AutoIOT: LLM-Driven Automated Natural Language Programming for AIoT Applications

Leming Shen[1], Qiang Yang[2], Yuanqing Zheng[1], Mo Li[3]

[1]The Hong Kong Polytechnic University, [2]University of Cambridge,
[3]Hong Kong University of Science and Technology

leming.shen@connect.polyu.hk,qy258@cam.ac.uk,csyqzheng@comp.polyu.edu.hk,lim@cse.ust.hk

ABSTRACT

The advent of Large Language Models (LLMs) has profoundly transformed our lives, revolutionizing interactions with AI and lowering the barrier to AI usage. While LLMs are primarily designed for natural language interaction, the extensive embedded knowledge empowers them to comprehend digital sensor data. This capability enables LLMs to engage with the physical world through IoT sensors and actuators, performing a myriad of AIoT tasks. Consequently, this evolution triggers a paradigm shift in conventional AIoT application development, democratizing its accessibility to all by facilitating the design and development of AIoT applications via natural language. However, some limitations need to be addressed to unlock the full potential of LLMs in AIoT application development. First, existing solutions often require transferring raw sensor data to LLM servers, which raises privacy concerns, incurs high query fees, and is limited by token size. Moreover, the reasoning processes of LLMs are opaque to users, making it difficult to verify the robustness and correctness of inference results. This paper introduces *AutoIOT*, an LLM-based automated program generator for AIoT applications. *AutoIOT* enables users to specify their requirements using natural language (input) and automatically synthesizes interpretable programs with documentation (output). *AutoIOT* automates the iterative optimization to enhance the quality of generated code with minimum user involvement. *AutoIOT* not only makes the execution of AIoT tasks more explainable but also mitigates privacy concerns and reduces token costs with local execution of synthesized programs. Extensive experiments and user studies demonstrate *AutoIOT*'s remarkable capability in program synthesis for various AIoT tasks. The synthesized programs can match and even outperform some representative baselines.

CCS CONCEPTS

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

KEYWORDS

Large Language Model, Penetrative AI, Program Synthesis

1 INTRODUCTION

Artificial Intelligence of Things (AIoT) [19, 41, 48, 51, 65, 77] is an emerging paradigm that leverages advanced artificial intelligence (AI) algorithms to process a vast amount of data generated by Internet of Things (IoT) devices. This technology brings a new level of intelligence and automation to various applications, including healthcare [57, 59], smart sensing [82–84], and autonomous driving [36].

Recent advances in large language models (LLMs) (*e.g.*, GPT-4 [16]) fundamentally changed the way we interact with AI. While initially designed to understand natural languages, recent pioneering works [43, 53, 79] have demonstrated considerable proficiency of LLMs in exploiting embedded world knowledge by interpreting IoT sensor data to perform various AIoT tasks. Recent works [79] term such an endeavor – Penetrative AI. Fig. 1 illustrates how LLMs can be tasked to comprehend and even interact with the physical world through integration with IoT sensors and actuators.

However, current LLMs on AIoT tasks [53, 61, 79] fall short in supporting AIoT applications [20, 38–40, 86, 87]: 1) The trustworthiness of the inference results is compromised since the inference process is performed inside a "black box" and opaque to users. Thus, the robustness of the applications or the correctness of the inference results are hard to
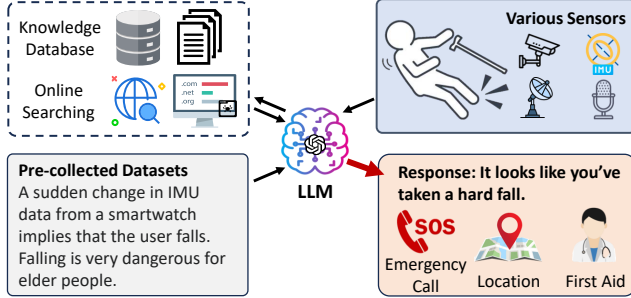
Figure 1: Illustration of how LLMs can sense and interact with the physical world in AIoT applications.

verify; 2) Transmitting the raw or intermediate sensor data from user devices to LLM servers raises privacy concerns, incurs prohibitive query fees, and increases response latency; 3) Sensor data typically exhibits extensive length and high dimensionality, making remote processing at LLM servers infeasible due to token limits [15, 85]. Ideally, the integration of LLMs with AIoT applications should be trustworthy, privacy-preserving, and communication-efficient.

On the other hand, existing works on LLMs have showcased their remarkable capabilities in code generation to accomplish a variety of programming tasks [44, 47, 54]. *Can we leverage LLMs to synthesize programs to fulfill AIoT application requirements?* This approach can 1) enhance the explainability and trustworthiness of the AIoT applications as the synthesized programs can be examined and interpreted by developers, 2) mitigate privacy concerns, and reduce the communication cost since the programs can be executed locally on user devices without offloading raw sensor data, and 3) efficiently process high-dimensional continuous sensor data without being limited by the token size or bounded by the round trip time over the network. To this end, we propose *AutoIOT*, a user-friendly natural language programming system based on LLMs. *AutoIOT* automatically identifies and retrieves the necessary domain knowledge over the Internet, intelligently synthesizes programs, and evolves the programs iteratively given sample inputs and ground truth. Surprisingly, we found that the synthesized programs can sometimes outperform some representative baselines and sample programs of recent academic papers.

