aim to strike a balance between performance and efficiency, thereby facilitating real-time applications on the edge device.

One of the main uses in this framework is machine logs and datasets analysis and inspection. Machine logs play a crucial part in system debugging, monitoring, and preventative maintenance, yet real-time processing in edge devices is not always possible due to limited resources. On the other hand, dataset parsing and interpretation on-site can be critical to operations such as data preprocessing, feature extraction, and report generation. Cloud computing has traditionally been the method for these operations, which causes latency, bandwidth limitations, and concerns over privacy, particularly in environments with intermittent connectivity or where data security levels are high.

To overcome these limitations, this paper targets the creation of SLMs that are capable of data set analysis, writing and running Python code, and presenting results to users directly on resource-limited devices. By activating these features on-device, we seek to provide real-time, offline data processing and code creation independent of cloud resources. In addition to removing latency and consumption of bandwidth, this also provides greater privacy and security by keeping sensitive data in local possession. For this study, we chose two advanced state-of-the-art SLMs: TinyLlama [11], with 1.1 billion parameters, is optimized for efficiency and flexibility and is best suited for a broad variety of tasks on the edge device and, StarCoder [6], which is fine-tuned specifically for code generation, is particularly strong at generating Python code for the processing of dataset and log files. To improve their edge deployability, we used optimization methods such as pruning, weight sharing, and early exit [2]. These methods are intended to minimize model size, latency, and memory usage but preserve the capability to generate functional and executable code.

The organization of this report is as follows: Section 2 provides information on related work in SLMs and optimization methods of edge computing. Section 3 describes our methodology, from dataset choice to model training and optimization methods. Section 4 contains experimental results, comparing the code generation, execution, and resource usage of the baseline and optimized models. Section 5 provides the implications of our results, the trade-offs we faced, and possible future directions. Lastly, Section 6 concludes the report with an overview of key insights and contributions.

SRIP, IIT Gandhinagar
2025.

2 Related Work

The rise of edge computing and IoT devices has increased the demand for AI models that operate efficiently on resource constrained hardware. While large language models (LLMs) excel in tasks like log analysis and code generation, their computational demands often render them impractical for such environments. Small Language Models are emerging as a promising alternative for enabling real-time applications on edge devices. This section reviews key works LogLLM, data efficient LLM fine-tuning, and LLaVaOLMoBitnet1B highlighting their relevance and limitations in the context of developing SLMs for code generation and execution.

2.1 LogLLM: Log-based Anomaly Detection Using Large Language Models

Guan et al. [4] proposed LogLLM, a framework that uses BERT for semantic vector extraction and Llama for classifying log sequences. A projector aligns the vector spaces of both models to ensure semantic consistency. Unlike traditional approaches requiring complex template extraction, LogLLM simplifies preprocessing with regular expressions and employs a three-stage training procedure. It achieves state-of-the-art results across diverse datasets including HDFS, BGL, Liberty, and Thunderbird.

However, LogLLM's reliance on large models like BERT and Llama makes it unsuitable for resource-constrained devices. Moreover, it targets anomaly detection rather than executable code generation, limiting its applicability to our focus on lightweight code-oriented SLMs.

2.2 Data-efficient LLM Fine-tuning for Code Generation

Lv et al. [7] tackled efficient fine-tuning for code generation using a data selection strategy that emphasizes quality and distribution alignment. Their "dynamic pack" technique also minimizes padding during tokenization. Using only 40% of the OSS-Instruct dataset, their fine-tuned DeepSeek-Coder-Base-6.7B outperforms the full-data model, achieving 66.9% average performance [7].

Despite these gains, the use of large models like the 6.7B-parameter DeepSeek-Coder remains a barrier for deployment on low-resource devices. The method improves training efficiency but doesn't address inference-time limitations or the need for compact, edge-friendly SLMs.

2.3 LLaVaOLMoBitnet1B: Ternary LLM goes Multimodal!

Sundaram and Iyer [10] introduced LLaVaOLMoBitnet1B, a ternary-weight (3-bit) multimodal LLM capable of processing both text and image inputs. This weight reduction greatly lowers resource consumption. The open-sourced model uses a two-step training process: feature alignment followed by instruction fine-tuning.

While efficient, its multimodal focus and potential accuracy trade-offs make it less suitable for our single-modality use case of code generation and log processing. Ternary quantization offers promise, but its relevance is limited in text-only applications requiring high precision, such as code execution.

