GPIoT: Tailoring Small Language Models for IoT Program Synthesis and Development

Leming Shen[1], Qiang Yang[2], Xinyu Huang[1], Zijing Ma[1], Yuanqing Zheng[1]
[1]The Hong Kong Polytechnic University, [2]University of Cambridge
{leming.shen,unixy-xinyu.huang,zijing.ma}@connect.polyu.hk,qy258@cam.ac.uk,csyqzheng@comp.polyu.edu.hk,

ABSTRACT

Code Large Language Models (LLMs) enhance software development efficiency by automatically generating code and documentation based on user requirements. However, code LLMs cannot synthesize specialized programs when tasked with IoT applications that require domain knowledge. While Retrieval-Augmented Generation (RAG) offers a promising solution by fetching relevant domain knowledge, it necessitates powerful cloud LLMs (*e.g.*, GPT-4) to process user requirements and retrieved contents, which raises significant privacy concerns. This approach also suffers from unstable networks and prohibitive LLM query costs. Moreover, it is challenging to ensure the correctness and relevance of the fetched contents. To address these issues, we propose *GPIoT*, a code generation system for IoT applications by fine-tuning locally deployable Small Language Models (SLMs) on IoT-specialized datasets. SLMs have smaller model sizes, allowing efficient local deployment and execution to mitigate privacy concerns and network uncertainty. Furthermore, by fine-tuning SLMs with our IoT-specialized datasets, the SLMs' ability to synthesize IoT-related programs can be substantially improved. To evaluate *GPIoT*'s capability in synthesizing programs for IoT applications, we develop a benchmark, *IoTBench*. Extensive experiments and user trials demonstrate the effectiveness of *GPIoT* in generating IoT-specialized code, outperforming state-of-the-art code LLMs with an average task accuracy increment of 64.7% and significant improvements in user satisfaction.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; • **Computer systems organization** → **Embedded and cyber-physical systems**.

KEYWORDS

Small Language Model, IoT Program Synthesis, Fine-tuning

ACM Reference Format:
Leming Shen[1], Qiang Yang[2], Xinyu Huang[1], Zijing Ma[1], Yuanqing Zheng[1] . 2025. GPIoT: Tailoring Small Language Models for IoT Program Synthesis and Development. In *The 23rd ACM Conference on Embedded Networked Sensor Systems (SenSys '25), May 6–9, 2025, Irvine, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3715014.3722064

SenSys '25, May 6–9, 2025, Irvine, CA, USA

ACM ISBN 979-8-4007-1479-5/25/05…$15.00
https://doi.org/10.1145/3715014.3722064

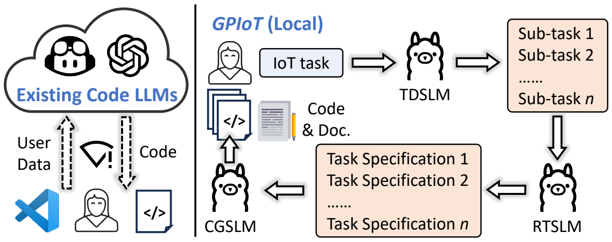


Figure 1: Existing code LLMs need to transmit sensitive data to remote servers. In contrast, *GPIoT* features three local SLMs to protect user privacy and reduce query costs.

1 INTRODUCTION

Large language models (LLMs) [65, 70] are revolutionizing various aspects of embedded system development and mobile computing, *e.g.*, smartphone task automation [74], advanced virtual assistants [46], and even IoT data comprehension [4, 41, 48, 79]. Code LLMs (*e.g.*, WizardCoder [43] and CodeLlama [53]) stand out as promising tools designed to synthesize programs based on user requirements described in natural language. As illustrated in Fig. 1, the integration of programming tools with code LLMs significantly enhances software development by automating code completion, code generation, bug detection, documentation writing, *etc.*

While powerful and promising, when confronted with IoT applications [6, 25, 26, 80–84] that require special domain knowledge, existing code LLMs tend to simply provide general solutions with sub-optimal performance (§ 2.2). This is because they focus on general-purpose programming tasks [34] rather than being tailored to any particular domain. Moreover, IoT-related knowledge and programs only occupy a small proportion of the datasets which code LLMs were trained on [85]. Consequently, IoT terminologies will be assigned a lower priority during inference with the generated code less dedicated to the IoT domain (§ 2.2). This motivates the following research question: *Can we build a code LLM specially tailored for IoT application code generation tasks?* If yes, we can synthesize IoT-related programs with higher task accuracy and efficiency, offering significant convenience for IoT developers.

A potential approach can be Retrieval-Augmented Generation (RAG) [36], which provides LLMs with retrieved domain knowledge to enhance their abilities in generating accurate and contextually relevant solutions. Existing works [17, 28, 56] construct a sophisticated LLM+RAG agent to gradually generate code through multiple intermediate steps via prompts. Nonetheless, they suffer from three main problems. 1) A powerful LLM with strong language comprehension capability is needed to learn from the retrieved knowledge. However, cloud LLMs (*e.g.*, GPT-4 [2]) may suffer from bad network conditions, high costs, and privacy concerns, while local LLMs (*e.g.*, Llama2-70b [61]) have harsh requirements in system resources (*e.g.*, memory and network). 2) Complicated RAG designs (*e.g.*, iterative retrieval [77]) are mandatory to ensure the correctness and high relevance of the retrieved knowledge, with extended processing time. Otherwise, LLMs may fail to focus on the IoT context and still provide general solutions [75]. 3) Meticulously designed prompts are required to ensure that outputs must strictly follow pre-defined formats [38], which is extremely challenging due to the hallucinations and unreliability of LLMs [62].

To tackle the above problems, we propose *GPIoT*, a code generation system tailored for IoT application development by fine-tuning local small language models[1] (SLMs) on IoT-specialized text-generation datasets. This approach has the following benefits: 1) The system overhead, privacy leakage, and network instability can be mitigated, as SLMs have smaller sizes and can be locally deployed without incurring heavy resource burdens. 2) SLMs tuned on IoT-specialized datasets can generate responses with significantly enhanced quality and higher relevance to the IoT domain [58]. 3) As our tuning datasets are well-structured text data, the tuned SLMs can produce intermediate outputs following the expected format with enhanced stability and avoid hallucinations.

We implement *GPIoT*[2] with three tailored SLMs to handle different stages of IoT application development: TDSLM for Task Decomposition, RTSLM for Requirement Transformation, and CGSLM for Code Generation. Though tuning one SLM to handle all the tasks is possible, it is extremely challenging due to SLMs' limited language understanding and processing capabilities [35, 42]. As shown in Fig. 1, TDSLM first decomposes an IoT application into multiple sub-tasks with detailed descriptions. Next, the descriptions are converted into well-structured specifications following pre-defined formats by RTSLM. Accordingly, CGSLM further generates a list of code snippets with detailed documentation. By sequentially executing the code based on the documentation, the IoT task can be solved. Note that we only fine-tune TDSLM and CGSLM as these two stages require IoT domain knowledge during inference while RTSLM only needs basic language processing.

In practice, we face three significant technical challenges. 1) **Lack of high-quality data.** To the best of our knowledge, there are no IoT-oriented user-requirement-to-sub-task and sub-task-to-program text-generation datasets. Thus, we first construct two datasets containing IoT knowledge retrieved from various public sources, aiming to enhance the task decomposition and code generation abilities of TDSLM and CGSLM, respectively. Moreover, we design an IoT-oriented text data augmentation method to enhance the datasets' quality and diversity, considering the unique characteristics (*e.g.*, sensor modality and resource heterogeneity) of IoT applications, thereby enhancing SLMs' knowledge comprehension and code generation capabilities for IoT tasks. 2) **Domain misalignment between SLMs.** The decomposed tasks generated by TDSLM may fall beyond the scope that CGSLM can handle due to domain misalignment. This is because the two SLMs focus on different stages of IoT application development during tuning, which could lead to thematic inconsistencies in task interpretation and execution (§ 2.2). To tackle this, we propose a parameter-efficient co-tuning (PECT) paradigm featuring a multi-path Low-Rank Adaptation (LoRA) pipeline. Unlike conventional LoRA tuning that tunes adapters separately, our designed PECT paradigm enables collaborative fine-tuning of multiple SLMs with a shared base model but with different adapters, thereby mitigating the inconsistency issues and facilitating knowledge sharing between SLMs. 3) **Format incompatibility.** Decomposed tasks are typically described in natural language, while the expected inputs of CGSLM should be well-structured. If we directly use the decomposed task descriptions as prompts to generate code, CGSLM can not provide programs strictly following user requirements (§ 2.2). To address this, we leverage Chain-of-Thought (CoT) prompting [73] that instructs RTSLM to transform the descriptions into well-structured specifications step by step. As such, CGSLM can better handle the specifications to provide IoT-specialized solutions.

To evaluate *GPIoT*, we also propose *IoTBench*, a benchmark to quantify LLMs' capabilities in synthesizing IoT-related programs. Extensive experiments and a user study demonstrate that *GPIoT* can generate code adopting more IoT-specialized algorithms and outperform SOTA code LLMs in terms of task accuracy (more than 64.7% on average), memory usage (less than 310 MB on average), and user satisfaction. In summary, we make the following contributions:

- *GPIoT* presents the first code generation system tailored for IoT application development featuring privacy-preserving local SLMs tuned on IoT-specialized datasets.
- We create IoT domain text-generation datasets with a novel augmentation method tailored for the unique characteristics of IoT tasks, significantly enhancing the IoT knowledge comprehension ability of our tuned SLMs. We also construct *IoTBench* to evaluate the capability of LLMs in synthesizing IoT-specialized programs.
- We propose the PECT paradigm, a new LLM tuning method that can collaboratively fine-tune multiple SLMs to mitigate their domain misalignment with facilitated knowledge exchange.

2 BACKGROUND & MOTIVATION

We first revisit existing code LLMs to underscore the importance of constructing tailored IoT-related LLMs. Then, we conduct some preliminary experiments on existing LLM+RAG methods to further motivate our work with several challenges we need to address.

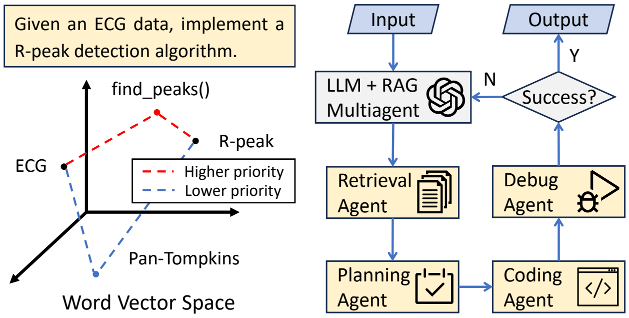
2.1 Code LLM and LLM+RAG

Existing code LLMs aim to synthesize programs and enhance software development efficiency and accuracy. While they perform well on general and simple programming tasks (*e.g.*, sorting algorithms), they often struggle with complex problems in the IoT domain. For example, when prompted to design an R-peak detection method for electrocardiogram (ECG) data, existing code LLMs can only use the `find_peaks()` function, which adopts a general peak detection algorithm rather than a dedicated one tailored for ECG data (*e.g.*, Pan-Tompkins [50]). The underlying reason is that IoT knowledge and programs only occupy a small proportion of the training dataset of code LLMs. As a result, despite being presented with abundant IoT terminologies in the prompt, LLMs still tend to prioritize and respond with more general words, due to their higher similarity (shorter distance in Fig. 2(a)) within the vector representation space.

LLM+RAG methods address this by retrieving domain knowledge for reference and establishing multiple cascaded agents to

[1] We consider SLMs as open-source language models that can be locally deployed and operated efficiently on commodity GPUs [29] (*e.g.*, Llama2-13b with INT8 quantization requires around 16 GB of GPU memory).

[2] The models and datasets are available at https://github.com/lemingshen/GPIoT



(a) Terminologies in vector space (b) General workflow of LLM+RAG

Figure 2: (a) Existing LLMs tend to prioritize general terms; (b) LLM+RAG systems require multiple cascaded agents.

