IoTCoder: A Copilot for IoT Application Development

Leming Shen, Yuanqing Zheng
The Hong Kong Polytechnic University, Hong Kong SAR, China
leming.shen@connect.polyu.hk,csyqzheng@comp.polyu.edu.hk,

ABSTRACT

Existing code Large Language Models are primarily designed for generating simple and general algorithms but are not dedicated to IoT applications. To fill this gap, we present *IoTCoder*, a coding copilot specifically designed to synthesize programs for IoT application development. *IoTCoder* features three locally deployed small language models (SLMs): a *Task Decomposition SLM* that decomposes a complex IoT application into multiple tasks with detailed descriptions, a *Requirement Transformation SLM* that converts the decomposed tasks described in natural language to well-structured specifications, and a *Modularized Code Generation SLM* that generates modularized code based on the task specifications. Experiment results show that *IoTCoder* can synthesize programs adopting more IoT-specific algorithms and outperform state-of-the-art code LLMs in terms of both task accuracy (by more than 24.2% on average) and memory usage (by less than 358.4 MB on average).

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**.

KEYWORDS

Large Language Models, IoT Applications

ACM Reference Format:
Leming Shen, Yuanqing Zheng. 2024. IoTCoder: A Copilot for IoT Application Development. In *The 30th Annual International Conference On Mobile Computing And Networking (ACM MobiCom '24), November 18–22, 2024, Washington D.C., DC, USA*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3636534.3697447

1 INTRODUCTION

Large language models (LLMs) have changed our daily lives in various aspects, such as task automation and IoT data interpretation [1]. Code LLMs (*e.g.*, CodeLlama [2]) are promising tools designed to generate code according to user requirements described in natural language. By integrating code LLMs with coding tools, they serve as copilots that significantly improve software development by automating code generation, bug detection, and documentation tasks (Fig. 1).

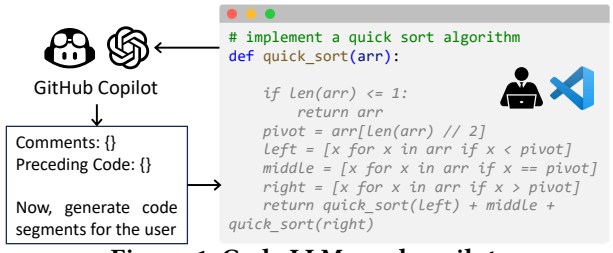


Figure 1: Code LLMs and copilot.

Existing code LLMs are mainly designed for general-purpose coding, *i.e.*, generating a single module or function based on well-defined specifications. When confronted with IoT applications that require specific domain knowledge, they tend to provide general solutions with sub-optimal performance rather than dedicated algorithms tailored for IoT. The reason is that IoT domain knowledge only occupies a small proportion of the training dataset of code LLMs. This observation motivates the following research question: *Can we build a code LLM tailored for IoT application development?* Constructing an IoT code LLM can improve user experiences with enhanced efficiency in IoT application development.

A potential solution is equipping LLMs with Retrieval-Augmented Generation (RAG). By providing LLMs with retrieved IoT domain knowledge, the generated code may adopt algorithms more relevant to the IoT domain. However, such methods suffer from three main problems. 1) Extensive human effort must be invested in the RAG design to ensure the correctness and high relevance of the retrieved knowledge. 2) Meticulously designed prompts are demanded to ensure that the output strictly follows pre-defined formats, which is extremely challenging due to hallucinations and unreliability [3]. 3) Even armed with specially designed RAG, the LLM still requires strong language understanding capability to learn from the retrieved contents, demanding a powerful LLM. However, cloud LLMs (*e.g.*, GPT-4) often suffer from long latency, high cost, and privacy concerns, while local LLMs (*e.g.*, CodeLlama) have harsh system resource requirements.

To tackle these problems, we propose *IoTCoder*, a copilot tailored for IoT application development by fine-tuning local SLMs on IoT-specific datasets. Specifically, the tuning process can embed IoT domain knowledge into the tuned