2.4 Additional Context: Log Analysis and Code Generation

Earlier deep learning methods like DeepLog and LogAnomaly used LSTMs for log analysis but lacked semantic depth and required extensive preprocessing [3, 8]. Similarly, models like Codex and Code Llama show high performance in code generation but are too large for edge deployment [1, 9]. Efforts like quantization and distillation offer efficiency gains, though often targeting general NLP rather than log/code tasks specifically.

2.5 Summary and Research Gap

These works demonstrate progress in LLMs for log analysis and code generation. LogLLM excels in anomaly detection but lacks efficiency; Lv et al.'s fine-tuning approach still depends on large models; and LLaVaOLMoBitnet1B targets multimodal tasks with possible precision limitations.

Our research fills these gaps by developing lightweight SLMs tailored for generating and executing Python code on constrained devices. Using models like TinyLlama [11] and StarCoder [6], and applying optimization techniques such as pruning, weight sharing, and early exit, we aim to deliver real-time performance with minimal overhead—specifically for edge deployments in code and log-based applications.

3 Methodology

In this work, we focus on developing Small Language Models (SLMs) for specific objectives, such as reading machine logs, that can be trained and perform inference on resource-constrained devices. Our approach combines dataset curation, parameter-efficient fine-tuning (PEFT), and model optimization techniques to enable scientific code generation on such devices. The methodology consists of three core phases: dataset curation and preparation, parameter-efficient fine-tuning, and model optimization techniques, followed by an evaluation framework.

3.1 Dataset Curation and Preparation

We integrate multi-domain scientific sources into a unified training corpus to ensure a diverse and comprehensive dataset for training our SLMs. The dataset is defined as:

$$\mathcal{D} = \mathcal{D}_{\text{scidocs}} \cup \mathcal{D}_{\text{astronomy}} \cup \mathcal{D}_{\text{science}} \cup \mathcal{D}_{\text{code}} \tag{1}$$

Where:

- $\mathcal{D}_{\text{scidocs}}$: 25k paper abstracts from AllenAI's SciDocs
- $\mathcal{D}_{\text{astronomy}}$: 15k astronomy problems (DPO-text)
- $\mathcal{D}_{\text{science}}$: 15k Wikipedia science entries
- $\mathcal{D}_{\text{code}}$: 20k Python files from The Stack (filtered for scientific libraries)

To prepare the data for training, we transform non-code entries using prompt templates to convert them into a format suitable for code generation. Specifically:

$$\text{prompt} = \begin{cases} f_{\text{title}}(\text{title}) + f_{\text{abstract}}(\text{abstract}) & \text{for papers} \\ f_{\text{problem}}(\text{problem\_text}) & \text{for astronomy} \\ f_{\text{task}}(\text{text}) & \text{for science topics} \end{cases} \tag{2}$$

Here, f_{title}, f_{abstract}, f_{problem}, and f_{task} are functions that extract and format the relevant parts of the input data into a standardized prompt. This ensures consistency across different data types. The overall dataset preparation process is illustrated in Figure 1, which shows the integration of multi-source data and the application of prompt engineering.

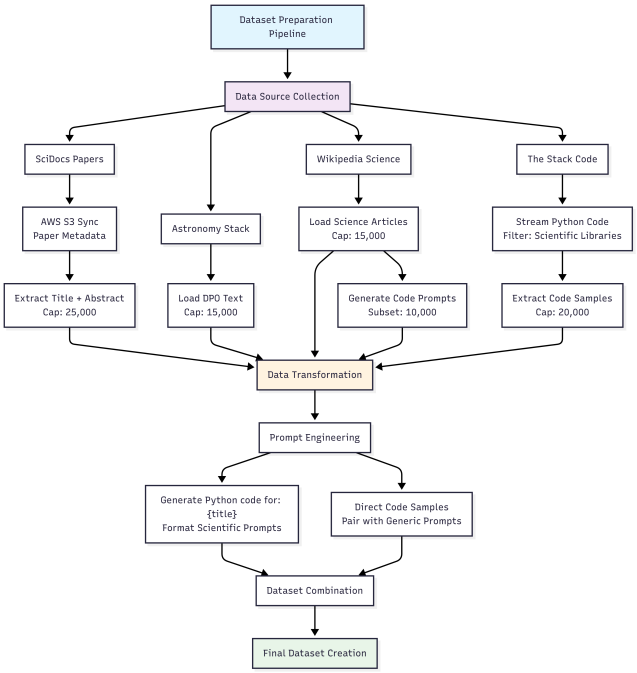


Figure 1. Dataset preparation workflow showing multi-source integration and prompt engineering