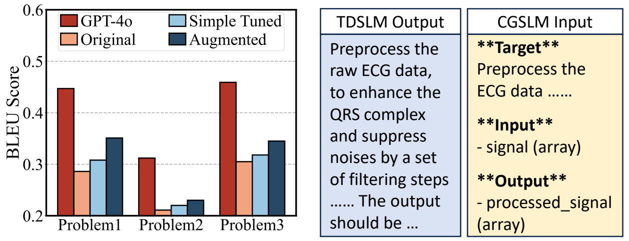
facilitate information transfer among modules. For example, in Fig. 2(b), multiple LLM-based agents are employed for different tasks during development (*i.e.*, domain knowledge retrieval, task planning, coding, and debugging). Sophisticated prompt design and meticulously structured intermediate outputs are necessary to ensure that one agent's output can be accurately parsed and interpreted by another agent. We conduct a preliminary experiment by prompting MapCoder [30], a multi-agent-based LLM+RAG framework, to synthesize programs for R-peak detection. We repeatedly generate 100 distinct versions of the programs and analyze them through code review and execution. Surprisingly, we find that only 28% of the programs adopt appropriate IoT-related algorithms to perform R-peak detection. This is because LLM+RAG requires sophisticated RAG design and user prompts. Otherwise, the retrieved knowledge is less accurate and relevant to the IoT context, and LLMs may fail to focus on the IoT domain and still provide simple and general solutions [75]. Moreover, this cascading process inevitably introduces noise and propagates errors [76], leading to a long self-recover time. More importantly, cloud LLMs suffer from bad network conditions, high costs, and privacy concerns.

To overcome these challenges, *GPIoT* fine-tunes SLMs on IoT-specialized text-generation datasets, as SLMs have smaller sizes and can be locally deployed. Additionally, by steering the parameter distribution towards the IoT domain via tuning, SLMs can focus on IoT-related semantic context, generating highly relevant responses that follow pre-defined formats with enhanced stability.

2.2 Preliminary Experiments & Findings

We conduct some preliminary experiments by separately fine-tuning two SLMs on our manually constructed datasets (§ 4.1), *i.e.*, the task decomposition dataset (TDD) and the code generation dataset (CGD). TDD aims to enhance TDSLM's capability to break a *problem statement* proposed by the user into multiple *decomposed tasks* described in natural language. CGD aims to enhance CGSLM's ability to generate *code & documentation* for the user based on the *decomposed tasks*. However, we find it extremely challenging to ensure the correctness of the generated code. Note that we use Llama2-13b [61] as the default SLM for demonstration purpose.

Lack of high-quality data. Directly fine-tuning SLMs incurs little performance gain, even with data augmentation. We first deploy four models: GPT-4o, the original SLM, the SLM tuned on TDD, and the SLM tuned on augmented TDD via Evol-Instruct [78]. Then, we randomly select three IoT problems from TDD and input them into the four models to obtain a set of responses. Next, we measure the



Preprocess the raw ECG data, to enhance the QRS complex and suppress noises by a set of filtering steps ...... The output should be ...

Target
Preprocess the ECG data ......

Input
- signal (array)

Output
- processed_signal (array)

(a) BLEU scores (b) Requirement-specification gap

Figure 3: (a) Directly tuning SLM with simple augmentation only yields small improvements; (b) Gap between SLMs.

similarity between the generated responses and the human-crafted references as ground truth (*decomposed tasks*) using the BLEU score [51], where a larger value indicates higher semantic similarity. As shown in Fig. 3(a), GPT-4o achieves the highest score with an acceptable value for such a text-generation task. However, the scores achieved by tuned SLMs increase slightly even with augmented TDD. Further analysis reveals that the solutions provided by the SLMs are either irrelevant to the IoT domain or contain hallucinations. This is because traditional augmentation methods (*e.g.*, Evol-Instruct) focus on augmenting linguistic characteristics of the original text data, which may fall short of effectively capturing intricate relationships among IoT terminologies. *This motivates us to design an IoT-tailored text augmentation method to enhance the quantity, quality, and diversity of the original dataset.*

Domain misalignment. Since TDSLM and CGSLM are tuned on distinct datasets for different tasks, domain misalignment occurs when used in tandem. Specifically, we feed the task descriptions generated by TDSLM into CGSLM to synthesize corresponding programs for each sub-task. Surprisingly, we find that only 53.4% of the programs can be successfully executed without bugs and only 10.6% of the programs adopt IoT-specialized algorithms for the IoT tasks. The main reason is that the two SLMs develop expertise in different domains with knowledge inconsistency during tuning, hindering the seamless integration of task decomposition and code generation. As a result, the responses generated by TDSLM may fall outside the scope that CGSLM can handle. *This motivates us to develop a knowledge-sharing strategy between the two SLMs during tuning so that they can reach a consensus when handling IoT tasks.*

Format incompatibility. TDSLM's outputs (*decomposed tasks*) are described in natural language while CGSLM's inputs (*task specifications*) should be well structured (Fig. 3(b)). When we directly feed TDSLM's output into CGSLM, only 23.6% of the synthesized programs can be successfully executed. The rest exhibits higher uncertainty with a lack of confidence in mapping the input task specification to the desired code [86]. The reason is that CGSLM is more sensitive to well-formatted inputs as it has been tuned on our dataset with structured text. Though directly tuning TDSLM to generate well-structured task specifications can be a solution, we find it challenging due to the limited language processing capabilities of SLMs, which cannot be sufficiently enhanced through tuning alone. *This motivates us to develop a method to convert the task descriptions in natural language into well-organized specifications.*

To address the above challenges, we propose three key technical modules, *i.e.*, an IoT-oriented text data augmentation method, a Parameter-Efficient Co-Tuning (PECT) paradigm with a multi-path LoRA pipeline, and a requirement transformation module.

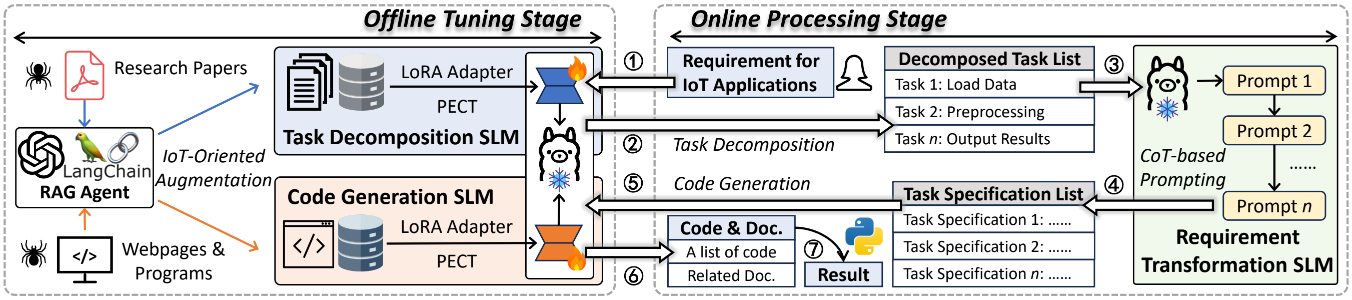


Figure 4: The system overview and workflow of *GPIoT* (All the local SLMs share the same foundation model).

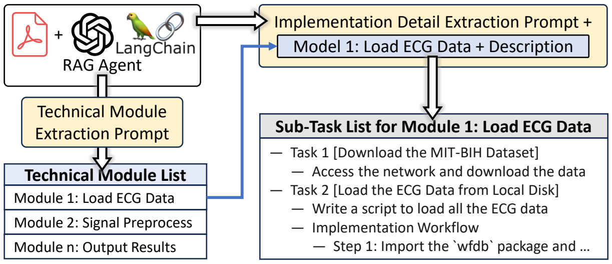

+
LangChain
RAG Agent
Technical Module
Extraction Prompt
Technical Module List
Module 1: Load ECG Data
Module 2: Signal Preprocess
Module n: Output Results
Implementation Detail Extraction Prompt +
Model 1: Load ECG Data + Description
Sub-Task List for Module 1: Load ECG Data
— Task 1 [Download the MIT-BIH Dataset]
— Access the network and download the data
— Task 2 [Load the ECG Data from Local Disk]
— Write a script to load all the ECG data
— Implementation Workflow
— Step 1: Import the `wfdb` package and ...

Figure 5: Task decomposition dataset construction.

3 SYSTEM OVERVIEW

Fig. 4 illustrates the overall architecture of *GPIoT*, consisting of an offline tuning stage and an online processing stage.

Offline Stage. The offline tuning stage (the left part in Fig. 4) constructs two IoT-specialized datasets and fine-tunes TDSLM and CGSLM, which will be used for task decomposition and code generation in the online stage, respectively. We first build a RAG agent to extract knowledge and programs from various IoT-related public sources (*e.g.*, websites and articles) to **construct high-quality datasets**. Then, we augment the datasets by adopting our IoT-oriented augmentation method (§ 4.1) to enhance their quantity, quality, and diversity. Note that the RAG agent is only used for high-quality dataset construction during the offline stage. With the two augmented datasets, we fine-tune two SLMs via our PECT paradigm, where certain model parameters are collaboratively tuned through a multiple-path LoRA pipeline with two projection layers for task decomposition and code generation, respectively. Our PECT paradigm **mitigates the domain misalignment** between TDSLM and CGSLM with facilitated knowledge transfer and sharing.

Online Stage. The online stage (the right part in Fig. 4) aims to synthesize IoT-specific programs based on the user requirement for an IoT application development. Specifically, *GPIoT* first leverages *Task Decomposition SLM* (TDSLM) to decompose the IoT application into multiple manageable sub-tasks with detailed descriptions (①~②). Next, through CoT-based prompting techniques, the sub-task descriptions will be gradually **transformed into well-structured specifications** by *Requirement Transformation SLM* (RTSLM) (③~④). Next, for each sub-task, *Code Generation SLM* (CGSLM) accordingly generates a code snippet with documentation (⑤~⑥). Users can execute the code sequentially to realize the IoT application based on the instructions from the documentation (⑦).

SLM Considerations. We consider SLMs as open-source models that can be locally deployed and operated efficiently on commodity GPUs (*e.g.*, RTX 4070 Ti). This aligns with the practical constraints of normal users, where local models offer advantages in terms of cost, privacy, and independence from the cloud. Note that although there are three SLMs working simultaneously, *they share the same foundation model and differ only in some additional tunable parameters*, which only occupy 1% of all the parameters. Such a low-cost tuning and inference process stems from our PECT paradigm, avoiding significant overhead when deploying *GPIoT* on local devices.

4 SYSTEM DESIGN

4.1 Data Collection & Augmentation

Since there are no text-generation datasets in the IoT domain, we need to first construct a task decomposition dataset (TDD) and a code generation dataset (CGD). Note that our datasets contain Q&A pairs in textual form, fundamentally differing from conventional IoT datasets that typically contain pairs of sensor data and labels.

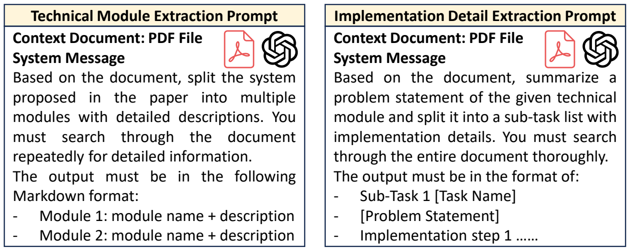
4.1.1 ***Task Decomposition Dataset.*** TDD contains pairs of "*problem statement* → *decomposed tasks*", aiming to enhance TDSLM's task decomposition ability for IoT problems. The construction process consists of three stages: raw IoT-related text data collection, data formatting, and IoT-oriented text data augmentation.