While the automatic program synthesis for AIoT applications is promising and exciting, it entails tremendous technical challenges. 1) **High Complexity of AIoT Tasks.** Contrary to existing works that generate code for individual modules or well-defined functions [17], AIoT applications typically need a systematic design and integration involving multiple functional components, leading to much higher reasoning and planning complexity beyond the capability of current LLMs. To address this issue, *AutoIOT* decomposes the programming task into several distinct modules and generates the corresponding code segments. In particular, *AutoIOT* leverages chain-of-thought (CoT) prompts [76] to divide the

task into a few sub-tasks and integrate their solutions, eventually making the sub-tasks manageable by LLMs. 2) **Lack of Domain Knowledge in AIoT.** LLMs are trained on pre-collected general corpus datasets, which may not include the latest domain-specific knowledge needed for the development of various emerging AIoT applications. To tackle this problem, *AutoIOT* guides the LLMs to search and retrieve necessary knowledge and algorithms, thereby enabling in-context training and inference augmented with domain knowledge for LLMs. 3) **Heavy Intervention and Constant Feedback.** Our preliminary experiments (§ 2.2) reveal that to generate functionally correct programs, developers have to give timely feedback to LLMs and constantly intervene in the entire development process. For example, developers need to provide specific reference materials and describe algorithms in great detail, which can be time-consuming and defeat the very purpose of automated natural language programming. Ideally, *AutoIOT* should be able to synthesize the program with no intervention from users and require minimum user input only when necessary. To this end, we develop *AutoIOT* that can execute, debug, and optimize the synthesized program given sample inputs and outputs.

We fully implement *AutoIOT*[1] and evaluate its synthesized programs with four representative AIoT applications: heartbeat detection, IMU-based human activity recognition (HAR), mmWave-based HAR, and multimodal HAR. Extensive experiments and user studies show that, the synthesized programs can achieve comparable performance to the corresponding baselines and significant improvements in user satisfaction. Besides, *AutoIOT* substantially reduces the communication cost and the total execution time. These findings demonstrate the LLM's exceptional proficiency and great potential in synthesizing programs for AIoT applications. In summary, we make the following contributions:

- To our best, *AutoIOT* is the first work that enables systematic natural language programming for AIoT applications.
- We design and implement three novel technical modules (*i.e.*, background knowledge retrieval module, CoT prompting-based program synthesis module, and automated code improvement module) to synthesize and optimize programs for AIoT applications.
- Our comprehensive experiments demonstrate that synthesized programs can achieve comparable performance to baselines and sometimes outperform them.

2 BACKGROUND & MOTIVATION

We first revisit the pioneering Penetrative AI efforts that leverage the embedded knowledge of LLMs to address AIoT tasks. Then, we present the results of our preliminary experiments to demonstrate the feasibility and identify the
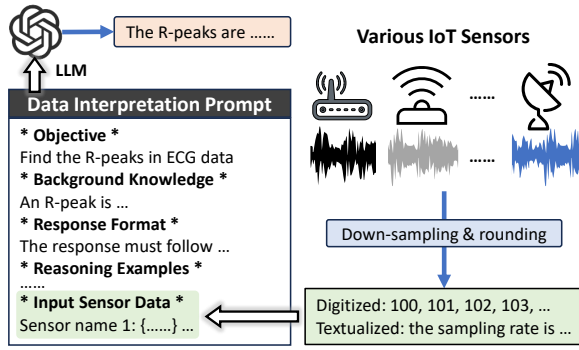
---

[1]The project is available at https://github.com/lemingshen/AutoIOT

Figure 2: Prior work processes sensor data with LLMs.

Figure 3: An example of direct code generation.

Figure 4: Code generation with user intervention.

key challenges of implementing an LLM-driven automated natural language programming system for AIoT applications.

2.1 Background

Fig. 2 illustrates an example of recent work in Penetrative AI [79], where various sensor data are textualized and embedded in a prompt, which is further used to instruct the LLM to perform inference tasks on the textualized sensor data.

Preliminary results indicate that the common knowledge embedded in LLMs can be leveraged to accomplish various real-world AIoT tasks (*e.g.*, inferring a user's location via WiFi fingerprints, and counting heartbeats via raw ECG data). However, the inference processes of LLMs are largely opaque to users, rendering the results less explainable and trustworthy. Moreover, raw sensor data is transmitted from users to LLM servers, raising privacy concerns about sensor data. Furthermore, limited by token size, existing work down-samples and quantizes raw sensor data, leading to degraded inference performance. The remote processing at LLM servers also necessitates the round-trip transmission of prompts and results over the network, which increases response latency.

To overcome these issues, we explore a new approach that leverages LLMs to synthesize AIoT programs and executes the programs locally to process users' data. This new approach allows developers to examine and verify the synthesized programs, protect data privacy, process sensor data

streams without compromising data resolution or quality, and avoid transmission time over networks.

2.2 Preliminary Experiments

The latest LLMs [16, 28, 71] have demonstrated extraordinary proficiency in generating code snippets. For example, Mercury [28] leverages LLMs to generate code for well-defined programming tasks. *Is it feasible to instruct LLMs to synthesize programs that can tackle AIoT tasks?* Our preliminary results show that it is possible yet extremely challenging for LLMs to synthesize functionally correct programs for AIoT tasks. Taking heartbeat detection as an example, as depicted in Fig. 3. When we instruct the LLM to generate a program to process raw ECG waveform and detect heartbeats, the LLM can only generate a few null functions without concrete implementation or import some nonexistent packages.