3.2 Parameter-Efficient Fine-Tuning

We employ the QLoRA fine-tuning method, which integrates 4-bit quantization with Low-Rank Adaptation (LoRA)[5] to efficiently fine-tune pre-trained small language models. The procedure is detailed in Algorithm 1. It begins by initializing the base model parameters θ_{base}, followed by applying 4-bit quantization to produce θ_{4bit}. Subsequently, LoRA adapters, represented as $\Delta W = B \cdot A$, are injected into the model. The adapters are then optimized to minimize the loss function $\mathcal{L}(\theta_{\text{4bit}} + \Delta W; \mathcal{D})$ with respect to a given dataset $\mathcal{D}$.

Algorithm 1 QLoRA Fine-Tuning

1: Initialize θ_{base} (pre-trained model)
2: Apply 4-bit quantization: $\theta_{\text{4bit}} \leftarrow Q_{\text{NF4}}(\theta_{\text{base}})$
3: Inject LoRA adapters: $\Delta W = B \cdot A$, where $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$
4: Optimize: $\min_{\Delta W} \mathcal{L}(\theta_{\text{4bit}} + \Delta W; \mathcal{D})$

3.3 Model Optimization Techniques

To make the models suitable for resource-constrained devices, we apply several optimization techniques that reduce the model size and inference time without significantly degrading performance. Specifically, we implement three compression strategies:

3.3.1 Structured Pruning: We remove the least important weights based on their L1 or L2 norms. The pruning loss is defined as:

$$\mathcal{L}_{\text{prune}} = \sum_{l=1}^{L} \|\mathbf{W}^{(l)} \odot \mathbf{M}^{(l)}\|_F^2 \tag{3}$$

where $\mathbf{M}^{(l)}$ is a binary mask that indicates which weights are kept, determined by thresholds based on weight magnitudes.

3.3.2 Weight Sharing: We share parameters between the embedding layer and the language model head:

$$\mathbf{W}_{\text{embed}} = \mathbf{W}_{\text{lm\_head}}^{\top} \tag{4}$$

This reduces the total number of parameters and can improve generalization.

3.3.3 Early Exit: We implement an early exit mechanism that terminates inference when the model's confidence exceeds a certain threshold:

$$\text{Exit if } \max(\mathbf{p}^{(l)}) > \tau \quad \forall l \in \{l_{\text{exit}_1}, \ldots, l_{\text{exit}_k}\} \tag{5}$$

This allows for faster inference by avoiding unnecessary computations in deeper layers. Figure 2 provides a visual representation of optimization techniques [2].

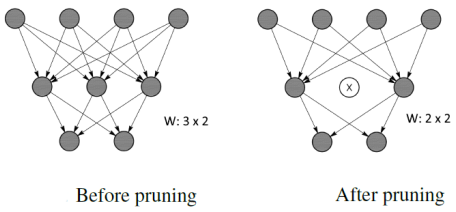


Figure 2. Pruning applied to a neural network

3.4 Evaluation Framework

We assess the performance of our models using a dual set of metrics that capture both inference efficiency and code quality.

3.4.1 Inference Efficiency

We measure:

$$\text{Latency} = \frac{1}{N}\sum_{i=1}^{N} t_{\text{gen}}^{(i)}$$

$$\text{Memory} = \max(\text{GPU\_mem})$$

$$\text{Throughput} = N_{\text{success}}/t_{\text{total}}$$

where $t_{\text{gen}}^{(i)}$ is the generation time for the i-th sample, GPU_mem is the peak GPU memory usage, and N_{success} is the number of successfully generated codes within the total time t_{total}.

3.4.2 Code Quality

For code quality, we define:

$$\text{Validity} = \mathbb{I}[\text{Code compiles}]$$

$$\text{Accuracy} = \mathbb{I}[\text{Execution produces correct output}]$$

$$\text{Q-Score} = \frac{1}{5}\sum_{k=1}^{5} \mathbb{I}[\text{Criterion}_k \text{ satisfied}]$$

The Q-Score is based on five criteria: appropriate library usage, correct result printing, inclusion of visualizations, data generation capabilities, and syntactic correctness. The entire evaluation process, including code generation, repair, and execution validation, is illustrated in Figure 3.