Raw Data Collection. IoT-related research papers contain a huge quantity of high-quality SOTA applications and algorithms. Moreover, the systems proposed are comprehensive and functional, which can be decomposed into multiple modules with clear motivation and implementation details. Therefore, we download IoT-related papers from several public literature databases[3] as our high-quality data sources, covering a wide range of IoT topics, such as communication, wireless sensing, edge computing, *etc.*

Data Formatting. We need to extract IoT knowledge from the papers and format it to pairs of "*problem statement* → *decomposed tasks*". Intuitively, we can regard the proposed system in each paper as an IoT problem, with its technical modules as the corresponding decomposed tasks. However, two challenges occur if we directly use such "*System* → *technical modules*" pairs for tuning: 1) These module descriptions are typically lengthy, which exceed the context length of SLMs [11]. 2) These modules are still sophisticated, often containing multiple sub-systems. TDSLM may struggle to extract IoT-specialized technical concepts and accurately generate manageable components. To tackle this, our insight is that we regard each module as an individual problem, which can be further decomposed into several manageable sub-tasks. The disintegrated sub-tasks can be easily handled by TDSLM with reduced context length.

Fig. 5 shows the entire process of how we extract pairs of "*problem statement* → *decomposed tasks*" from the papers. Specifically, we first build a RAG agent by combining the downloaded papers with an LLM (GPT-4o). Based on the provided context documents,

[3] We download papers from public databases via our institution's certification. The papers are for research only, adhering to ethical standards.



Technical Module Extraction Prompt

Context Document: PDF File

System Message

Based on the document, split the system proposed in the paper into multiple modules with detailed descriptions. You must search through the document repeatedly for detailed information.

The output must be in the following Markdown format:

- Module 1: module name + description
- Module 2: module name + description

Implementation Detail Extraction Prompt

Context Document: PDF File

System Message

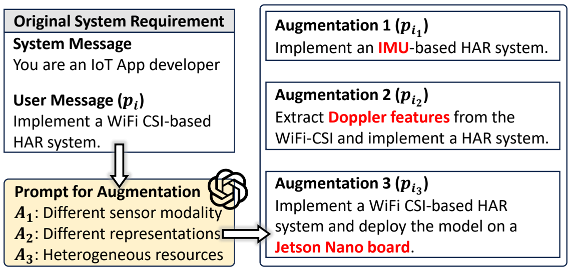
Based on the document, summarize a problem statement of the given technical module and split it into a sub-task list with implementation details. You must search through the entire document thoroughly.

The output must be in the format of:

- Sub-Task 1 [Task Name]
- [Problem Statement]
- Implementation step 1 ……

(a) Technical module extraction (b) Implementation detail extraction

Figure 6: Prompts for paper information extraction.



Original System Requirement

System Message

You are an IoT App developer

User Message (p_i)

Implement a WiFi CSI-based HAR system.

Prompt for Augmentation

A_1: Different sensor modality

A_2: Different representations

A_3: Heterogeneous resources

Augmentation 1 (p_{i_1})

Implement an **IMU**-based HAR system.

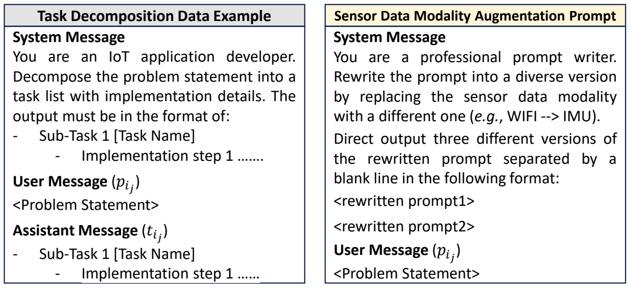
Augmentation 2 (p_{i_2})

Extract **Doppler features** from the WiFi-CSI and implement a HAR system.

Augmentation 3 (p_{i_3})

Implement a WiFi CSI-based HAR system and deploy the model on a **Jetson Nano board**.

Figure 7: Examples of IoT-oriented data augmentation with different modalities, representations, and resource budgets



Task Decomposition Data Example

System Message

You are an IoT application developer. Decompose the problem statement into a task list with implementation details. The output must be in the format of:

- Sub-Task 1 [Task Name]
  - Implementation step 1 …….

User Message (p_{i_j})

<Problem Statement>

Assistant Message (t_{i_j})

- Sub-Task 1 [Task Name]
  - Implementation step 1 ……

Sensor Data Modality Augmentation Prompt

System Message

You are a professional prompt writer. Rewrite the prompt into a diverse version by replacing the sensor data modality with a different one (*e.g.,* WIFI --> IMU).

Direct output three different versions of the rewritten prompt separated by a blank line in the following format:

<rewritten prompt1>

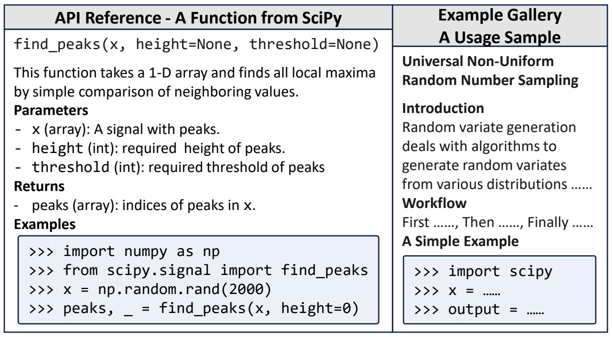
<rewritten prompt2>

User Message (p_{i_j})

<Problem Statement>

(a) Task decomposition data sample (b) Augment sensor modality

Figure 8: (a) Tuning data sample for task decomposition. (b) Prompt for sensor modality augmentation.



API Reference - A Function from SciPy

`find_peaks(x, height=None, threshold=None)`

This function takes a 1-D array and finds all local maxima by simple comparison of neighboring values.

Parameters

- x (array): A signal with peaks.
- `height` (int): required height of peaks.
- `threshold` (int): required threshold of peaks

Returns

- peaks (array): indices of peaks in x.

Examples

```
>>> import numpy as np
>>> from scipy.signal import find_peaks
>>> x = np.random.rand(2000)
>>> peaks, _ = find_peaks(x, height=0)
```

Example Gallery
A Usage Sample

Universal Non-Uniform Random Number Sampling

Introduction

Random variate generation deals with algorithms to generate random variates from various distributions ……

Workflow

First ……, Then ……, Finally ……

A Simple Example

```
>>> import scipy
>>> x = ……
>>> output = ……
```

Figure 9: An example of a Python package's website.

we then prompt (Fig. 6(a)) the agent to split the proposed system in the paper into multiple technical modules with detailed descriptions. Next, for each technical module, we prompt (Fig. 6(b)) the agent to further decompose it into several sub-tasks with detailed implementation steps. As such, we encapsulate the *problem statement* p_i of each technical module and the corresponding sub-tasks t_i into a Q&A pair Q_i to construct a raw dataset $\mathcal{D}_t$:

$$\mathcal{D}_t = \{Q_1, Q_2, \cdots, Q_{n_t}\},\ Q_i = (p_i, t_i) \tag{1}$$

where n_t is the total number of technical modules from all the papers. Fig. 8(a) shows a data sample from the dataset. Note that each sub-task is separated by a blank line, allowing us to parse and split t_i into multiple task description strings for further code generation in a divide-and-conquer way.

IoT-Oriented Data Augmentation. As revealed in § 2.2, existing text augmentation methods are ineffective in the IoT domain as they focus on expanding language characteristics rather than IoT knowledge. As a result, IoT terminologies are still assigned lower priority during inference, preventing the tuned model from generating IoT-specialized solutions. To address this, we propose a novel IoT-oriented data augmentation method that considers unique properties of IoT applications, *i.e.*, sensor modality, data representation, and system resource heterogeneity, as shown in Fig. 7.

Our augmentation considers three aspects: 1) *Sensor modality.* For the same IoT problem, we can use different sensor modalities. For instance, to implement human activity recognition (HAR), we can utilize IMU data, WiFi CSI, *etc.* 2) *Data representation.* For the same modality, we can leverage distinct data representations to achieve the same task. For example, we can use WiFi CSI, 2D spectrograms, or extracted Doppler features to implement HAR. 3) *Resource heterogeneity.* For the same task, various IoT devices require heterogeneous system resources. When deploying an AI model, smartphones typically have less memory than PCs, requiring model optimization methods. Based on the three aspects, we prompt (Fig. 8(b)) GPT-4o to rewrite and augment each *problem statement* from $\mathcal{D}_t$. To generate reference decomposed tasks, we build a search agent to retrieve relevant IoT domain knowledge and prompt it to produce results. We then manually filter out incorrect results and craft the formats (*i.e.*, each sub-task is separated by a blank line as aforementioned). The augmented dataset $\mathcal{D}'_t$ is:

$$\mathcal{D}'_t = \bigcup_{j}^{3} \{(p_{i_j}, t_{i_j}) \mid p_{i_j} = A_j(p_i),\ t_{i_j} = G(p_{i_j})\},\ \forall p_i \in \mathcal{D}_t \tag{2}$$

where $A_j(\cdot)$ is the j-th type of augmentation operation and $G(\cdot)$ is the black-box function of GPT-4o.

Remark. We take the diversity of both language expression and IoT characteristics into account, demonstrating significant performance improvement in task decomposing (§ 6.5). Note that the data collection and augmentation processes are both performed offline.

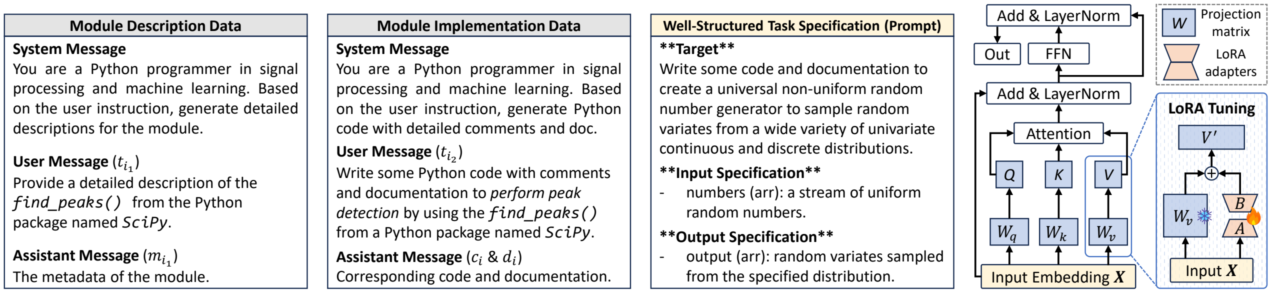
4.1.2 *Code Generation Dataset.*

CGD contains pairs of "*task specification* ⟶ *code & documentation*", aiming to enhance CGSLM's ability in generating IoT-related code for decomposed tasks. The data construction includes two stages: raw data collection and target diversity-aware augmentation for different code generation tasks.

Raw Data Collection. Open-source IoT-related Python[4] packages (*e.g.*, SciPy [64]) contain abundant hand-crafted IoT algorithms and applications with high performance, which can serve as our data sources. Thus, we first collect numerous public repositories from GitHub and extract Python packages they used, covering areas of signal processing, machine learning, and data processing (IoT data I/O and visualization). We then build a web crawler to automatically retrieve information from each package's official website.

A package's website (Fig. 9) typically contains two parts: 1) *API reference* includes a list of modules (*i.e.*, functions and classes) with comprehensive guidance on how to use them effectively in code. For example, `find_peaks()` is a function from the SciPy package, which identifies the local maxima (peaks) in an input signal array and returns the indices of the peaks. We denote the detailed information of each module as its metadata, m_i. 2) *Example gallery*

[4] We focus on Python since it is a cross-platform programming language.



Module Description Data

System Message
You are a Python programmer in signal processing and machine learning. Based on the user instruction, generate detailed descriptions for the module.

User Message (t_{i_1})
Provide a detailed description of the *find_peaks()* from the Python package named *SciPy*.

Assistant Message (m_{i_1})
The metadata of the module.

(a) Module description

Module Implementation Data

System Message
You are a Python programmer in signal processing and machine learning. Based on the user instruction, generate Python code with detailed comments and doc.

User Message (t_{i_2})
Write some Python code with comments and documentation to *perform peak detection* by using the *find_peaks()* from a Python package named *SciPy*.

Assistant Message (c_i & d_i)
Corresponding code and documentation.