We hypothesize that the reasons for this might be threefold: 1) LLMs lack domain-specific knowledge, let alone the latest algorithms tailored for AIoT tasks. As a result, for highly specialized AIoT applications, LLMs can only offer some suggestions or generate high-level code outlines rather than detailed functional implementation. 2) AIoT applications typically require systematic programming, where multiple functional modules are first developed for different subtasks (*e.g.*, signal preprocessing, data cleaning, neural network initialization), which are then constructively integrated to form a comprehensive and cohesive program.
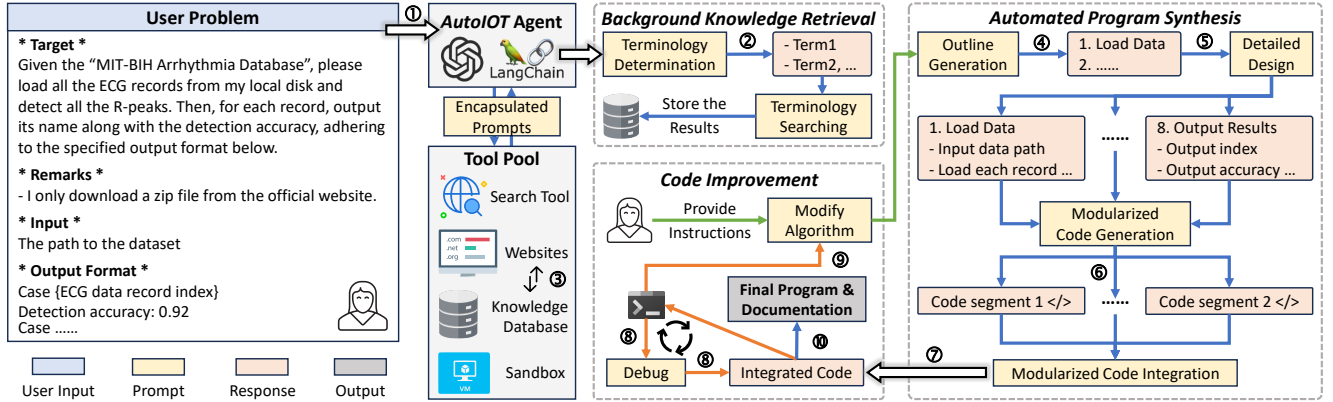
Leming Shen, Qiang Yang, Yuanqing Zheng, Mo Li

Figure 5: The system overview and workflow of *AutoIOT*.

This development process involves much higher reasoning, planning, and programming complexity than other simple well-defined programming tasks. 3) Current LLMs inherently lack code validation and optimization mechanisms to ensure the correctness of synthesized programs and improve the performance of programs in terms of execution efficiency and inference accuracy in AIoT program synthesis tasks.

To validate the above hypotheses, we conduct follow-up experiments 1) by providing necessary domain-specific knowledge to facilitate the design and implementation of corresponding algorithms to address the AIoT task and 2) by explicitly instructing the LLM to synthesize programs with clear structures via a divide-and-conquer approach. Fig. 4 illustrates the code generation process involving user intervention. We first manually retrieve relevant background knowledge (*e.g.*, definitions of ECG data and R-peak, research papers about heartbeat detection) from the Internet and feed the information to the LLM, enabling in-context learning. Second, we instruct the LLM to learn the relevant context and comprehend the papers. Then, we ask the LLM to generate an outline of the algorithm in the paper. We further request the LLM to enrich the outline with more advanced and detailed technologies. Later, we ask the LLM to generate code snippets corresponding to each step of the algorithm. The final program is thereafter synthesized by integrating all the code snippets. Finally, we fix bugs if there are any, and execute the program to evaluate its performance with test data. We further give feedback and ask the LLM to refine the program. With several rounds of iterations, the synthesized program evolves and improves its performance in the task.

In summary, although the synthesized programs eventually achieve reasonable performance in the tested AIoT tasks, this LLM-driven development method demands specialized domain expertise and constant manual intervention through several rounds of iterations for program optimization.

2.3 Motivation & Key Ideas

In this paper, we aim to develop an LLM-driven automated natural language programming system named *AutoIOT* to synthesize programs for AIoT applications. *AutoIOT* features three key modules: 1) *Background knowledge retrieval* module that automatically collects domain knowledge from the Internet for in-context learning; 2) *Automated program synthesis* module that emulates the program development lifecycle [42] via CoT prompting. This module decomposes an AIoT task into several subtasks and generates corresponding functional code snippets; and 3) *Code improvement* module that executes the synthesized program and feeds the compiler and interpreter feedback to the LLM, facilitating iterative code correction and improvement. We note that although the program synthesis process needs communication and interaction with remote LLM servers, the synthesized program can be executed locally on the client side. This approach fundamentally differs from existing approaches such as Penetrative AI [79], and allows users to not only preserve data privacy but also improve the interpretability of synthesized programs as well as inference results.

3 SYSTEM OVERVIEW

AutoIOT builds an intelligent agent that can automatically synthesize programs to fulfill user requirements in AIoT applications. As shown in Fig. 5, *AutoIOT* comprises three key modules: *background knowledge retrieval* (§ 4.2), *automated program synthesis* (§ 4.3), and *code improvement* (§ 4.4).