3.5 Datasets

To support our methodology, we utilize the following datasets:

3.5.1 Datasets

- Wikipedia Field of Science: https://huggingface.co/datasets/millawell/wikipedia_field_of_science
- Astronomy Stack DPO Text: https://huggingface.co/datasets/David-Xu/astronomy-stack-dpo-text
- SciDocs: https://github.com/allenai/scidocs
- The Stack: https://huggingface.co/datasets/bigcode/the-stack

These components form the foundation of our approach, ensuring that the SLMs are both effective and efficient for resource-constrained devices.

4 Results

In this section, we present the performance evaluation of various optimization techniques applied to two small language models: StarCoder and TinyLLaMA. The techniques evaluated include the Base Model, Pruning, Weight Sharing, and Early Exit (for StarCoder), and Base Model and Pruning (for TinyLLaMA). We assess the models based on metrics such as inference latency, throughput, memory usage, inference success rate, syntax errors, valid code rate, execution success rate, and code quality score.

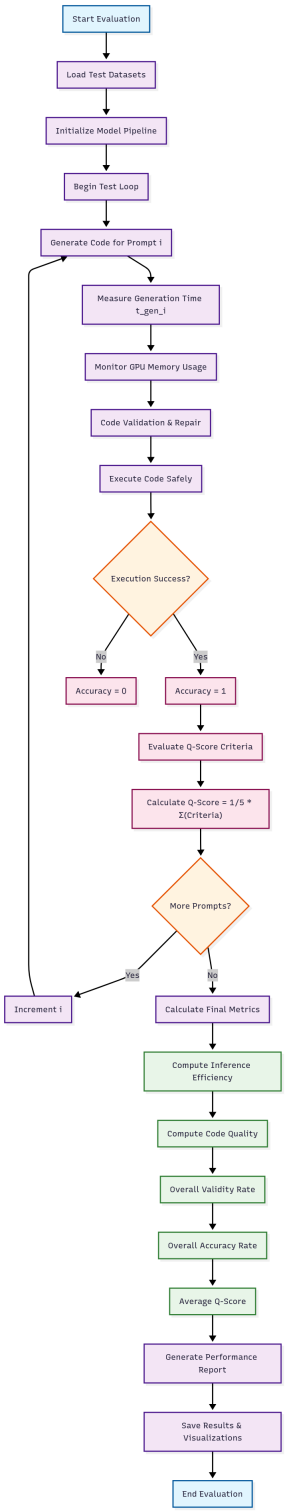


Figure 3. Evaluation pipeline showing code generation, repair, and execution validation

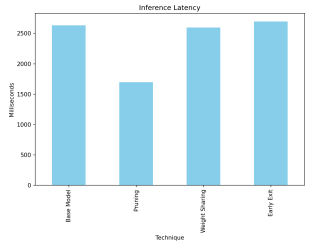
4.1 Results for StarCoder

Table 1 presents the performance metrics for the StarCoder model across different optimization techniques. Figure 4a

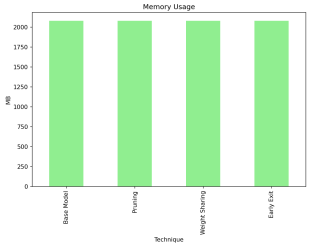
visualizes the inference latency, while Figure 4b shows the memory usage.

Technique	Latency (ms)	Throughput (samples/s)	Memory (MB)	Inference Success (%)	Syntax Errors
Base Model	2664.09	0.342	2080.48	100	5
Pruning	2737.22	0.333	2080.61	100	0
Weight Sharing	2711.61	0.339	2080.61	100	1
Early Exit	2571.31	0.356	2081.48	100	3

Table 1. Performance Metrics for StarCoder Optimization Techniques



(a) Inference Latency for StarCoder Optimization Techniques



(b) Memory Usage for StarCoder Optimization Techniques

Figure 4. Comparison of Optimization Techniques applied on StarCoder

From Table 1, we observe that the Early Exit technique achieves the lowest inference latency (2571.31 ms) and the highest throughput (0.356 samples/s), while maintaining a high inference success rate of 100%. However, it results in 3 syntax errors, which is higher than Pruning (0 errors) and Weight Sharing (1 error). The Base Model has the highest number of syntax errors (5). Memory usage remains relatively consistent across all techniques, with a slight increase for Early Exit (2081.48 MB).