(b) Module implementation

Well-Structured Task Specification (Prompt)

Target
Write some code and documentation to create a universal non-uniform random number generator to sample random variates from a wide variety of univariate continuous and discrete distributions.

Input Specification
- numbers (arr): a stream of uniform random numbers.

Output Specification
- output (arr): random variates sampled from the specified distribution.

(c) Task specification

(d) Traditional LoRA tuning

Figure 10: (a) & (b) Two tuning data samples. (c) Well-structured task specification. d) Traditional LoRA tuning.

provides practical usage samples of how to use various modules and features of the package to implement specific algorithms. For instance, a usage sample provides detailed documentation with code about performing universal non-uniform random number sampling using the SciPy package in an end-to-end manner. We denote the detailed information of each usage sample as its metadata, u_j. By combing these two types of metadata, we form a raw dataset $\mathcal{D}_c$:

$$\mathcal{D}_c = \{m_i \mid \forall i \in \{1, 2, \cdots, n_m\}\} \cup \{u_j \mid \forall j \in \{1, 2, \cdots, n_u\}\} \quad (3)$$

where n_m and n_u is the total number of modules and usage samples. **Target Diversity-Aware Augmentation.** This augmentation aims to enhance the diversity of the metadata in $\mathcal{D}_c$. We target each module in the packages for two text-generation tasks: 1) *Module Description*: providing detailed descriptions of a module and 2) *Module Implementation*: writing code & documentation to demonstrate usage samples of the module. aSince the example gallery already contains abundant algorithms with sample code and detailed descriptions, they can be directly used as code generation tasks.

1) *Module Description.* We format the task specification to "Provide detailed descriptions of <module> from <package>." The corresponding reply contains the module metadata in a pre-defined format. This Q&A mapping relation from a *task specification* t_i to a *module description* m_i is expressed as:

$$\mathcal{D}_1 = \{t_i \to m_i\} \quad (4)$$

Such a "*task specification* → *module description*" mapping can teach CGSLM to be familiar with the module's information, strengthening the semantic correlation between the module's name and the detailed descriptions. Fig. 10(a) illustrates a data sample from $\mathcal{D}_1$.

2) *Module Implementation.* We format the task specification to "Write some Python code with comments and documentation to perform <target> by using <module> from <package>." The corresponding reply contains the sample code and documentation that provides the workflow and guidance on how to execute the code. To obtain well-structured documentation, we prompt GPT-4o to format the module's metadata into Markdown. This Q&A mapping relation from a *task specification* t_i to the corresponding *module implementation* (*i.e.*, code c_i and documentation d_i) is:

$$\mathcal{D}_2 = \{t_i \to (c_i, d_i) \mid d_i = G(m_i)\} \quad (5)$$

Such a "*task specification* → *module implementation*" mapping relationship aims to enhance CGSLM's capability in generating code and documentation according to the module specification. Fig. 10(b) shows a data sample from $\mathcal{D}_2$.

3) *Example Implementation.* We format the task specification to a well-structured Markdown format as shown in Fig. 10(c), including the task target and the I/O specifications for the expected code. Correspondingly, we prompt GPT-4o to convert the usage sample's metadata into well-structured documentation. This Q&A mapping relation from a *task specification* t_j to the code c_j and documentation d_j can be expressed as:

$$\mathcal{D}_3 = \{t_j \to (c_j, d_j) \mid d_j = G(u_j)\} \quad (6)$$

This aims to enhance CGSLM's ability in generating IoT-related code and detailed documentation following well-structured task specifications. Ultimately, by concatenating all three augmented datasets, the final CGD $\mathcal{D}'_c$ becomes:

$$\mathcal{D}'_c = \mathcal{D}_1\{t_i \to m_{i_1}\} \cup \mathcal{D}_2\{t_i \to (c_i, d_i)\} \cup \mathcal{D}_3\{t_j \to (c_j, d_j)\} \quad (7)$$

4.1.3 ***IoTBench***. To evaluate LLMs' abilities in task decomposition and code generation for IoT applications, we create *IoTBench*, a benchmark of text-generation tasks in the IoT domain. Specifically, we choose 100 samples from TDD and CGD with manually created test cases, covering various IoT topics (*e.g.*, signal processing, edge AI, *etc.*). All the selected data samples are first manually filtered to ensure correctness and relevance to the IoT domain. Then, we format the sub-tasks separated by a blank line in between. Note that although many SOTA benchmarks (*e.g.*, HumanEval [10]) can also evaluate LLMs' code generation abilities, they are not tailored to IoT tasks. Besides, the data in *IoTBench* is excluded from the tuning processes (§ 4.2) to test the generalizability of the tuned SLMs.

4.2 Parameter-Efficient Co-Tuning (PECT)

With the two augmented datasets ($\mathcal{D}'_t$ and $\mathcal{D}'_c$), our next step is to fine-tune TDSLM and CGSLM to enhance their ability in task decomposition and code generation, respectively. In the following, we first introduce the traditional LoRA tuning method [27] for SLMs, and then explain our Parameter-Efficient Co-Tuning paradigm.

4.2.1 ***LoRA Tuning.*** Fig. 10(d) shows the Low-Rank Adaptation (LoRA) tuning process of a Transformer block [63]. Specifically, each Transformer block in an LLM contains two main components: a self-attention mechanism and a feed-forward network (FFN), both of which are followed by residual connections and layer normalization. The self-attention features three tunable weight matrices (W_q, W_k, and W_v) to capture contextual relationships between input embeddings, while the FFN processes the outputs from the attention mechanism to refine the feature representations. In conventional LLM full-tuning, the entire weight matrices are updated, leading to extensive GPU memory requirements and high computational costs. Instead of fully updating the weight matrices, LoRA reduces the number of tunable parameters [15, 18], where two low-rank matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ (*i.e.*, LoRA adapters) are inserted alongside the weight matrix. Given an input $\boldsymbol{X}$, the tuning process can be expressed as:

$$\boldsymbol{V}' = (\boldsymbol{W}_v + \boldsymbol{B}\boldsymbol{A}) \cdot \boldsymbol{X} \quad (8)$$

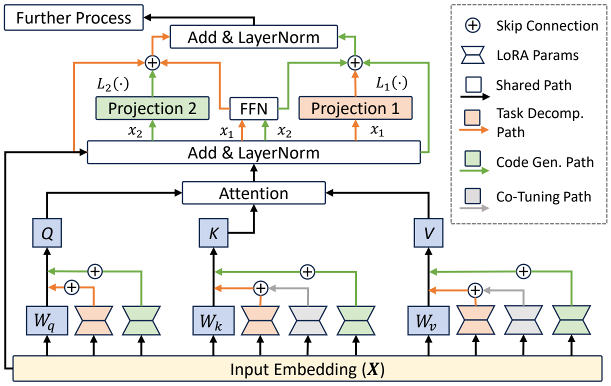


Figure 11: PECT in one Transformer block with both independently and collaboratively tuned LoRA adapters.

This reduces the computational burden by only updating the smaller low-rank matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, significantly cutting down resources.

However, as demonstrated in § 2.2, domain misalignment arises when separately tuning TDSLM and CGSLM using this vanilla LoRa method. This is because they focus on two distinct text-generation tasks with different semantic attention, thereby hindering *GPIoT* from synthesizing IoT-related programs. To tackle this issue, we propose a parameter-efficient co-tuning (PECT) paradigm. ***Unlike conventional LoRA tuning that tunes adapters separately, PECT enables collaborative fine-tuning of several SLMs with a shared base model but with different LoRA adapters.*** PECT features a Multi-Path LoRA Pipeline (MPLP) and two lightweight projection layers, which can promote information sharing between TDSLM and CGSLM, thereby narrowing the semantic comprehension gap between task decomposition and code generation.

4.2.2 *Multi-Path LoRA Pipeline.*

MPLP selects a subset of shared LoRA adapters to be collaboratively tuned by TDSLM and CGSLM, with another adapter set independently tuned.

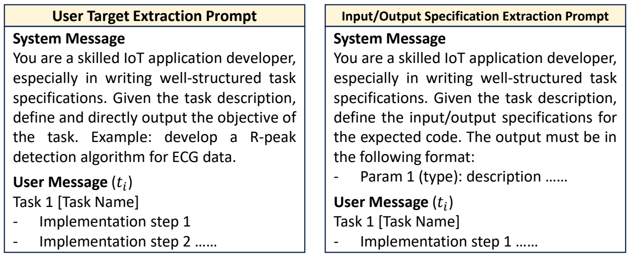
Pipeline Construction. In the lower part of Fig. 11, we create three pipelines of LoRA adapters in each Transformer block. Two pipelines (the orange one and the green one) are independently tuned by TDSLM and CGSLM with respect to TDD and CGD. The other pipeline (the gray one) is co-tuned on both TDD and CGD. For example, given a data sample from TDD, only the orange LoRA adapters and gray LoRA adapters are updated, as shown in Fig. 11. Note that we only assign the shared adapters beside the key and value weight matrices ($\boldsymbol{W}_k$, $\boldsymbol{W}_v$). The insight behind this is that the value vector provides the information to be activated based on the key vector [63]. In other words, the mapping from *problem statement* to *decomposed tasks* and the mapping from *task specification* to *code & documentation* are determined by the key and value vectors in TDSLM and CGSLM, respectively. Domain misalignment is thus caused by such different mapping relations during tuning on distinct datasets with disparate semantic focuses. Therefore, by co-tuning the shared adapters integrated into the key and value vectors, the mapping relations will be shared between the two SLMs, allowing TDSLM's outputs to align with CGSLM's scope.

Co-Tuning. In Fig. 11, we designate the orange line as the task decomposition path (TDP), through which only data from TDD will pass. The green line is the code generation path (CGP), through which only data from CGD will pass. The grey line represents the co-tuning path, through which all data will pass. During co-tuning, the LoRA adapters will be tuned either independently or collaboratively, depending on the path they occupy. Specifically, take the adapter alongside the projection matrix $\boldsymbol{W}_k$ as an example, the key vectors after projection in the two paths are calculated by:

$$\begin{aligned}\boldsymbol{K}_1 &= (\boldsymbol{W}_k + \boldsymbol{B}_1\boldsymbol{A}_1 + \lambda \cdot \boldsymbol{B}_c\boldsymbol{A}_c) \cdot \boldsymbol{X}\\ \boldsymbol{K}_2 &= (\boldsymbol{W}_k + \boldsymbol{B}_2\boldsymbol{A}_2 + (1-\lambda) \cdot \boldsymbol{B}_c\boldsymbol{A}_c) \cdot \boldsymbol{X}\end{aligned} \tag{9}$$

where $\boldsymbol{X}$ is the input text embedding, $\boldsymbol{K}_1$ and $\boldsymbol{K}_2$ are the key vectors within TDP and CGP, respectively. $\boldsymbol{B}_1\boldsymbol{A}_1$ and $\boldsymbol{B}_2\boldsymbol{A}_2$ are the parameters of LoRA adapters independently tuned within the two paths, respectively. $\boldsymbol{B}_c\boldsymbol{A}_c$ are the LoRA adapters collaboratively tuned by the two paths. λ is a hyper-parameter to balance the data flow between the two paths. During the co-tuning process, we first randomly sample data from TDD and CGD. Next, if the data is sampled from TDD, it will pass through TDP; otherwise, it will pass through CGP. We then calculate the loss and update the corresponding LoRA adapters based on the source of the data sample.

Remarks. By orchestrating the independent and collaborative tuning paths, MPLP dismantles the information barrier between TDSLM and CGSLM, fostering their consensus during inference and thereby alleviating the misalignment issue.



User Target Extraction Prompt

System Message
You are a skilled IoT application developer, especially in writing well-structured task specifications. Given the task description, define and directly output the objective of the task. Example: develop a R-peak detection algorithm for ECG data.

User Message (t_i)
Task 1 [Task Name]
- Implementation step 1
- Implementation step 2 ......

(a) User target extraction

Input/Output Specification Extraction Prompt

System Message
You are a skilled IoT application developer, especially in writing well-structured task specifications. Given the task description, define the input/output specifications for the expected code. The output must be in the following format:
- Param 1 (type): description ......

