Journal Publication of International Research for Engineering and Management (JOIREM)
Volume: 03 Issue: 11 | Nov-2025
ISSN (O) 3107-6696

Small Language Model for coding and debugging

Abhigyan Ranjan[1], Ritesh Kumar[2],

[1,2] Scholar, B.Tech. (AI&DS) 4th Year, Department of Artificial Intelligence and Data Science, Dr. Akhilesh Das Gupta Institute of Professional Studies, New Delhi
[1]abhigyanranjanofficial@gmail.com, [2]Assistant Professor (AI & DS)

ABSTRACT: The proliferation of Large Language Models (LLMs) has significantly impacted software development, yet their substantial computational and resource demands create barriers to widespread accessibility. This paper details the development and evaluation of a Small Language Model (SLM) designed as an efficient, practical alternative for coding assistance. The primary goal is to create a lightweight, low-latency model specialized in Python, capable of performing real-time code completion and generating functions from natural language prompts. The methodology employs a transformer-based decoder-only architecture (100-300M parameters) trained on a filtered, high-quality dataset of open-source code. Model performance is assessed using the pass@k metric from the HumanEval benchmark for functional correctness, alongside measurements of inference speed and memory footprint to validate its efficiency. This research will deliver a proof-of-concept prototype, demonstrating that specialized SLMs can offer a sustainable and effective solution that enhances developer productivity while democratizing access to advanced AI-powered coding tools.

Keywords –Small Language Models (SLMs), Code Generation, Large Language Models (LLMs), AI-Assisted Coding, Natural Language Processing (NLP), Transformer Architecture, Program Synthesis, Code Completion, Model Efficiency, Low-Latency AI, HumanEval Benchmark, pass@k, AI Developer Tools, Python.

Abbreviations -

SLM- Small Language Model

LLM- Large Language Model

NLP- Natural Language Processing

GPU- Graphics Processing Unit

API- Application Programming Interface

INTRODUCTION:

Software development has evolved into one of the most critical components of the digital era. From powering large-scale enterprise systems to enabling simple mobile applications, code has become the language that drives modern technology.

Industries such as healthcare, finance, education, and entertainment depend heavily on software solutions to improve efficiency, enhance customer experience, and innovate faster. As a result, developers are constantly challenged to deliver high-quality, scalable, and secure code under tight deadlines.

However, traditional coding practices present certain limitations. Writing code manually requires significant time and attention to detail. Repetitive tasks like creating boilerplate code, fixing syntax errors, and maintaining uniform code style often slow down the development process. These challenges are compounded in large projects where multiple developers work simultaneously, increasing the risk of inconsistencies and integration issues.

Recent advancements in Artificial Intelligence (AI) have begun to transform the way programming is approached. Natural Language Processing (NLP), in particular, has made it possible for machines to interpret and generate human-like text. Building upon this, **Specialized Language Models (SLMs)** have been developed to handle programming-specific tasks, enabling them to understand syntax, logical structures, and the semantics of multiple programming languages.

An SLM trained for coding is more than just a smart text editor. It can generate optimized code snippets, detect and correct logical errors, recommend best coding practices, and even provide explanations for complex algorithms. This reduces not only development time but also the chances of introducing errors into the codebase. For novice programmers, it serves as a learning companion, guiding them step-by-step through programming concepts while offering instant feedback.

Furthermore, SLMs can adapt to different programming paradigms and frameworks, making them suitable for a wide range of applications—from web development and data

analysis to embedded systems programming. This versatility allows developers to focus more on creative problem-solving rather than repetitive implementation tasks.

The **SLM for Coding** minor project aims to explore and demonstrate the capabilities of such a system in a practical environment. The primary goal is to develop an AI-powered coding assistant capable of supporting multiple programming languages, offering intelligent suggestions, automating repetitive tasks, and improving the overall coding experience.

During the course of the project, the SLM will be evaluated on parameters such as speed, accuracy, adaptability, and user-friendliness. Multiple coding scenarios will be tested, ranging from simple script generation to debugging and optimization of complex functions. This evaluation will help assess the true potential of AI-assisted coding in real-world applications.

Additionally, the project will address important ethical and practical considerations. These include the need to ensure that AI-generated code is original, free from licensing issues, and compliant with security standards. It will also explore the balance between automation and human oversight, ensuring that the technology complements rather than replaces human creativity and problem-solving skills.