Users can specify their requirements on AIoT applications in natural language (①). Then, the *background knowledge retrieval* module identifies a set of relevant terminologies (②) and searches over the Internet (③). With the retrieved domain-specific knowledge, the *automated program synthesis* module instructs the agent to draft an algorithm outline (④). The agent is then requested to elaborate on each step of the algorithm and produce a detailed design (⑤). Such a process decomposes a complex AIoT task into several manageable subtasks. Then, the agent is instructed to generate a code segment for each subtask (⑥). Afterward, the agent is requested to integrate the codes for subtasks and synthesize the final program (⑦). Next, the *code self-improvement* module executes the synthesized program and feeds the compiler and
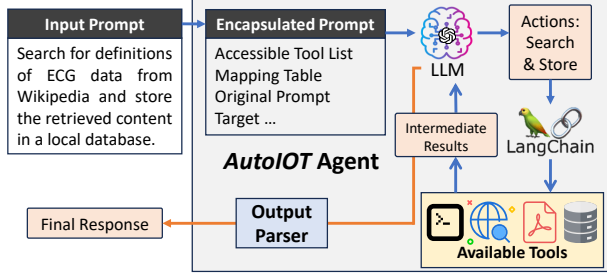
Figure 6: *AutoIOT* agent answers user's prompt with LLMs and LangChain tools.

interpreter output back to the agent. The agent iteratively corrects syntax and semantics errors (⑧). With the obtained output from the synthesized program, *AutoIOT* explicitly instructs the agent to explore more advanced algorithms using the web search tool, aiming to optimize the program and improve the performance of inference tasks (⑨). After several iterations (④-⑨), *AutoIOT* will present the final program with detailed documentation (⑩). In addition, *AutoIOT* provides an interface for users to offer specific algorithms or instructions for code improvement.

To enable the interaction between the LLM and the web search tool, the knowledge database, and the code executor, we leverage the LangChain [7] framework to build an intelligent agent. LangChain assembles various tools (abstracted as functions) and provides descriptions of the available functions (added into prompts) to the LLM. With such a prompt, the LLM performs reasoning and selects suitable tools to answer the user's query (potentially via multiple rounds of function invocations and message exchanges initiated by the LLM). This approach allows the LLM to answer queries that require context information (*e.g.*, local weather, user's local documents), augmenting the LLM with retrieved knowledge.

Taking step ③ in Fig. 5 as an example, Fig. 6 shows how *AutoIOT* agent works with LangChain tools and the LLM to answer the query about terminology searching via network. Given an input prompt, *AutoIOT* encapsulates it with additional information (*e.g.*, a list of available tools) and sends it to the LLM. Then, the LLM performs a sequence of actions possibly leveraging the available tools in the list, and generates an output. The output will then be passed to a parser to generate the final response. We note that the above processes involved in step ③ as well as other steps (①-⑩ in Fig. 5) are all automatically orchestrated by the *AutoIOT* agent.

Usage Scenario. Suppose a user wants to develop a heartbeat detection application, she can interact with *AutoIOT* with natural language, which describes her requirements for the application. Then, *AutoIOT* will automatically synthesize a corresponding program and documentation for the user. Following the instructions in the documentation, she can deploy and execute the program on a target device, which contains the patient's ECG data. The program will then generate the final heartbeat detection results for her.

4 SYSTEM DESIGN

In this section, we select heartbeat detection as an example to illustrate how *AutoIOT* works.

4.1 User Interface

Users can describe an AIoT task in natural language as input to *AutoIOT*. To help the LLM interpret the intention and desired outcome (*i.e.*, synthesized program and inference results), we design a prompt template for the user to describe the problem, since LLMs can comprehend and process well-structured instructions more efficiently [30]. As shown in Fig. 5, the user problem includes four parts: target, remarks, and program input and output specifications. The target part describes the user's objective and task in natural language, *e.g.*, "Given the MIT-BIH Arrhythmia Database, please load all the ECG records and detect all the R-peaks. Then, evaluate the detection results and output the detection accuracy for each record." The remarks provide additional information, *e.g.*, "I only downloaded a zip file from the dataset's official website". The program input and output specifications clarify the I/O format of the synthesized program. The path to the dataset is required during the *code improvement* process for program execution and optimization.

4.2 Background Knowledge Retrieval

LLMs are typically trained on extensive and pre-collected corpus datasets that include a wide range of general common knowledge over the Internet. These training datasets, however, may not include domain-specific knowledge or the latest advances in research literature. In the rapidly evolving AIoT field, with new technologies and algorithms constantly emerging, the knowledge gap is particularly pronounced. To fill this gap, we develop the *background knowledge retrieval* module to automatically identify and fetch necessary information online so as to enable in-context learning for LLMs augmented with up-to-date domain knowledge.

Terminology Determination. The *background knowledge retrieval* module first instructs the agent to identify some relevant key terminologies given the user problem, with the prompt shown in Fig. 7(a). For example, given the user problem in Fig. 5, the terminologies generated by the LLM are "MIT-BIH Arrhythmia Database", "ECG data", and "R-peaks". Next, to obtain the relevant knowledge, *AutoIOT* instructs (Fig. 7(b)) the LLM to actively search for the definitions and descriptions of these terminologies from public websites, such as Wikipedia and GitHub, with the web search tool. Additionally, for the terminologies with multiple interpretations, *AutoIOT* requests the LLM to filter out the irrelevant content and focus on those pertinent to the user problem.