User Message (t_i)
Task 1 [Task Name]
- Implementation step 1 ......

(b) I/O specification extraction

Figure 12: CoT-based prompts for RTSLM.

4.2.3 *Projection Layers.*

To further enhance knowledge sharing between the two SLMs during tuning, we create two projection layers for the two paths. We place the projection layers in parallel with the FFN layers in the Transformer block. As such, they can serve as extra FFNs that apply non-linear transformations to the attention representations, thereby enhancing token-level feature extraction and increasing the complexity of the model's learning capabilities. By receiving representations from the other path with enhanced non-linearity, the cross-domain IoT knowledge comprehension capabilities of the two SLMs will be further strengthened. Specifically, take TDP as an example, as shown in the upper part of Fig. 11, with the obtained value (denoted as x_1) from the LayerNorm in TDP, we feed it into a projection layer $L_1(\cdot)$. The output is then added with the FFN's output $F(x_2)$ and x_2 in CGP. The sum will be sent to the next Transformer block for further processing. Such a knowledge transfer process can be expressed as:

$$\begin{aligned}x_1' &= x_1 + F(x_1) + \gamma \cdot L_2(x_2)\\ x_2' &= x_2 + F(x_2) + (1-\gamma) \cdot L_1(x_1)\end{aligned} \tag{10}$$

where x_1' and x_2' are the final output of the two paths. x_1 and x_2 are the input attention representations from TDP and CGP, respectively. $F(\cdot)$ represents the FFN layer, $L_1(\cdot)$ and $L_2(\cdot)$ are the projection layers in the two paths, and γ is a hyper-parameter to balance the knowledge-sharing between the two paths. Note that each projection layer has the same architecture as the FFN, consisting of two fully connected layers and a non-linear SwiGLU function [54].

Remarks. By combining independent and collaborative tuning of LoRA adapters with the projection layers, PECT optimizes the task-specific performance of TDSLM and CGSLM while minimizing domain conflicts. As a result, the decomposed tasks generated by

TDSLM will have closer semantic alignment with CGSLM and thus can be better handled. Note that during both tuning and inference, the SLMs share the same foundation model architecture, with the only difference being the LoRA parameters shown in Fig. 11. Therefore, our proposed PECT paradigm fundamentally differs from those traditional PEFT approaches that tune SLMs separately.

4.3 Requirement Transformation

When cascading TDSLM and CGSLM together, a huge gap exists between TDSLM's outputs (*decomposed task descriptions*) and CGSLM's inputs (*task specifications*). The task descriptions are typically natural language while the task specifications are well-structured. Directly feeding the task descriptions into CGSLM will lead to sub-optimal performance of the generated code. To fill this gap, we leverage RTSLM to transform the descriptions into well-structured specifications. Considering that RTSLM has limited IoT knowledge comprehension ability, we enhance it with RAG and several CoT-based prompts to perform requirement transformation.

RAG Construction. To enhance RTSLM's ability in understanding IoT domain knowledge during requirement transformation, we first transform all the downloaded papers into a text embedding database. Then, armed with such an IoT knowledge database, we build a RAG agent based on RTSLM to retrieve relevant context for reference. As such, RTSLM can better comprehend and handle IoT terminologies in the task descriptions during requirement transformation.

CoT Prompting. Fig. 10(c) shows an example of a well-structured task specification for code generation, consisting of three parts: task target, input and output specifications of the expected code. For each decomposed task t_i generated by TDSLM, we prompt the agent to generate such well-structured specifications step-by-step. Specifically, we first prompt (Fig. 12(a)) the agent to summarize a target for the task. Next, we further instruct (Fig. 12(b)) the agent to generate a list of parameter descriptions for the input and output of the expected code. Each single parameter description item contains the parameter name, the parameter type, and a brief explanation of its meaning. For example, "signal (numpy.ndarray): the raw ECG data collected from patients with noises." Finally, RTSLM reorganizes and formats the above information into a well-structured task specification, which will be further handled by CGSLM to generate corresponding code snippets and documentation.

Remarks. Note that tuning is excluded in this process since it only needs basic language comprehension and processing capabilities of RTSLM. Therefore, RTSLM shares the same base model without additional tunable LoRA parameters to perform the transformation.

5 EXPERIMENT SETUP

5.1 Implementation

System Configurations. We deploy *GPIoT* on an edge PC equipped with an RTX 4090 GPU (24 GB). We use selenium [45] to create a web crawler for data retrieving from public websites. To perform data formatting and augmentation, we construct an agent based on GPT-4o and LangChain [8]. For SLM tuning, we use a high-performance cloud server with an NVIDIA A100 GPU (80 GB).

Hyper-parameters. TDD contains 36,098 pairs of "*problem statement → decomposed tasks*". CGD contains 35,419 pairs of "*task specification → code & documentation*". Llama2-13b [61] with INT8 quantization serves as the foundation model and is fine-tuned via LoRA [27], with a rank of 64 and a dropout rate of 0.1. The number of tuning epochs is 5, with an initial learning rate of 0.0001, varied by a cosine learning rate scheduler. The λ in Eq. 9 and the γ in Eq. 10 are both set to 0.5 by default. The tuning process takes around 80 GPU hours. Since TDSLM, RTSLM, and CGSLM share the same foundation model, only about 16 GB of GPU memory is needed for the whole system, which is affordable for a commodity GPU [52].

5.2 IoT Applications

Considering the different technologies required during development, we select three IoT applications, focusing on healthcare and edge computing. 1) **Heartbeat Detection (HD)** is essential for continuously monitoring patient vitals with enhanced healthcare and ensuring timely intervention in case of abnormalities [47]. We instruct *GPIoT* to develop a heartbeat (R-peak) detection algorithm and test it on the MIT-BIH dataset [44]. 2) **Human Activity Recognition (HAR)** [3, 7, 31–33] deployed on edge devices is important for real-time analysis of daily human activities. We instruct *GPIoT* to develop a WiFi-based HAR model using the WiAR dataset [23] and deploy it on a Jetson Nano board that has limited resources [40]. 3) **Multimodal HAR** leverages different sensors to capture complementary information, thereby enhancing HAR systems' robustness and versatility [19]. We instruct *GPIoT* to construct a multimodal HAR model based on the Harmony dataset [49], which contains three sensor modalities: audio [68], depth camera, and radar [13].

Notes: HD requires signal processing methods, HAR demands technologies in both signal processing and machine learning, and multimodal HAR necessitates advanced multimodal processing algorithms. Though we use HD as an example for demonstration throughout the paper, all the tasks are unseen to *GPIoT*.

6 EVALUATION

6.1 Metrics

We compare the programs synthesized by *GPIoT* and several baselines by measuring the following evaluation metrics.

HD. 1) *Precision*: The fraction of correctly detected R-peaks out of all detected peaks: $\frac{TP}{TP+FP}$. 2) *Recall rate*: The proportion of correctly detected R-peaks out of all actual R-peaks: $\frac{TP}{TP+FN}$. The larger these two metrics are, the more accurate the heartbeat detection becomes.

HAR. 1) *Classification accuracy*: The portion of the test data that is correctly classified based on the label. A higher accuracy implies a more robust and accurate HAR model. 2) *GPU memory usage*: The amount of GPU memory used during model inference. 3) *Inference time*: The time it takes from feeding the data into the code to the generation of the recognition result. The less memory and inference time consumed, the more resource-efficient the HAR model is.

6.2 Baselines

Given the same user problem, we input it into the following baselines to compare their performance with ***GPIoT* (GT)**. 1) **GPT-4o (G4)** [2] is an advanced LLM from OpenAI, optimized for instruction following and code generation tasks. 2) **DeepSeek-Coder (DC)** [22] is a high-performance code LLM, particularly effective in understanding and generating programming code across various domains. 3) **CodeLlama-34b (CL)** [53] is a specialized version of the Llama designed to generate, understand, and assist with coding. 4) **WizardCoder-33b (WC)** [43] incorporates complex instruction fine-tuning by adopting evolving instructions. 5) **CodeQwen-7b**

Heartbeat Detection (User Input)

Given the 'MIT-BIH' Arrhythmia Dataset, write Python code to perform R-peak detection for all the signal records in the dataset. The code should also output the detection accuracy for each data.

Code Input
1. The local file path to the dataset
Code Output Format
Case {ECG data record name}
Detection accuracy: 91% ……

Remarks
The dataset folder contains several data samples, each of which contains four files, *i.e.*, '.atr', '.dat', '.hea', and '.xws'.

(a) Problem statement for HD

Human Activity Recognition (User Input)

Given the WiFi-based Activity Recognition (WiAR) dataset, write some Python code to perform human activity recognition. First split the dataset into 'train' and 'test' parts. Then, train an AI model to output the recognition accuracy of the test data.

Code Input
1. The local file path to the dataset
Code Output Format
The average recognition accuracy: 91%

Remarks
The dataset is a NumPy array with a shape of (450, 90, 250), containing 15 activities with each repeated 30 times.

(b) Problem statement for HAR

Figure 13: Problem statements of the two applications (HAR and multimodal HAR share a similar prompt).

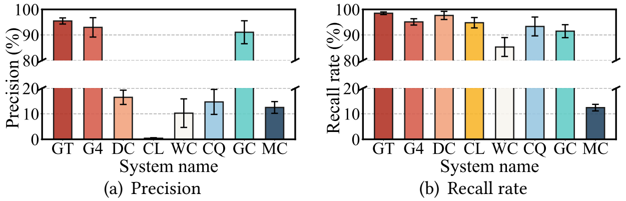
(CQ) [5] is the Code-Specific version of Qwen1.5, which is a decoder-only LLM pre-trained on a large amount of data of programs. 6) **GitHub Copilot (GC)** [21] is an AI-powered code generation tool that assists developers by suggesting code snippets and functions. 7) **MapCoder (MC)** [30] is an LLM+RAG-based code generation framework that cascades multiple LLM-based agents to solve competitive problems, where GPT-4o is selected as the built-in LLM.

We access GPT-4o and DeepSeek-Coder via API keys, interact with GitHub Copilot via Visual Studio Code's chat window, and deploy the rest on an edge server. Note that to our best, currently there is no LLM-based program synthesis system tailored for IoT application development. Therefore, we choose some SOTA code generation LLMs and systems as our baselines.

6.3 Application Evaluation

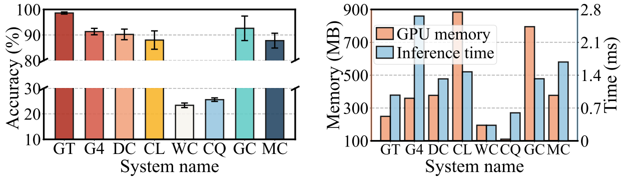
With the designed two problem statements (Fig. 13) for the three IoT applications, we input them into *GPIoT* and the baselines to synthesize 20 different programs for each task. We then evaluate their performance based on the metrics described in § 6.1.

6.3.1 ***HD****.* As shown in Fig. 14, the code generated by *GPIoT* significantly outperforms all the baselines, with an average precision gain of 64.7% and an average RR increase of 16.9%. It's also worth noting that CodeLlama, WizardCoder, and CodeQwen achieve moderate RR (above 80%) but exhibit lower precision. With further analysis of the code, we find that they all adopt a simple peak detection function, `scipy.signal.find_peaks()`, which typically fails when handling abnormal ECG data from patients. As a result, the detection results contain numerous false positives with low precision. Additionally, after reviewing the code generated by MapCoder, we observe that it incorporates more advanced heartbeat signal processing algorithms (*e.g.*, bandpass filtering and adaptative thresholding). However, the final program exhibits a significant performance drop compared with other baselines. This is because heartbeat detection is a relatively simple IoT application that does not require highly sophisticated planning and iterative debugging. Integrating many advanced algorithms into a simple signal-processing program may lead to inconsistency issues. In other words, the heartbeat signal may be over-processed by these algorithms, leading to degraded performance. [24, 60]. On the contrary, the code generated by *GPIoT* utilizes dedicated algorithms (*e.g.*, Pan-Tompkins) for R-peak detection due to embedded IoT domain knowledge during tuning, consistently achieving high precision and RR.