By the end of this project, a working prototype of the **SLM for Coding** system will be presented. This prototype will serve as proof of concept for how AI-powered tools can improve efficiency, reduce errors, and accelerate the software development lifecycle. Ultimately, this work will contribute to the growing body of knowledge on integrating AI into the coding process, highlighting its potential to reshape the future of programming in both academic and professional contexts

1.1 Challenges

The rapid advancement of Large Language Models (LLMs) in code generation has set a high performance benchmark, yet their immense scale presents significant challenges in accessibility, computational cost, and deployment latency. This project, by focusing on a Small Language Model (SLM), confronts the central challenge of this performance-versus-efficiency trade-off: achieving robust functional correctness with a model intentionally constrained to 100-300 million parameters. A primary obstacle is the rigorous demand of data curation; sourcing and filtering a high-quality dataset from vast, noisy open-source repositories like The Stack is a complex data engineering task critical to model success. Furthermore, the project's goal of achieving functional correctness, measured by the pass@k benchmark, introduces a significant hurdle beyond simple syntactic accuracy. This requires the creation of a secure sandbox environment to safely execute untrusted, AI-generated code against unit tests—a substantial engineering challenge in itself. Finally, even with a smaller architecture, the training process demands careful management of limited GPU resources to effectively train the model to understand the complex logic and dependencies inherent in programming.

1.2 Need of Efficient Coding Model

The development of efficient coding models has become essential for the future of software engineering. It enables the democratization of advanced AI assistance, placing powerful tools directly into developer workflows without requiring massive computational resources. Through lightweight, low-latency models, companies can provide real-time code completion and bug detection, optimizing the development lifecycle. This understanding helps in reducing repetitive tasks, improving code quality, and accelerating project timelines, ultimately leading to increased productivity and innovation across the industry.

LITERATURE REVIEW

[1] The emergence of models like GPT-3 showcased the potential of LLMs in coding tasks, leading to specialized models trained extensively on code, such as OpenAI's Codex, which powers GitHub Copilot. Vaswani et al. (2017) in "Attention Is All You Need" introduced the transformer architecture, which became foundational for these advancements. Chen et al. (2021) in "Evaluating Large Language Models Trained on Code" demonstrated that LLMs could effectively solve introductory programming problems. However, their work also highlighted the significant computational resources and costs required for training and deploying such models.

[2] To address the challenges of LLMs, research has pivoted toward smaller, specialized models optimized for code-related tasks. CodeT5 (Wang et al., 2021) introduced a unified pre-trained encoder-decoder model capable of handling tasks like code generation and summarization with a compact architecture. Similarly, CodeGen (Nijkamp et al., 2022) explored the balance between model size and performance, showing that smaller models could achieve competitive results through targeted training on high-quality, domain-specific datasets.

[3] To enhance the efficiency of Small Language Models (SLMs), techniques such as knowledge distillation and quantization have been widely adopted. Knowledge distillation involves training a smaller "student" model to replicate the

behavior of a larger "teacher" model, effectively compressing knowledge into a more efficient form. Bucilă et al. (2006) provided foundational insights into model compression, which have been applied to modern LLMs to create distilled models that maintain much of the original performance while being significantly smaller and faster.

[4] Assessing the quality of generated code remains a complex challenge. Traditional NLP metrics, such as BLEU, are often inadequate for code evaluation. The HumanEval framework, introduced by Chen et al. (2021), proposed a robust evaluation approach based on functional correctness, where generated code is tested against a suite of unit tests. The pass@k metric has since become a standard for benchmarking code generation models.

Objectives and Scope of work

3.1 Objectives

The primary aim of this project is to develop and evaluate a highly efficient Small Language Model (SLM) specifically designed for practical coding assistance, focusing on robust performance and computational efficiency for real-world developer workflows. This will be achieved by creating an SLM architecture, likely leveraging transformer-based designs with knowledge distillation that emphasizes low latency, minimal memory footprint, and reduced computational overhead. The model will be trained on a carefully curated dataset of high-quality Python source code from reputable repositories to achieve deep domain expertise and the ability to generate accurate, idiomatic code. The resulting system will be capable of performing key assistance tasks like real-time code completion, function generation from natural language, and simple bug detection. Its performance will be comprehensively evaluated against larger LLMs using industry-standard metrics, such as the pass@k HumanEval benchmark for functional correctness and code BLEU for syntactic similarity, to quantify its efficiency gains. Finally, a proof-of-concept prototype, such as a web interface or IDE extension, will be created to demonstrate the SLM's practical utility.

Scope of Work

To ensure the project is achievable, its scope is clearly defined. The in-scope work focuses on specializing the model for a single programming language, Python, to ensure a deep understanding of its nuances. The model's core functionalities will be limited to real-time code completion and simple function generation from clear docstrings, trained on a publicly available, filtered dataset like "The Stack." Performance will be strictly measured by accuracy, functional correctness against unit tests, and inference speed. Conversely, the project will not attempt to generate entire applications, complex multi-file algorithms, or perform deep logical debugging, though it may identify simple syntax errors. The focus will remain on the model itself, not on building a scalable, production-grade cloud service.