Context Database Construction. After retrieving necessary information from relevant websites, *AutoIOT* uses OpenAI's text embedding [56] to convert the HTML documents into vector representations, containing semantic meanings

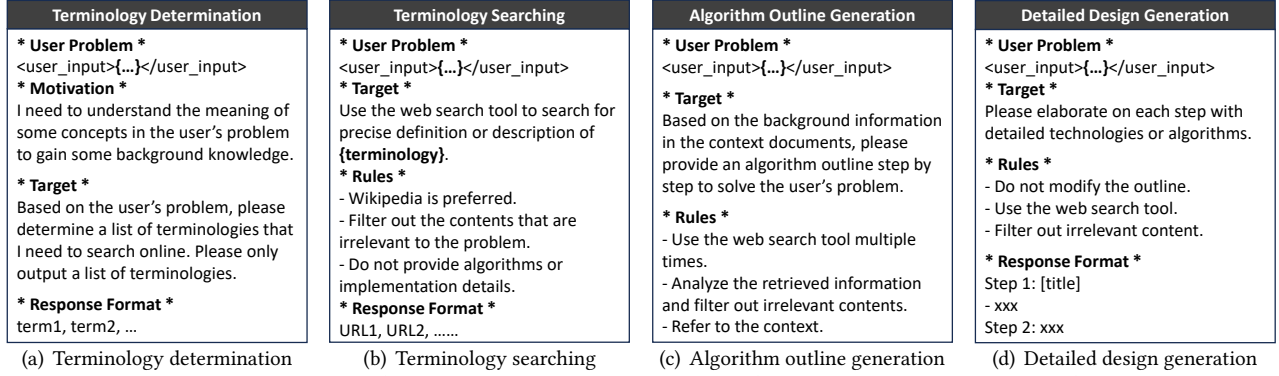| Terminology Determination	Terminology Searching	Algorithm Outline Generation	Detailed Design Generation
* User Problem * <user_input>{...}</user_input> *** Motivation *** I need to understand the meaning of some concepts in the user's problem to gain some background knowledge. *** Target *** Based on the user's problem, please determine a list of terminologies that I need to search online. Please only output a list of terminologies. *** Response Format *** term1, term2, ...	*** User Problem *** <user_input>{...}</user_input> *** Target *** Use the web search tool to search for precise definition or description of **{terminology}**. *** Rules *** - Wikipedia is preferred. - Filter out the contents that are irrelevant to the problem. - Do not provide algorithms or implementation details. *** Response Format *** URL1, URL2, ...	*** User Problem *** <user_input>{...}</user_input> *** Target *** Based on the background information in the context documents, please provide an algorithm outline step by step to solve the user's problem. *** Rules *** - Use the web search tool multiple times. - Analyze the retrieved information and filter out irrelevant contents. - Refer to the context.	*** User Problem *** <user_input>{...}</user_input> *** Target *** Please elaborate on each step with detailed technologies or algorithms. *** Rules *** - Do not modify the outline. - Use the web search tool. - Filter out irrelevant content. *** Response Format *** Step 1: [title] - xxx Step 2: xxx
(a) Terminology determination	(b) Terminology searching	(c) Algorithm outline generation	(d) Detailed design generation

Figure 7: The prompt template for (a) terminology determination, (b) terminology searching, (c) algorithm outline generation, and (d) detailed design generation.

comprehensible to the LLM. These representations are then used to build a local vector database with Faiss [27], serving as a contextual knowledge base for the LLM. During the inference process, the LLM retrieves relevant content from the database with a high degree of similarity to the user problem in the vector space. This approach ensures that the LLM understands the user problem and objective with necessary context information and domain knowledge.

Remarks. The *background knowledge retrieval* module is user-friendly, as it is operated automatically by our *AutoIOT* agent without any user intervention. In addition, we provide an interface for users to explicitly complement necessary background information as well as highly specialized domain knowledge (*e.g.*, research papers, detailed algorithm descriptions) to enrich the context database.

4.3 Automated Program Synthesis

As observed in our preliminary experiments (§ 2.2), if we directly instruct LLMs to generate programs for AIoT applications, multiple subtasks should be undertaken manually by the user. Specifically, the user needs to decompose the AIoT task into several subtasks and request the LLM to generate a solution for each subtask. After integrating the solutions, the user has to manually debug, execute, and improve the program. Although the synthesized program eventually meets the user's requirement, the program synthesis necessitates frequent user intervention and active involvement throughout the process, which is cumbersome and time-consuming. To address this problem, we develop the *automated program synthesis* module, aiming to automate the programming procedure, reduce the involved workload, and improve the development efficiency and user experience. In particular, the *automated program synthesis* module uses Chain-of-Thought (CoT) prompts to guide LLMs through step-by-step reasoning processes, thereby enhancing their capability of tackling complex AIoT problems by mimicking human-like divide-and-conquer reasoning processes.

CoT 1: Algorithm Outline Generation. *AutoIOT* first prompts the LLM to analyze the user problem and design a preliminary algorithm outline. As illustrated in Fig. 7(c), the

prompt for algorithm outline generation consists of three parts: 1) The "User Problem" is reiterated at the beginning to ensure continuity and coherence in the LLM's responses since the LLM may forget the previous context [75]. 2) The "Target" specifies our request, *i.e.*, algorithm outline generation. 3) The "Rules" add detailed instructions for quality assurance. For example, *AutoIOT* explicitly requests the LLM to actively search for advanced AIoT algorithms using the web search tool throughout the process. *AutoIOT* also asks the LLM to filter out irrelevant information. With such well-structured prompts, the LLM can generate an algorithm outline according to the problem specification.