(a) Precision (b) Recall rate

Figure 14: The overall performance of HD.

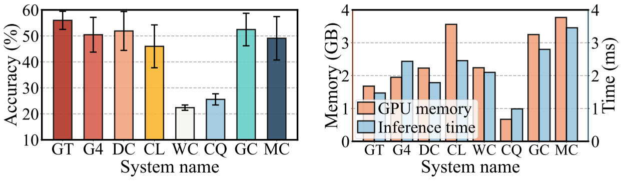


(a) Classification accuracy (b) GPU memory & inference time

Figure 15: The overall performance of HAR.

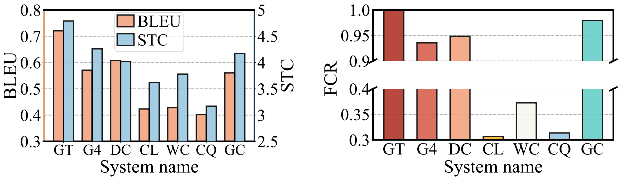
6.3.2 ***HAR****.* In this evaluation, for all the generated HAR models, we set the training epochs to 10 and the batch size to 32 for a fair comparison. Besides, during our implementation, we find that after around 15 training epochs, all the models gradually converge. Therefore, we compare the model performance at the 10th epoch. As shown in Fig. 15(a), the program synthesized by *GPIoT* achieves a 17.2% higher accuracy with 47.8% less GPU memory and 38.3% shorter inference time on average. By analyzing the generated code, we find *GPIoT* applies: 1) a data preprocessing method, Butterworth low-pass filtering, on the WiFi data, considering that the low-frequency components of WiFi CSI are primarily influenced by human activities [69]. 2) an augmentation method tailored for IoT data (*e.g.*, time-frequency masking [87]) on the WiFi signal to further enhance the diversity of the dataset. 3) a CUDA optimization mechanism [12] to reduce GPU memory usage while enhancing runtime efficiency. In contrast, the baselines directly input raw WiFi data into HAR models without GPU optimization, leading to diminished performance and heightened memory consumption. Moreover, the error bar of *GPIoT* is smaller than that of the baselines, indicating that *GPIoT* generates more stable responses with a more robust performance of the synthesized program. Note that MC can synthesize programs with competitive performance since HAR is a more complex application. Such advancements of *GPIoT* originate from our meticulously crafted IoT-specialized datasets and the PECT paradigm, which embeds abundant IoT domain knowledge from our datasets into the tuned SLMs for more consistent outputs.

Additionally, considering that our HAR model is deployed on edge devices with various resource constraints [39, 55, 57, 66, 67], we involve different resource requirements in the prompts for the generated code to facilitate a more comprehensive evaluation. Specifically, we first design a base prompt: "I need to deploy the HAR model on Jetson Nano" (**P1**) without any resource specifications. Based on **P1**, we then create three variations by adding different resource constraints: "Do not consider any resource constraints but only model accuracy" (**P2**), "The GPU memory usage should not exceed 200 MB" (**P3**), and "The GPU memory should not exceed 50 MB" (**P4**). We then instruct *GPIoT* to synthesize 20 different versions of programs using each of the prompts above. After executing the programs with the same configurations, we record the average classification accuracy and GPU memory usage, as shown in Fig 19(a). We find that: 1) Given P2, *GPIoT* can construct an AI model with a large number of parameters to achieve ultimate performance, with its classification accuracy approaching nearly 100%. 2) Given P3, *GPIoT* can adopt a smaller model within



(a) Classification accuracy (b) GPU memory & inference time

Figure 16: The overall performance of multimodal HAR



(a) The BLEU score (b) The Format Correctness Rate

Figure 17: Breakdown evaluation on TDSLM.

the resource budget (*i.e.*, GPU memory usage not exceeding 200 MB) with a slight performance drop. 3) Given P4, *GPIoT* employs a highly optimized model requiring only approximately 50 MB of GPU memory, resulting in an acceptable accuracy drop of 5%. These results demonstrate that *GPIoT* can generate code tailored to different resource budgets, stemming from our *IoT-Oriented Data Augmentation* method, which augments data samples considering resource heterogeneity of target devices in IoT applications.

6.3.3 *Multimodal HAR.*

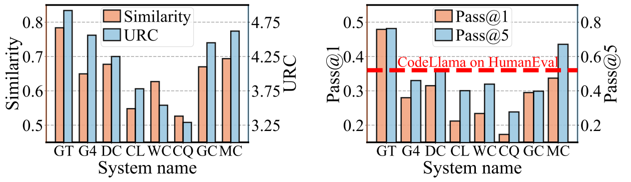
We further instruct *GPIoT* to synthesize programs for the multimodal HAR application, aiming to evaluate its programming ability for more complex tasks. As shown in Fig. 16, compared with the baselines, the program synthesized by *GPIoT* achieves an average accuracy improvement of 13.44% while requiring moderate GPU memory and inference time. After reviewing the source code, we find that both *GPIoT* and the baselines train three encoders to first extract useful features from different modalities, followed by a classifier to recognize the corresponding activity. However, the program synthesized by *GPIoT* adopts some model optimization methods (*e.g.*, quantization or pruning) and data augmentation methods tailored for IoT sensor data (*e.g.*, time-frequency masking). As such, the synthesized program can train a memory-optimized model while maintaining high classification accuracy. These results indicate that, benefiting from our SLM tuning, the program synthesized by *GPIoT* can incorporate more IoT-specific data processing and model optimization algorithms, thereby achieving high performance even for multimodal HAR.

6.4 Breakdown Evaluation

We separately evaluate TDSLM and CGSLM on ***IoTBench*** to explore the effectiveness of fine-tuning in the IoT domain.

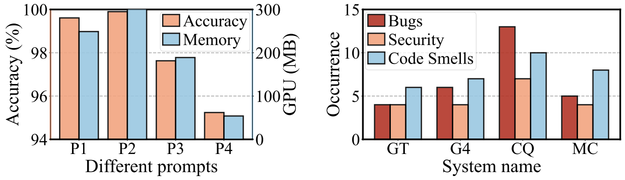
6.4.1 *Metrics.*

We adopt different metrics for the two SLMs. **TDSLM.** 1) *BLEU*: we measure the BLEU score [51] between the generated decomposed tasks and the reference from *IoTBench*. A larger BLEU score indicates higher semantic similarity and higher task decomposition quality. 2) *Format Correctness Rate* (FCR): the portion of TDSLM's outputs that correctly separate each decomposed task with a blank line for the convenience of further processing. This aims to quantify TDSLM's instruction-following and text-formatting abilities. 3) *Sub-Task Completeness* (STC): we invite 10 IoT experts to assess the extent to which the decomposed tasks cover all essential parts of the application based on the reference.



(a) The embedding similarity (b) The pass@k

Figure 18: Breakdown evaluation on CGSLM



(a) Specifying resource constraints (b) Code quality evaluation

Figure 19: (a) Evaluating *GPIoT*'s performance using prompts with different resource constraints. (b) Evaluating the quality of the code generated by CGSLM using SonarQube.

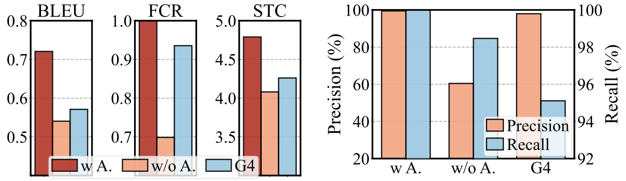
CGSLM. 1) *Code embedding similarity*: we use CodeT5+ [71] to convert code snippets into embeddings and compute the cosine similarity between embeddings of the generated and reference code. A higher similarity indicates a stronger ability to generate IoT-related code. 2) *Pass@k*: we measure the pass@k value by calculating the portion of programs that pass all the test cases. A higher value indicates better performance of the generated code. 3) *User Requirement Coverage* (URC): we first ask the users to execute the generated code and review the documentation. Next, they are asked to evaluate the extent to which the generated code and documentation fulfill all the user requirements. 4) *Code quality*: we assess the quality of the generated code by adopting a commercial-off-the-shelf (COTS) code quality verification tool, SonarQube [59], detecting bug/logic errors, security issues, and code smells [1]. Code smells are not bugs but bad coding styles (*e.g.*, variable name mismatching regular expression) or potential weaknesses (*e.g.*, package version incompatibility). Note that STC and URC are user-related metrics, which are rated on a scale from 1 (not at all) to 5 (completely).

6.4.2 *TDSLM.*

We input each *problem statement* from *IoTBench* into TDSLM and the baselines to generate 20 different *decomposed tasks* and calculate the average BLEU score, FCR, and URC. From Fig. 17, we observe: 1) The decomposed tasks generated by TDSLM achieve a 48% higher BLEU score than the baselines on average, indicating a stronger decomposition ability for IoT tasks. 2) TDSLM achieves 99% FCR, indicating remarkable stability to generate intermediate output (*decomposed tasks*) based on pre-defined formats. 3) TDSLM also achieves a 28% higher STC on average, showcasing strong abilities in understanding IoT knowledge and generating comprehensive decomposed tasks for IoT applications. Such superior IoT task decomposition and data formatting performance of TDSLM originate from the tuning process on TDD with our IoT-oriented text data augmentation method.

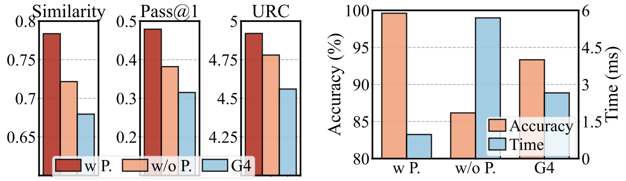
6.4.3 *CGSLM.*

We input each *task specification* from *IoTBench* into CGSLM and the baselines to generate 20 different *code & documentation*. We then report the average code embedding similarity, pass@1, pass@5, URC, and the number of various issues detected by SonarQube. by executing and reviewing the code. For comparison, we also report the pass@1 achieved by CodeLlama on a general-purpose programming benchmark, HumanEval [10]. From Fig. 18,



(a) Impact on TDSLM (b) Impact on HD

Figure 20: Ablation of IoT-oriented augmentation (A.).



(a) Impact on CGSLM (b) Impact on HAR

Figure 21: Ablation of PECT (P. in the figure).

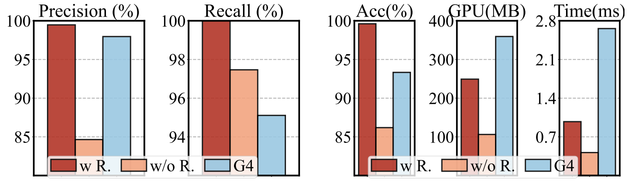
we observe that: 1) 1) CGSLM achieves an 18% higher code embedding similarity than the baselines on average, implying stronger capability and generalizability in generating IoT-related code snippets. 2) CGSLM achieves higher pass@1 and pass@5 than the baselines, with an average increase of 21.5% and 31%, respectively, showcasing higher quality and accuracy of the generated code for solving IoT tasks. 3) Nearly all the baselines achieve much lower pass@1 for IoT programming tasks than CodeLlama on HumanEval due to limited capabilities in the IoT domain. 4) Users show a stronger preference for CGSLM over the baselines, with an average increase of 23% in URC. 5) The program synthesized by CGSLM contains fewer bugs, security issues, and code-smell-related issues. This is primarily because, CGSLM, fine-tuned on our manually crafted datasets, can synthesize programs with enhanced code quality. Such superior abilities of CGSLM in generating IoT-related code snippets with high URC stems from our co-tuning on CGD with well-structured data. Consequently, CGSLM can generate code adopting more IoT-specific algorithms following task specifications.

6.5 Ablation Study

We conduct an ablation study by removing some proposed technical modules to investigate their importance to *GPIoT*.