Methodology

4.1 Data Collection

Architecture:

The model architecture will be a transformer-based, decoder-only structure, similar to GPT-2, which is adopted due to its proven effectiveness in generative tasks. A "small" model configuration will be explicitly chosen, targeting a parameter count significantly lower than large-scale models (e.g., in the range of 100-300 million parameters). This decision ensures the final model remains lightweight and efficient, prioritizing low latency and minimal resource footprint for practical deployment.

4.2 Data Preprocessing:

The data preparation process will begin with the selection of a large-scale, open-source code dataset, such as a subset of The Stack. This dataset will then undergo a rigorous filtering process to isolate only high-quality Python source code, using criteria such as the presence of valid licenses, adherence to code formatting standards, and the exclusion of auto-generated or low-quality files. Finally, the curated code will be preprocessed and tokenized using a specialized tokenizer designed specifically for programming languages, ensuring it can effectively handle syntax elements like indentation and special characters.

4.3 Model Selection:

For model selection, a transformer-based decoder-only architecture, similar to GPT-2, will be adopted due to its proven effectiveness in generative tasks. A "small" model configuration will be chosen, with a parameter count significantly lower than that of large-scale models (e.g., in the range of 100-300 million parameters). This ensures the model remains lightweight and efficient.

4.4 Model Training:

The model will first be pre-trained on the curated code dataset using a causal language modeling objective, which involves predicting the next token in a sequence. This initial training will be conducted on a cloud-based GPU platform, such as Google Colab Pro or Kaggle, where key hyperparameters like learning rate, batch size, and training steps will be systematically tuned.

Following pre-training, the model will be fine-tuned on a more specific dataset of instruction-based prompts (e.g., "write a Python function that...") to enhance its ability to understand and follow natural language commands.

4.5 Model Evaluation:

The model's performance will be comprehensively evaluated across three key dimensions: functional correctness, code quality, and efficiency. Functional correctness will be assessed using a benchmark similar to HumanEval, where the pass@k metric will be calculated to measure the percentage of problems for which at least one of k generated solutions passes the unit tests. Code quality will be measured using the BLEU score, which compares the generated code against reference solutions to provide a measure of syntactic similarity. Finally, efficiency will be quantified by measuring inference speed (tokens per second) and memory footprint to determine the model's suitability for real-time applications.

Fig. 1 Accuracy Graph & Loss Graph

Conclusion And Future Work

5.1 Conclusion:

This project will explore the development of a Small Language Model (SLM) for coding, addressing the critical need for efficient, accessible, and practical AI-powered developer tools. By moving away from the resource-intensive paradigm of Large Language Models, this work aims to demonstrate that a well-designed, specialized model can deliver significant value in real-world software development workflows. The successful implementation of an SLM for code completion and generation will confirm that high performance does not have to come at the cost of extreme computational expense.

The methodology outlined, from careful data curation to rigorous evaluation using industry-standard benchmarks, will provide a comprehensive framework for building and assessing such models. The final prototype will serve as a tangible proof-of-concept, showcasing the potential for SLMs to be integrated seamlessly into development environments, thereby boosting productivity, reducing repetitive work, and lowering the barrier to entry for new programmers. Ultimately, this project will contribute to the ongoing democratization of AI, making powerful coding assistance tools more sustainable and widely available.

5.2 Future Work:

While this project establishes a strong foundation, numerous avenues exist for future exploration. These include enhancing the model with multi-modal capabilities to understand UI mockups or diagrams for frontend code generation, and extending its scope beyond generation to include advanced debugging and automated code refactoring. Furthermore, the model could be personalized by fine-tuning it on a team's specific codebase to learn unique coding styles and private APIs. Further research into model compression techniques like quantization could enable on-device execution, ensuring privacy and offline functionality. Finally, a collaborative filtering system could be developed, allowing the model to learn from team interactions to suggest code snippets that are popular or highly-rated within an organization.

REFERENCES

[1]Chen, Mark, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, et al. "Evaluating Large Language Models Trained on Code." *arXiv*, arXiv:2107.03374, 2021.

[2]BibaultNijkamp, Erik, Bo Pang, Hiroaki Hayashi, Lifu Tu, Han Wang, Yingbo Zhou, et al. "CodeGen: A Conversational Paradigm for Program Synthesis." *arXiv*, arXiv:2203.13474, 2022.

[3]Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, et al. "Attention Is All You Need." *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[4]Wang, Yue, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. "CodeT5: Identifier-Aware Unified Pre-Trained Encoder-Decoder Models for Code Understanding and Generation." *arXiv*, arXiv:2109.00859, 2021.

[5]Austin, Jacob, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, et al. "Web-Scale Language Models for Program Synthesis." *arXiv*, arXiv:2103.03874, 2021.