CoT 2: Detailed Design Generation. Given the generated algorithm outline, the LLM is then tasked with further elaborating on each step in the outline with more detailed technologies and algorithms to refine the approach comprehensively. The prompt for this stage includes "User Problem", "Target", and "Rules". In addition, it also includes a new requirement - "Response Format" to specify the expected format of the LLM's output, as illustrated in Fig. 7(d). Given such a prompt, the LLM can generate detailed steps and specific actions in each step to achieve the overall objective and solve the user problem. In this stage, essentially *AutoIOT* guides the LLM to decompose the AIoT task into multiple subtasks, facilitating a divide-and-conquer strategy to synthesize the corresponding program in the next stage.

CoT 3: Modularized Code Generation. Given a set of subtasks generated at the previous stage, *AutoIOT* instructs the LLM to generate one function for each module with a clear function name, signature, and input/output specification. The modules can then be developed independently, ensuring specific functionalities are effectively implemented according to the algorithm descriptions generated above. This divide-and-conquer approach is proven effective in synthesizing complex programs by fully exploiting the modularized code generation capabilities [17] of LLMs.

In our initial trials, we observed that LLMs sometimes generate null functions with placeholders, invoke undefined functions, or import nonexistent packages. To tackle this

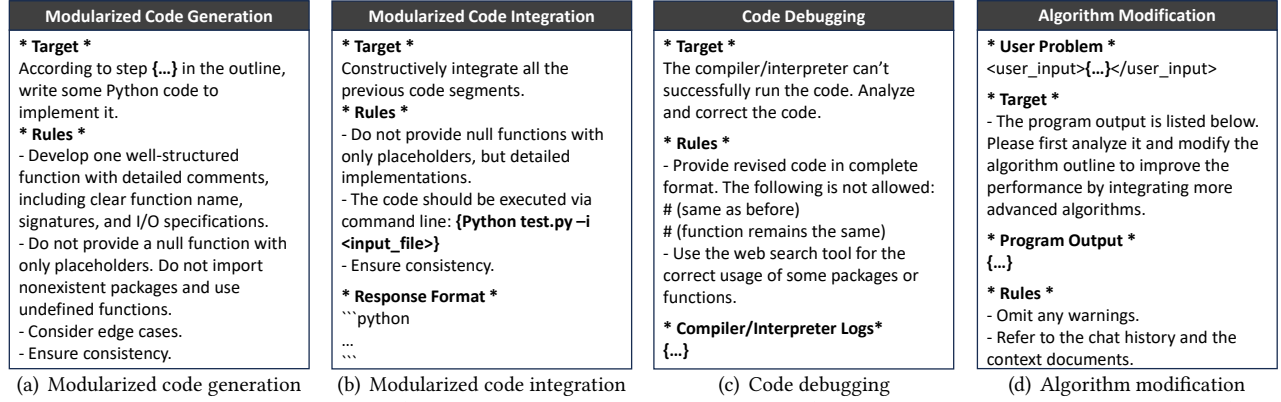| **Modularized Code Generation**	**Modularized Code Integration**	**Code Debugging**	**Algorithm Modification**

* Target *
According to step {...} in the outline, write some Python code to implement it.
* Rules *
- Develop one well-structured function with detailed comments, including clear function name, signatures, and I/O specifications.
- Do not provide a null function with only placeholders. Do not import nonexistent packages and use undefined functions.
- Consider edge cases.
- Ensure consistency.
	* Target *
	Constructively integrate all the previous code segments.
	* Rules *
	- Do not provide null functions with only placeholders, but detailed implementations.
	- The code should be executed via command line: {Python test.py –i <input_file>}
	- Ensure consistency.
	* Response Format *
	``python
	...
	``
		* Target *
		The compiler/interpreter can't successfully run the code. Analyze and correct the code.
		* Rules *
		- Provide revised code in complete format. The following is not allowed:
		# (same as before)
		# (function remains the same)
		- Use the web search tool for the correct usage of some packages or functions.
		* Compiler/Interpreter Logs*
		{...}
			* User Problem *
			<user_input>{...}</user_input>
			* Target *
			- The program output is listed below. Please first analyze it and modify the algorithm outline to improve the performance by integrating more advanced algorithms.
			* Program Output *
			{...}
			* Rules *
			- Omit any warnings.
			- Refer to the chat history and the context documents.

(a) Modularized code generation	(b) Modularized code integration	(c) Code debugging	(d) Algorithm modification

Figure 8: The prompt template for (a) modularized code generation, (b) modularized code integration, (c) code debugging, and (d) code improvement via algorithm modification

problem, *AutoIOT* adds more stringent rules and requirements to explicitly ask the LLM to avoid generating null functions and verify the availability of imported packages and invoked functions by involving the web search tool. With the prompts shown in Fig. 8(a), the LLM can generate cohesive code segments with detailed comments for each module, facilitating the module integration in the next stage.

CoT 4: Modularized Code Integration. Given the generated code segments, *AutoIOT* prompts the LLM to constructively integrate all modularized code segments and create a cohesive and comprehensive program. Since the code generated for different modules may have disparate input/output variable names, *AutoIOT* first prompts the LLM to ensure the consistency among all the modules and synthesize the final program without null functions as illustrated in Fig. 8(b).