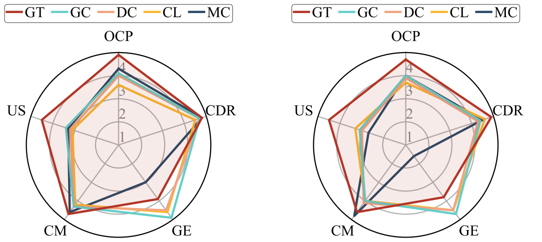
6.5.1 ***IoT-Oriented Data Augmentation.*** We directly tune TDSLM on the raw dataset without our IoT-oriented data augmentation, which contains only 273 data samples. We then evaluate the tuned model on *IoTBench* and report the average BLUE score, FCR, and STC. As shown in Fig. 20(a), without our augmentation, the tuned model exhibits a substantial performance decline across all metrics, much lower than GPT-4o. The main reason is that the raw dataset lacks generalizability and diversity in the IoT domain, which limits the tuned SLM's ability to decompose IoT problems into manageable sub-tasks, occasionally leading to incorrect results due to hallucinations. As a result, if we use such an insufficiently tuned TDSLM for heartbeat detection, *GPIoT* still adopts a simple peak detection algorithm, leading to significant performance degradation of the generated code in both precision and recall rate (Fig. 20(b)). The results demonstrate the importance of our IoT-oriented text data augmentation method in improving TDSLM's capability in task decomposition and IoT domain knowledge comprehension.

6.5.2 ***PECT.*** We separately fine-tune TDSLM and CGSLM on their own datasets without our PECT paradigm. We then evaluate the performance of the tuned CGSLM on *IoTBench* and measure the average code embedding similarity, pass@1, and URC. As shown in Fig. 21(a), without PECT, the code generated by CGSLM exhibits performance degradation across all the metrics. This is because some IoT domain knowledge possessed by TDSLM cannot be shared with CGSLM. As a result, CGSLM cannot handle some programming tasks that are out of the scope, providing simple programs with degraded performance. However, even such insufficiently tuned CGSLM still outperforms GPT-4o, highlighting the advantage of fine-tuning in enhancing IoT-related code generation ability. Furthermore, without PECT in the HAR task, the final code neither adopts data pre-processing methods nor designs high-performance neural networks, leading to decreased classification accuracy, as shown in Fig. 21(b). The results confirm the importance of our PECT paradigm in mitigating the domain misalignment issue and facilitating the IoT knowledge sharing between TDSLM and CGSLM.



(a) Impact on HD (b) Impact on HAR

Figure 22: Ablation of requirement transformation.



(a) Signal processing-related tasks (b) Machine learning-related tasks

Figure 23: User study on different tasks.

6.5.3 ***RTSLM.*** We directly feed the natural-language-described decomposed tasks from TDSLM into CGSLM. We then compare the performance of the generated code by *GPIoT*, *GPIoT* without RTSLM, and GPT-4o. As illustrated in Fig. 22, without RTSLM, we find that: 1) In HD, *GPIoT* tends to generate code unrelated to the IoT domain with substantial precision degradation but a high recall rate. This implies that the heartbeat detection results contain numerous false positives. 2) In HAR, *GPIoT* designs a basic HAR model with only a few simple layers, leading to a notable accuracy drop. These results highlight the importance of RTSLM in aiding CGSLM to understand the decomposed tasks generated by TDSLM, thereby improving its code generation ability for IoT applications.

6.6 User Study

We conduct a user study to evaluate the functionality, generalizability, and overall satisfaction of *GPIoT* for IoT application development. Specifically, with *GPIoT* deployed on an edge server, we invite 5 experts and 15 non-experts in IoT and ask them to freely express their requirements for any IoT application development that requires signal processing or AI technologies. By sequentially executing the generated code based on the instructions in the documentation, we ask the users to rate *GPIoT* based on five metrics: 1) *Overall Code Performance* **(OCP)** evaluates the overall performance of the generated code on corresponding test data considering task accuracy, runtime efficiency, and resource consumption; 2) *Code & Documentation Readability* **(CDR)** measures the clarity and structure of the code and documentation; 3) *Generation Efficiency* **(GE)** accesses how efficiently *GPIoT* operates in terms of speed and resource usage to produce the final results; 4) *Code Modularity* **(CM)** judges whether the code is properly modularized for easy reuse and extension; 5) *User Satisfaction* **(US)** captures users' feedback regarding their overall personal experience. All the above metrics are rated by the users on a scale from 1 (not at all) to 5 (completely). Github Copilot, DeepSeek-Coder (cloud), CodeLlama (local), and MapCoder (agent) serve as representative baselines for comparison.

As shown in Fig. 23, we observe: 1) *GPIoT* significantly outperforms the baselines in terms of OCP and US. The main reason is that, tuned on our IoT-specialized datasets, *GPIoT* can generate code containing more dedicated algorithms with better performance. Therefore, the users provide a higher score for *GPIoT* regarding the overall code performance and user satisfaction. 2) *GPIoT* achieves similar scores to the baselines in terms of CDR and CM, because our datasets mainly focus on generating IoT-related code snippets. Therefore, the readability of the code and documentation are not explicitly enhanced via tuning. 3) *GPIoT* gets a lower GE score as it performs requirement transformation and code generation for each decomposed task. Nevertheless, we can enhance its efficiency by adopting various LLM inference and serving optimization methods [20]. Additionally, during our implementation, we find that MapCoder costs around $20 to synthesize a program for a single IoT application while *GPIoT* incurs no query costs. The inferior performance of MapCoder may be attributed to the fact that LLM+RAG-based agents typically incorporate multiple modules to generate intermediate results in a cascaded manner, making them susceptible to unstable networks. This yields longer generation time, prohibitive token costs, and degraded user experiences. The user study results demonstrate the superior performance of *GIoT* in synthesizing IoT-related programs in an end-to-end manner, with the ability to generalize to other unseen IoT applications.

7 DISCUSSION

System Cost of *GPIoT*. In *GPIoT*, though there are three SLMs (*i.e.*, TDSLM, RTSLM, and CGSLM) operating simultaneously in *GPIoT*, they share the same foundation model architecture and only differ in a small subset of tunable parameters (§ 4.2). Moreover, though LLM+RAG systems (*e.g.*, MapCoder) require less memory (no more than 1 GB), *GPIoT* does not need sophisticated prompt design and requires a shorter generation time for the final output (§ 6.6).

Applicability to Resource-Constrained IoT Devices. The main focus of *GPIoT* is generating domain-specific and high-quality code for IoT application development. Considering the resource heterogeneity of various IoT devices for data processing, our *IoT-Oriented Data Augmentation* method (§ 4.1) augments the original task decomposition dataset by considering various resource requirements of different target platforms for IoT application development. Further experiments (Fig. 19(a)) demonstrate the effectiveness of *GPIoT* in handling different resource requirements with optimized models.

Generalizability of *GPIoT*. Existing IoT applications can be categorized into four types based on the functionality they deliver: data collection, data transmission, data processing, and decision-making. Data collection, transmission, and decision-making have been extensively studied using fixed programs. For instance, manufacturers typically provide COTS sample code for sensor data collection [14]. In contrast, IoT data processing demands more complex algorithms due to fluctuating sensor data with noises and various resource constraints. Therefore, *GPIoT* focuses on IoT data processing tasks by offering end-to-end solutions with executable programs. We believe that the general workflow (*i.e.*, task decomposition → requirement transformation → code generation) of *GPIoT* can be applied to other programming tasks. In future work, we plan to comprehensively assess *GPIoT* in other complex programming tasks.

8 RELATED WORK

Code LLMs & Programming Copilot. Code LLMs have significantly impacted the field of code generation, with prominent examples such as CodeLlama [53] and DeepSeek-Coder [88]. Trained on vast datasets comprising diverse code repositories, these models are capable of synthesizing programs based on user requirements, effectively bridging the gap between natural language and code. One notable application is GitHub Copilot, an LLM-powered assistant that provides real-time code suggestions and auto-completion. Though powerful and promising, existing code LLMs and copilots are primarily designed for general-purpose programming, lacking customization to the IoT domain when tasked with IoT applications. *GPIoT* addresses this by tuning local SLMs on IoT-specialized text-generation datasets with scrupulous augmentation. Additionally, by locally deploying *GPIoT*, it can potentially serve as a copilot for IoT application developers, enhancing task accuracy and development efficiency in a privacy-preserving manner.

Data Augmentation for LLMs. Existing text augmentation methods [16, 72] for LLM tuning harness the advanced language processing capabilities of powerful LLMs (*e.g.*, GPT-4) to synthesize diverse and high-quality text data. These methods have two categories: 1) Depth-based augmentation [9, 78] aims to increase the complexity of the original text data by adding constraints, concretizing the problem, and increasing reasoning steps. 2) Breadth-based augmentation [37] directly uses powerful LLMs to rewrite the original text data and generate a completely new instruction. However, these augmentation methods focus on linguistic characteristics rather than the IoT domain knowledge. In *GPIoT*, we propose a novel IoT-oriented text augmentation method tailored for the IoT domain, considering unique features of IoT applications, *i.e.*, sensor modalities, data representations, and system resource constraints.

9 CONCLUSION

We present *GPIoT*, a tailored local code generation system that synthesizes programs with documentation based on user requirements for IoT application development. Armed with two IoT-specialized text-generation datasets, the IoT-oriented augmentation method, and our PECT paradigm, *GPIoT* can generate more IoT-related code in a privacy-preserving way, achieving enhanced task accuracy and user satisfaction for IoT application development. As IoT technologies are emerging rapidly, it is also worthwhile to explore the construction of a dynamic IoT knowledge database and continuous fine-tuning of local SLMs in the future.

ACKNOWLEDGMENTS

We sincerely thank our anonymous shepherd and reviewers for their constructive comments and invaluable suggestions that helped improve this paper. This work is supported by Hong Kong GRF Grant No. 15211924 and 15206123. Yuanqing Zheng is the corresponding author.

REFERENCES

[1] 2024. Python static code analysis. Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your PYTHON code. https://rules.sonarsource.com/python/RSPEC-2316/

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[3] Aakriti Adhikari and Sanjib Sur. 2024. MiSleep: Human sleep posture identification from deep learning augmented millimeter-wave wireless systems. *ACM Transactions on Internet of Things* (2024), 1–33.

[4] Tuo An, Yunjiao Zhou, Han Zou, and Jianfei Yang. 2024. IoT-LLM: Enhancing Real-World IoT Task Reasoning with Large Language Models. *arXiv preprint arXiv:2410.02429* (2024).

[5] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[6] Jiani Cao, Jiesong Chen, Chengdong Lin, Yang Liu, Kun Wang, and Zhenjiang Li. 2024. Practical Gaze Tracking on Any Surface with Your Phone. *IEEE Transactions on Mobile Computing* (2024).

[7] Jiani Cao, Yang Liu, Lixiang Han, and Zhenjiang Li. 2024. Finger Tracking Using Wrist-Worn EMG Sensors. *IEEE Transactions on Mobile Computing* (2024).

[8] Harrison Chase. 2022. *LangChain.* https://github.com/langchain-ai/langchain

[9] Liuqing Chen, Yiyan Tsang, Qianzhi Jing, and Lingyun Sun. 2024. A LLM-augmented Morphological Analysis Approach for Conceptual Design. (2024).

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[11] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595* (2023).

[12] Jake Choi, Heon Young Yeom, and Yoonhee Kim. 2021. Implementing cuda unified memory in the pytorch framework. In *IEEE ACSOS-C.*

[13] Kaiyan Cui, Leming Shen, Yuanqing Zheng, Fu Xiao, and Jinsong Han. 2024. Talk2Radar: Talking to mmWave Radars via Smartphone Speaker. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications.* IEEE, 2358–2367.

[14] Dejan. 2024. Arduino and MPU6050 Accelerometer and Gyroscope Tutorial. https://howtomechatronics.com/tutorials/arduino/arduino-and-mpu6050-accelerometer-and-gyroscope-tutorial/

[15] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2024. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems* (2024).

[16] Bosheng Ding, Chengwei Qin, Ruochen Zhao, Tianze Luo, Xinze Li, Guizhen Chen, Wenhan Xia, Junjie Hu, Anh Tuan Luu, and Shafiq Joty. 2024. Data augmentation using llms: Data perspectives, learning paradigms and challenges. *arXiv preprint arXiv:2403.02990* (2024).