ACM MobiCom '24, November 18–22, 2024, Washington D.C., DC, USA

ACM ISBN 979-8-4007-0489-5/24/11…$15.00
https://doi.org/10.1145/3636534.3697447

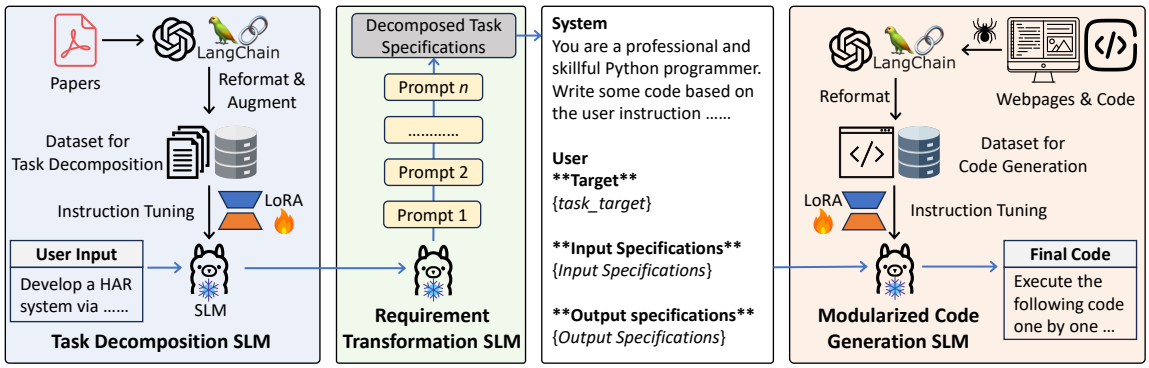


Figure 2: The system overview and workflow of *IoTCoder*.

model by steering the parameter distributions toward IoT-relevant context. As such, the SLM will provide solutions adopting more IoT-specific algorithms to solve users' problems, thereby enhancing task accuracy and relevance in the IoT domain. We can also improve the reliability of producing outputs following specified formats. By locally deploying the tuned SLMs with smaller model sizes, the privacy and latency concerns will be mitigated. As shown in Fig. 2, given an IoT programming task, the *Task Decomposition SLM* first breaks it into multiple tasks with descriptions. Next, the *Requirement Transformation SLM* converts the decomposed tasks described in natural language into well-structured task specifications. Accordingly, the *Modularized Code Generation SLM* generates a list of modularized codes with detailed documentation. Note that the task decomposition and code generation processes are based on two tuned local SLMs on our manually collected datasets tailored for the IoT domain.

2 SYSTEM DESIGN

2.1 Task Decomposition SLM

Following the software life cycle, we deploy an SLM to decompose the complex IoT application into multiple tasks.

Dataset Construction & Augmentation. IoT-related research papers contain high-quality algorithms and paradigms for IoT applications. Thus, we first download some papers as high-quality data sources, covering a wide range of IoT topics (*e.g.*, communication, wireless sensing, edge computing, *etc*). Next, we build an RAG agent by integrating GPT-4o with the papers. Then, for each technical module proposed in the paper, we instruct the agent to generate a list of implementation descriptions via specially designed prompts. Furthering this, we design an IoT-specific text data augmentation method to enhance the quantity, diversity, and creativity of the original data. Our augmentation considers: 1) For the same IoT task, we can use different sensor data modalities. For example, to perform human activity recognition (HAR), we can use IMU data, Wi-Fi CSI, *etc.* 2) For the same sensor modality, we can leverage distinct representations of the data. For instance, we can use Wi-Fi CSI, spectrograms, or Doppler features to implement HAR. 3) For the same task, different target platforms have various resource budgets (*e.g.*, memory). After augmentation, we obtain 39,000 "user requirement - decomposed task list" pairs in total. Note that this augmentation method considers the diversity of both language expression and IoT characteristics.

Instruction Tuning. We use Llama2-13b [4] as the local SLM and adopt supervised fine-tuning via LoRA. After several epochs of tuning, we can effectively transform the general model into a dedicated SLM tailored for IoT task decomposition. The tuned model can generate a list of decomposed tasks with descriptions following user instructions.

2.2 Modularized Code Generation SLM

We deploy a fine-tuned SLM to generate multiple code snippets for each decomposed task. By sequentially executing the generated code snippets, the complex IoT task can be solved via such a divide-and-conquer manner.

Dataset Construction. Open-source Python packages contain abundant hand-crafted algorithms with high performance for IoT tasks. Therefore, we select some Python packages as our data source, covering areas including signal processing, machine learning, and data visualization. We build a web crawler to automatically retrieve information from the package's official website and formulate three types of code generation tasks. 1) *Module description.* We format the user instruction to "Provide a detailed description of the `<module_name>` from a Python package named `<package_name>` to `<target>`." The corresponding reply contains comprehensive information of the module, including its prototype, description, specification, tips, and sample code. This aims to make the SLM familiar with the module by enhancing the correlation between user instruction and the sample code. 2) *Module implementation.* We format the user instruction to "Write some Python code with comments and documentation to `<target>` by using the `<module_name>` from a Python package named `<package_name>`." The corresponding reply includes the sample code with documentation with detailed information, including the workflow and guidance on how to execute the code. This aims to enhance the instruction-following capability of the SLM to generate code and documentation according to the specification. 3) *Example implementation.* We organize the user instruction to a

well-structured format, including the task target and the I/O specifications for the expected code. This aims to enhance the code generation capability of the SLM in following the well-structured task specifications. Ultimately, by concatenating and mixing all three types of code generation tasks, we obtain 35,339 "task specification - code & documentation" pairs in total for fine-tuning.

Instruction Tuning. After several epochs of fine-tuning via LoRA, the SLM can generate IoT-specific code following the user specifications. Further experiments reveal that our *Modularized Code Generation SLM* can synthesize more IoT-specific programs with better performance. Note that the code generation SLM shares the same foundation model (*i.e.*, Llama2-13b) with the *Task Decomposition SLM* without incurring significant resource overhead.