For the convenience of code execution, debugging, and optimization, *AutoIOT* asks the LLM to add a main function so that the program can be directly executed from the command console (*e.g.*, `python3 test.py –i <input_file>`). Moreover, *AutoIOT* also asks the LLM to generate user documentation in Markdown format, specifying how to properly install, execute, and troubleshoot the program for end users.

Remarks. The *automated program synthesis* module facilitates a seamless transformation from natural language to a readily executable program with CoT prompts. *AutoIOT* is regarded as an experienced developer, adept at decomposing complex AIoT tasks into multiple modules, generating modularized code, and organically integrating them. The program synthesis process can be fully automated by the agent.

4.4 Code Improvement

In § 2.2, we found that the LLM can evaluate and improve the code with heavy user intervention. To alleviate the user's workload involved in debugging and code optimization, we develop the *code improvement* module.

Automated Debugging. Upon obtaining the final program after integration, *AutoIOT* constructs a code executor to run the generated code within a virtue environment (*e.g.*, a

sandbox), ensuring safe and controlled code execution. The code executor loads the sensor dataset from the user's local device for program execution and exports the compiler or interpreter output to the LLM. If the program encounters execution issues (*e.g.*, syntax or I/O errors), *AutoIOT* embeds the logs from the compiler into a prompt (Fig. 8(c)) and instructs the LLM to debug the code for several rounds of interactions until the generated code can be executed successfully.

Code Optimization via Algorithm Modification. To achieve better performance, *AutoIOT* progressively refines the synthesized program via several iterations. In particular, *AutoIOT* processes the test dataset with the first version of the integrated program. Then, *AutoIOT* prompts (Fig. 8(d)) the LLM with the context information (*e.g.*, algorithm outline, chat history) of generating the first program as well as the program output, and asks the LLM to improve the performance by integrating more advanced algorithms. Specifically, *AutoIOT* uses the web search tool to search for solutions that can achieve higher accuracies referring to academic papers and relevant webpages. This initiates a new recursive cycle of program synthesis, starting from refining the algorithm outline accordingly, enriching the outline with the retrieved algorithms, generating modularized code for the updated design, combing the modularized code, debugging, and improving code quality. This code optimization cycle is not a one-time process but is repeated multiple times. Empirically, *AutoIOT* takes five iterations to progressively generate five different programs, striking a balance between thoroughness and efficiency. Finally, *AutoIOT* requests the LLM to analyze the execution results of all the programs and select the one that achieves the best performance as the final program.

Fig. 9 shows a specific example of how the synthesized program evolves over multiple iterations. Version 1 is generated with a bug. By providing the error message to the LLM, *AutoIOT* can automatically fix the bug and generate Version 2. *AutoIOT* further instructs the LLM to modify the algorithm and adopt advanced technologies to generate new versions that iteratively achieve higher accuracies.
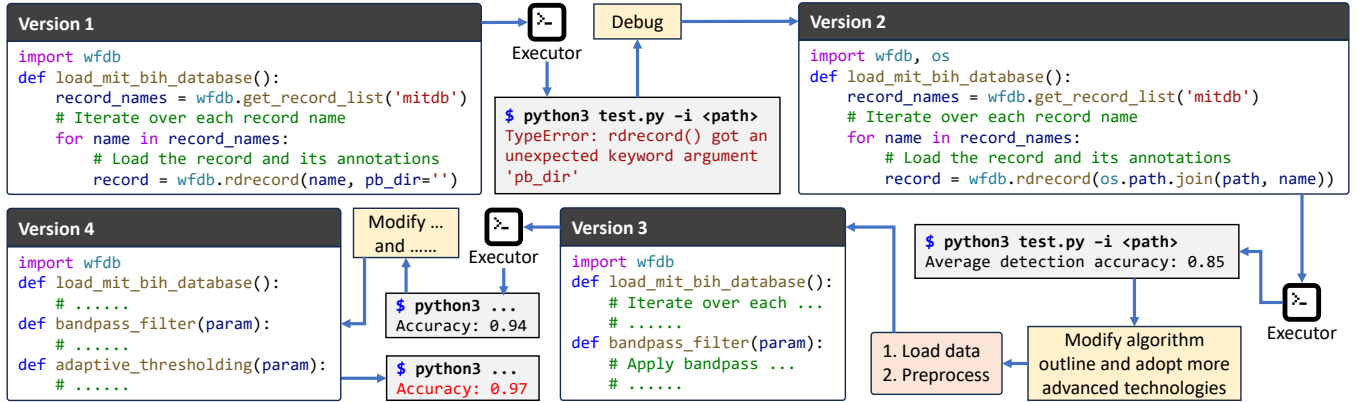
Figure 9: An example of code improvement via iterations (details omitted).

Supporting User-in-the-Loop Optimization. After each iteration of code optimization, *AutoIOT* also provides an interface for the user to *optionally* provide instructions that can help the LLM improve the synthesized program. For example, when the user finds that the LLM fails to recall relevant information from the retrieved contents, the user can prompt *AutoIOT* to refer to a specified algorithm or provide a recent academic paper. This enables a user-in-the-loop optimization that requires minimal user intervention and promotes code optimization iteratively.

Remarks. The *code improvement* module automates code debugging and optimization by leveraging the LLM's proficiency in debugging and refining code based on the compiler and interpreter feedback [23, 55]. This automation not only releases the manual burden but also heralds a new era where AIoT applications can evolve iteratively and autonomously with minimum user intervention.