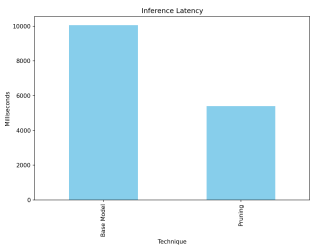
4.2 Results for TinyLLaMA

Table 2 presents the performance metrics for the TinyLLaMA model, comparing the Base Model and Pruning techniques. Figures 5a, 5b, 6a, and 6b visualize the inference latency, valid code rate, execution success rate, and code quality score, respectively.

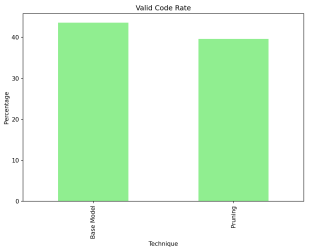
Table 2. Performance Metrics for TinyLLaMA Optimization Techniques

Technique	Latency (ms)	Throughput (samples/s)	Memory (MB)	Inference Success (%)	Valid Code (%)	Execution Success (%)	Quality Score
Base Model	10057.68	0.10	4208.8	100.0	43.6	56.9	65.6
Pruning	5402.01	0.19	4209.3	100.0	39.6	47.5	62.0

For TinyLLaMA, the Pruning technique significantly reduces inference latency from 10057.68 ms to 5402.01 ms, nearly halving the time required for inference. However, this comes at the cost of reduced code quality, with the valid code rate decreasing from 43.6% to 39.6%, execution success rate from 56.9% to 47.5%, and the quality score from 65.6 to 62.0. Memory usage remains almost unchanged, with a negligible increase from 4208.8 MB to 4209.3 MB.

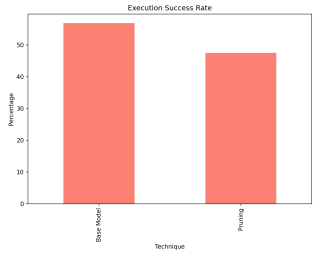


(a) Inference Latency for TinyLLaMA Optimization Techniques

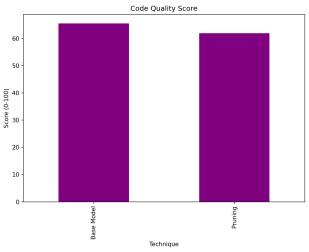


(b) Valid Code Rate for TinyLLaMA Optimization Techniques

Figure 5. Performance Metrics for TinyLLaMA Optimization Techniques



(a) Execution Success Rate for TinyLLaMA Optimization Techniques



(b) Code Quality Score for TinyLLaMA Optimization Techniques

Figure 6. Execution and Code Quality Metrics for TinyLLaMA Optimization Techniques

4.3 Comparative Analysis

Comparing the two models, StarCoder generally exhibits lower inference latency and higher throughput than TinyLLaMA across all techniques. For instance, the Base Model for StarCoder has a latency of 2664.09 ms, compared to 10057.68 ms for TinyLLaMA. However, TinyLLaMA's Pruning technique still results in higher latency (5402.01 ms) than any technique applied to StarCoder. This suggests that StarCoder may be more efficient in terms of speed, while TinyLLaMA might require further optimization to achieve comparable performance.

In terms of code quality, TinyLLaMA's Base Model achieves a higher quality score (65.6) compared to the metrics available for StarCoder, which do not include direct quality scores but rather syntax errors. The Pruning technique for TinyLLaMA reduces the quality score to 62.0, indicating a trade-off between speed and code quality.

Overall, the results demonstrate that optimization techniques such as Pruning and Early Exit can significantly improve inference speed, but may impact other performance metrics such as code quality and syntax errors. The choice of technique should be guided by the specific requirements of the application, balancing speed, efficiency, and output quality.

5 Discussion

The results of this study demonstrate the potential of Small Language Models (SLMs) for generating Python code on resource-constrained devices. For TinyLlama, the pruning optimization technique significantly reduced inference latency from 10057.68 ms to 5402.01 ms, making it more suitable for real-time applications on edge devices. However, this improvement came at the cost of code quality, as evidenced by reductions in valid code rate, execution success rate, and overall quality score. This trade-off underscores the need for careful consideration when applying optimization techniques, as the reduction in computational demands may compromise the model's ability to generate accurate and executable code.