[17] Hao Ding, Ziwei Fan, Ingo Guehring, Gaurav Gupta, Wooseok Ha, Jun Huan, Linbo Liu, Behrooz Omidvar-Tehrani, Shiqi Wang, and Hao Zhou. 2024. Reasoning and Planning with Large Language Models in Code Development. In *ACM KDD.*

[18] Ning Ding, Xingtai Lv, Qiaosen Wang, Yulin Chen, Bowen Zhou, Zhiyuan Liu, and Maosong Sun. 2023. Sparse low-rank adaptation of pre-trained language models. *arXiv preprint arXiv:2311.11696* (2023).

[19] Jinxiao Fan, Mengshi Qi, Liang Liu, and Huadong Ma. 2025. Diffusion-driven Incomplete Multimodal Learning for Air Quality Prediction. *ACM Transactions on Internet of Things* (2025), 1–24.

[20] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. {ServerlessLLM}:{Low-Latency} Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation.*

[21] GitHub. 2024. GitHub Copilot - The world's most widely adopted AI developer tool. https://github.com/features/copilot

[22] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[23] Linlin Guo, Lei Wang, Chuang Lin, Jialin Liu, Bingxian Lu, Jian Fang, Zhonghao Liu, Zeyang Shan, Jingwen Yang, and Silu Guo. 2019. Wiar: A public dataset for wifi-based activity recognition. *IEEE Access* (2019).

[24] Dong Han, Syed Khairul Bashar, Jesús Lázaro, Fahimeh Mohagheghian, Andrew Peitzsch, Nishat Nishita, Eric Ding, Emily L Dickson, Danielle DiMezza, Jessica Scott, et al. 2022. A real-time PPG peak detection method for accurate determination of heart rate during sinus rhythm and cardiac arrhythmia. *Biosensors* (2022), 82.

[25] Ningning Hou, Xianjin Xia, Yifeng Wang, and Yuanqing Zheng. 2024. One shot for all: Quick and accurate data aggregation for LPWANs. *IEEE/ACM Transactions on Networking* (2024), 2285–2298.

[26] Ningning Hou, Xianjin Xia, and Yuanqing Zheng. 2023. Don't miss weak packets: Boosting LoRa reception with antenna diversities. *ACM Transactions on Sensor Networks* (2023), 1–25.

[27] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[28] Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010* (2023).

[29] IBM. 2024. What are small language models? https://www.ibm.com/think/topics/small-language-models

[30] Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-Agent Code Generation for Competitive Problem Solving. *arXiv preprint arXiv:2405.11403* (2024).

[31] Sijie Ji, Yaxiong Xie, and Mo Li. 2022. SiFall: Practical online fall detection with RF sensing. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems.* 563–577.

[32] Sijie Ji, Xuanye Zhang, Yuanqing Zheng, and Mo Li. 2023. Construct 3d hand skeleton with commercial wifi. In *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems.* 322–334.

[33] Sijie Ji, Xinzhe Zheng, and Chenshu Wu. 2024. Hargpt: Are llms zero-shot human activity recognizers?. In *2024 IEEE International Workshop on Foundation Models for Cyber-Physical Systems & Internet of Things (FMSys).* IEEE, 38–43.

[34] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *arXiv preprint arXiv:2406.00515* (2024).

[35] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[36] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* (2020).

[37] Zheng Li, Lijia Si, Caili Guo, Yang Yang, and Qiushi Cao. 2024. Data Augmentation for Text-based Person Retrieval Using Large Language Models. *arXiv preprint arXiv:2405.11971* (2024).

[38] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient Serving of LLM-based Applications with Semantic Variable. In *USENIX OSDI.*

[39] Chengdong Lin, Kun Wang, Zhenjiang Li, and Yu Pu. 2023. A workload-aware dvfs robust to concurrent tasks for mobile devices. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking.* 1–16.

[40] Neiwen Ling, Kai Wang, Yuze He, Guoliang Xing, and Daqi Xie. 2021. Rt-mdl: Supporting real-time mixed deep learning tasks on edge platforms. In *ACM SenSys.*

[41] Kaiwei Liu, Bufang Yang, Lilin Xu, Yunqi Guo, Neiwen Ling, Zhihe Zhao, Guoliang Xing, Xian Shuai, Xiaozhe Ren, Xin Jiang, et al. 2024. Tasking Heterogeneous Sensor Systems with LLMs. In *ACM SenSys.*

[42] Zhenyan Lu, Xiang Li, Dongqi Cai, Rongjie Yi, Fangming Liu, Xiwen Zhang, Nicholas D Lane, and Mengwei Xu. 2024. Small Language Models: Survey, Measurements, and Insights. *arXiv preprint arXiv:2409.15790* (2024).

[43] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).

[44] George B Moody and Roger G Mark. 2001. The impact of the MIT-BIH arrhythmia database. *IEEE engineering in medicine and biology magazine* (2001).

[45] Baiju Muthukadan et al. [n. d.]. *Selenium with Python.* https://github.com/baijum/selenium-python

[46] Jingping Nie, Hanya Shao, Yuang Fan, Qijia Shao, Haoxuan You, Matthias Preindl, and Xiaofan Jiang. 2024. LLM-based Conversational AI Therapist for Daily Functioning Screening and Psychotherapeutic Intervention via Everyday Smart Devices. *arXiv preprint arXiv:2403.10779* (2024).

[47] Xiaomin Ouyang, Xian Shuai, Yang Li, Li Pan, Xifan Zhang, Heming Fu, Sitong Cheng, Xinyan Wang, Shihua Cao, Jiang Xin, et al. 2024. ADMarker: A Multi-Modal Federated Learning System for Monitoring Digital Biomarkers of Alzheimer's Disease. In *ACM MobiCom.*

[48] Xiaomin Ouyang and Mani Srivastava. 2024. LLMSense: Harnessing LLMs for High-level Reasoning Over Spatiotemporal Sensor Traces. *arXiv preprint arXiv:2403.19857* (2024).

[49] Xiaomin Ouyang, Zhiyuan Xie, Heming Fu, Sitong Cheng, Li Pan, Neiwen Ling, Guoliang Xing, Jiayu Zhou, and Jianwei Huang. 2023. Harmony: Heterogeneous multi-modal federated learning through disentangled model training. In *ACM MobiSys.* 530–543.

[50] Jiapu Pan and Willis J Tompkins. 1985. A real-time QRS detection algorithm. *IEEE transactions on biomedical engineering* (1985).

[51] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics.*

[52] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. 2020. Training large neural networks with constant memory using a new execution algorithm. *arXiv preprint arXiv:2002.05645* (2020).

[53] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[54] Noam Shazeer. 2020. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202* (2020).

[55] Leming Shen, Qiang Yang, Kaiyan Cui, Yuanqing Zheng, Xiao-Yong Wei, Jianwei Liu, and Jinsong Han. 2024. Fedconv: A learning-on-model paradigm for heterogeneous federated clients. In *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*. 398–411.

[56] Leming Shen, Qiang Yang, Yuanqing Zheng, and Mo Li. 2025. AutoIOT: LLM-Driven Automated Natural Language Programming for AIoT Applications. In *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*.

[57] Leming Shen and Yuanqing Zheng. 2023. FedDM: data and model heterogeneity-aware federated learning via dynamic weight sharing. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 975–976.

[58] Leming Shen and Yuanqing Zheng. 2024. IoTCoder: A Copilot for IoT Application Development. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 1647–1649.

[59] Sonar Source. 2024. SonarQube. https://www.sonarsource.com/

[60] Javier Tejedor, Constantino A García, David G Márquez, Rafael Raya, and Abraham Otero. 2019. Multiple physiological signals fusion techniques for improving heartbeat detection: A review. *Sensors* (2019), 4708.

[61] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[62] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. Improving LLM code generation with grammar augmentation. *arXiv preprint arXiv:2403.01632* (2024).

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* (2017).

[64] Pauli Virtanen et al. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* (2020).

[65] Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng, Jiachen Liu, Zhongnan Qu, Shen Yan, Yi Zhu, Quanlu Zhang, et al. 2023. Efficient large language models: A survey. *arXiv preprint arXiv:2312.03863* (2023).

[66] Kun Wang, Jiani Cao, Zimu Zhou, and Zhenjiang Li. 2024. SwapNet: Efficient Swapping for DNN Inference on Edge AI Devices Beyond the Memory Budget. *IEEE Transactions on Mobile Computing* (2024).

[67] Kun Wang, Zimu Zhou, and Zhenjiang Li. 2024. LATTE: Layer Algorithm-aware Training Time Estimation for Heterogeneous Federated Learning. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 1470–1484.

[68] Tianben Wang, Zhangben Li, Honghao Yan, Xiantao Liu, Boqin Liu, Shengjie Li, Zhongyu Ma, Jin Hu, Daqing Zhang, and Tao Gu. 2023. AudioGuard: Omnidirectional Indoor Intrusion Detection Using Audio Device. *ACM Transactions on Internet of Things* (2023), 1–22.

[69] Wei Wang, Alex X Liu, Muhammad Shahzad, Kang Ling, and Sanglu Lu. 2015. Understanding and modeling of wifi signal based human activity recognition. In *ACM MobiCom*.

[70] Xin Wang, Yu Zheng, Zhongwei Wan, and Mi Zhang. 2024. Svd-llm: Truncation-aware singular value decomposition for large language model compression. *arXiv preprint arXiv:2403.07378* (2024).

[71] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[72] Zhenhua Wang, Guang Xu, and Ming Ren. 2024. LLM-Generated Natural Language Meets Scaling Laws: New Explorations and Data Augmentation Methods. *arXiv preprint arXiv:2407.00322* (2024).

[73] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* (2022).

[74] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. Autodroid: Llm-powered task automation in android. In *ACM MobiCom*.

[75] Kevin Wu, Eric Wu, and James Zou. 2024. How faithful are RAG models? Quantifying the tug-of-war between RAG and LLMs' internal prior. *arXiv:2404.10198* (2024).

[76] Shengqiong Wu, Hao Fei, Leigang Qu, Wei Ji, and Tat-Seng Chua. 2023. Next-gpt: Any-to-any multimodal llm. *arXiv preprint arXiv:2309.05519* (2023).

[77] Guangzhi Xiong, Qiao Jin, Xiao Wang, Minjia Zhang, Zhiyong Lu, and Aidong Zhang. 2024. Improving Retrieval-Augmented Generation in Medicine with Iterative Follow-up Questions. *arXiv preprint arXiv:2408.00727* (2024).

[78] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244* (2023).

[79] Huatao Xu, Liying Han, Qirui Yang, Mo Li, and Mani Srivastava. 2024. Penetrative ai: Making llms comprehend the physical world. In *ACM HotMobile*.

[80] Qiang Yang and Yuanqing Zheng. 2021. Model-based head orientation estimation for smart devices. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* (2021), 1–24.

[81] Qiang Yang and Yuanqing Zheng. 2023. Aquahelper: Underwater sos transmission and detection in swimming pools. In *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems*. 294–307.

[82] Qiang Yang and Yuanqing Zheng. 2024. Neural Enhanced Underwater SOS Detection. In *IEEE INFOCOM*. 971–980.

[83] Shiming Yu, Xianjin Xia, Ningning Hou, Yuanqing Zheng, and Tao Gu. 2024. Revolutionizing lora gateway with xgate: Scalable concurrent transmission across massive logical channels. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 482–496.

[84] Shiming Yu, Xianjin Xia, Ziyue Zhang, Ningning Hou, and Yuanqing Zheng. 2024. FDLoRa: Tackling Downlink-Uplink Asymmetry with Full-duplex LoRa Gateways. In *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*. 281–294.

[85] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).

[86] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *ICML*. PMLR.

[87] Hao Zhou, Taiting Lu, Yilin Liu, Shijia Zhang, and Mahanth Gowda. 2022. Learning on the Rings: Self-Supervised 3D Finger Motion Tracking Using Wearable Sensors. *ACM IMWUT* (2022).

[88] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024).