2.3 Requirement Transformation SLM

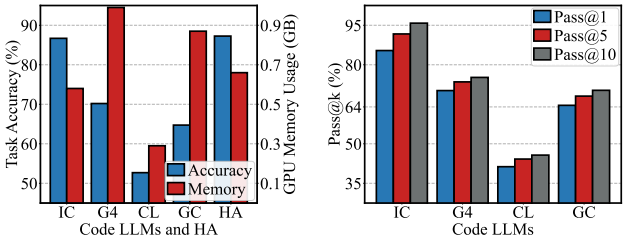
There exists a huge gap between the descriptions of the decomposed tasks and the expected inputs for the code generation SLM. The main reason is that the decomposed tasks are described in natural language, while the expected inputs for the code generation SLM should be well-structured specifications. Directly inputting the descriptions of decomposed tasks into the tuned code generation SLM cannot generate solutions with high performance. Therefore, we design a *Requirement Transformation SLM* to fill this gap. Specifically, we design a chain of prompts that guide the SLM to convert the task descriptions into well-structured specifications step by step. This leverages the basic language processing capability of the SLM and, therefore, does not require fine-tuning.

3 IMPLEMENTATION & EVALUATION

We implement and deploy *IoTCoder* on an edge server with an NVIDIA RTX 4090 GPU. For instruction tuning, we use a cloud server with an NVIDIA RTX A800 GPU.

Overall Performance. We select Wi-Fi-based HAR on the XRF55 dataset [5] as a representative IoT application [6]. From Fig. 3(a) we can see that the programs synthesized by *IoTCoder* achieve better performance than the baselines (GPT-4o, CodeLlama-34b, and GitHub Copilot) in terms of task accuracy (recognition accuracy) and system overhead (GPU memory usage). This observation indicates that by tuning local SLM on IoT-specific datasets, the synthesized programs can outperform SOTA code LLMs and even achieve similar performance to hand-crafted algorithms! This is because the programs synthesized by *IoTCoder* adopt more dedicated algorithms for IoT data processing and model optimization.

Code Generation Capability. We manually establish a benchmark specifically designed to evaluate the code generation capability for IoT applications. The IoT benchmark contains 100 IoT applications described in natural language with several test cases. Fig. 3(b) shows the average Pass@k values of *IoTCoder* and baselines, where a higher value indicates



(a) Overall performance (b) Code generation capability

Figure 3: Evaluation results (IC for *IoTCoder*, G4 for GPT-4o, CL for Code Llama, GC for GitHub Copilot, and HA for the hand-crafted algorithm).

a stronger code generation capability for IoT applications. We can see that the programs synthesized by *IoTCoder* even outperform GPT-4o in some cases. The performance gain stems from the embedded IoT domain knowledge during instruction tuning, ensuring that the tuned model assigns a higher probability to IoT-related corpus data and thereby generates IoT-specific solutions for the user.

4 CONCLUSION AND FUTURE WORK

We present *IoTCoder*, a tailored programming copilot that can synthesize programs with documentation according to the user requirements for IoT application development. *IoTCoder* features three locally deployed SLMs responsible for distinct tasks, *i.e.*, task decomposition, requirement transformation, and modularized code generation. Experiments demonstrate the effectiveness and prior performance of *IoTCoder* compared with SOTA code LLMs and GitHub Copilot.

The effectiveness of *IoTCoder* relies on the quality of the training data. Insufficient or biased data can lead to suboptimal performance. Besides, IoT technologies are emerging rapidly, requiring an evolving *IoTCoder* that continuously learns new domain knowledge. Therefore, ensuring the quality and diversity of the datasets while keeping them up-to-date is crucial. We plan to design an IoT-specific data quality enhancement approach with continuous learning to improve *IoTCoder*'s performance and adaptability.

REFERENCES

[1] H. Xu, L. Han, Q. Yang, M. Li, and M. Srivastava, "Penetrative ai: Making llms comprehend the physical world," in *ACM HotMobile*, 2024.
[2] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, *et al.*, "Code llama: Open foundation models for code," *arXiv:2308.12950*, 2023.
[3] C. Lin, Z. Han, C. Zhang, Y. Yang, F. Yang, C. Chen, and L. Qiu, "Parrot: Efficient serving of llm-based applications with semantic variable," *arXiv:2405.19888*, 2024.
[4] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv:2307.09288*, 2023.
[5] F. Wang, Y. Lv, M. Zhu, H. Ding, and J. Han, "Xrf55: A radio frequency dataset for human indoor action analysis," *ACM IMWUT*, 2024.
[6] L. Shen, Q. Yang, K. Cui, Y. Zheng, X.-Y. Wei, J. Liu, and J. Han, "Fedconv: A learning-on-model paradigm for heterogeneous federated clients," in *ACM MobiSys*, 2024.