In contrast, StarCoder exhibited lower inference latency (e.g., 2571.31 ms with early exit) and higher throughput (0.356 samples/s) compared to TinyLlama across all techniques, suggesting it may be inherently more efficient for speed-critical tasks. However, even after optimization, StarCoder struggled with generating correct executable code for scientific data, as indicated by the presence of syntax errors (ranging from 1 to 5 across techniques). This limitation points to a potential area for improvement, possibly through further refinement of the model architecture or training data to better handle domain-specific tasks.

The findings also reveal that while optimization techniques such as pruning and early exit can enhance inference speed, they may not uniformly improve all performance metrics. For instance, the early exit technique in StarCoder achieved the lowest latency but resulted in a higher number of syntax errors compared to other methods. This suggests that the choice of optimization technique should be guided by the specific requirements of the application, balancing the need for speed with the necessity for high-quality code generation.

Overall, the study confirms the viability of SLMs for domain-specific applications on low-resource hardware but also emphasizes the importance of ongoing research to address the trade-offs between efficiency and functionality.Additionally, enhancing the models' ability to handle scientific data through targeted fine-tuning or the incorporation of domain-specific knowledge could improve their performance in specialized tasks.

6 Conclusion

This research has demonstrated the potential of Small Language Models for generating Python code on resource-constrained devices, addressing the growing need for efficient AI models in edge computing and IoT applications. By evaluating TinyLlama and StarCoder with various optimization techniques, we have shown that SLMs can be effectively deployed on low-resource hardware, offering a viable alternative to large language models that are computationally intensive.

However, the study also revealed significant challenges in achieving an optimal balance between efficiency and functionality. While optimization techniques like pruning and early exit improved inference speed, they often came at the expense of code quality, particularly in terms of syntax errors and execution success rates. These limitations highlight the need for further research to refine optimization strategies and enhance the models' ability to generate accurate and executable code, especially for domain-specific tasks involving scientific data.

Incorporating domain-specific knowledge or fine-tuning the models on specialized datasets could improve their effectiveness in generating high-quality code for niche applications. By addressing these challenges, SLMs can be further optimized for real-time use cases on edge devices, paving the way for more widespread adoption of AI in resource-constrained environments.

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Oliveira Pinto, Jared Kaplan, ..., and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint* abs/2107.03374 (2021). https://arxiv.org/abs/2107.03374

[2] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A Survey of Model Compression and Acceleration for Deep Neural Networks. *arXiv preprint* abs/1710.09282 (2017). https://arxiv.org/abs/1710.09282

[3] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1285–1298. doi:10.1145/3133956.3134015

[4] Wei Guan, Jian Cao, Shiyou Qian, Jianqi Gao, and Chun Ouyang. 2024. LogLLM: Log-based Anomaly Detection Using Large Language Models. *CoRR* abs/2411.08561 (2024). doi:10.48550/ARXIV.2411.08561

[5] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. https://openreview.net/forum?id=nZeVKeeFYf9 Also available as arXiv:2106.09685.

[6] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Indraneil Paul, and Nicolas Chapados. 2024. StarCoder2 and The Stackv2: The Next Generation. *CoRR* abs/2402.19173 (2024). doi:10.48550/ARXIV.2402.19173

[7] Weijie Lv, Xuan Xia, and Sheng-Jun Huang. 2025. Data-efficient LLM Fine-tuning for Code Generation. *CoRR* abs/2504.12687 (2025). doi:10.48550/ARXIV.2504.12687

[8] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, and Rong Zhou. 2019. LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*. 4739–4745. doi:10.24963/ijcai.2019/658

[9] Benoît Rozière, Jonas Gehring, Florian Gloeckle, ..., and Thomas Scialom. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint* abs/2308.12950 (2023). https://arxiv.org/abs/2308.12950

[10] Jainaveen Sundaram and Ravi Iyer. 2024. LLaVaOLMoBitnet1B: Ternary LLM goes Multimodal! *CoRR* abs/2408.13402 (2024). doi:10.48550/ARXIV.2408.13402

[11] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. TinyLlama: An Open-Source Small Language Model. *CoRR* abs/2401.02385 (2024). doi:10.48550/ARXIV.2401.02385