5 EXPERIMENT SETUP
5.1 Implementation
We implement *AutoIOT* with GPT-4 [16] based on LangChain [7], which provides various tools (*e.g.*, web search engine, vector database, *etc*) for LLMs to collect relevant information. We select Tavily [9] as the web search tool to search for relevant information. It uses OpenAI's text embedding model [56] to convert the retrieved webpages into vector representations. *AutoIOT* then uses Faiss [27] for efficient similarity search of vector representations. The code executor controlled by *AutoIOT* is deployed on a Linux Ubuntu workstation equipped with an NVIDIA RTX 4090 GPU.

5.2 AIoT Applications & Datasets
We select four representative AIoT applications from the domains of healthcare and human activity recognition (HAR). Unlike other program synthesis tasks, these AIoT tasks require domain-specific knowledge and highly specialized algorithms in signal processing and machine learning.

Heartbeat Detection. R-peak detection in electrocardiogram (ECG) data is a crucial task in cardiac signal processing, serving as a foundational step for heart rate variability

studies, and arrhythmia detection [78]. We use MIT-BIH Arrhythmia Database [1] and five representative baseline algorithms, including Hamilton [35], Christov [24], Engzee [31], Pan-Tompkins [60], and SWT [45].

IMU-based Human Activity Recognition. Inertial measurement unit (IMU)-based HAR enables continuous identification of a wide range of daily activities (*e.g.*, sitting, walking) by capturing and analyzing motion characteristics from the IMU data [33, 80]. For the baselines, we select five open-source GitHub repositories: LSTM-RNN [3], 1D-CNN [4], Conv-LSTM [6], BiLSTM [5], and NN [8]. We compare their performance with *AutoIOT* on the WISDM dataset [46].

mmWave-based Human Activity Recognition. mmWave can capture fine-grained human gestures with high resolution [25, 26, 81]. We select the XRF55 dataset [73] with the models proposed in the paper as the baselines, including ResNet-18, 34, 50, 101, and 152. This task is more challenging because: 1) The XRF55 dataset was recently published on websites only a few months ago, which means LLMs have not yet seen knowledge about this dataset; 2) The mmWave data has high dimensionality, necessitating the use of more sophisticated models with optimized configurations.

Multimodal Human Activity Recognition. By leveraging different sensors to capture complementary information, HAR systems can achieve higher robustness and versatility. We select the Harmony dataset [59] containing three sensor modalities: audio, depth image, and radar. The baseline system consists of three encoders, each designed to extract unique features from the respective modalities, followed by feature concatenation and a classifier model. This task is also challenging for *AutoIOT* as it involves the fusion of data from different modalities with cross-modal interaction.

6 EVALUATION
6.1 Metrics
We adopt the following evaluation metrics: 1) *Task accuracy*: we repeat the experiment 10 times and calculate the average task accuracy. In heartbeat detection, we consider task accuracy as the percentage of correctly identified peaks within a
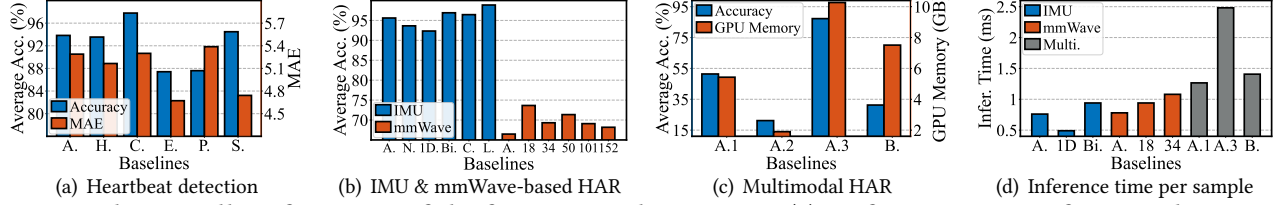
(a) Heartbeat detection   (b) IMU & mmWave-based HAR   (c) Multimodal HAR   (d) Inference time per sample

Figure 10: The overall performance of the four IoT applications. In (a), A. for _AutoIOT_, H. for Hamilton, C. for Christov, E. for Engzee, P. for Pan-Tompkins, and S. for SWT. In (b), N. for NN, 1D for 1D-CNN, Bi. for BiLSTM, C. for Conv-LSTM, L. for LSTM-RNN, and _n_ for ResNet-_n_. In (c) & (d), A.1, A.2, and A.3 for three different _AutoIOT_-generated programs; B. for the baseline in the multimodal HAR application.

predefined tolerance window compared to the ground truth. In HAR, we consider classification accuracy. 2) _MAE_: We use medium absolute error (MAE) to measure the discrepancy in beat positions between the predicted R-peaks and the ground truth. 3) _Communication cost_: we use _psutil_ [2] to monitor the network traffic. 4) _Wall-clock execution time_: we record the total time consumed from the moment the user inputs the problem to the generation of the final inference results for all the sensor data. 5) _Memory consumption_: we record the GPU memory consumption during code execution if AI models are adopted. 6) _Inference time per sample_: we compute the inference time per data sample if AI models are used.

6.2 Performance against Baselines

6.2.1 Average accuracy & MAE. Fig. 10(a) shows the heartbeat detection accuracy with MAE of _AutoIOT_ (denoted as A.) and baselines. First of all, _AutoIOT_ can synthesize a program automatically to achieve comparable performance with baselines in the heartbeat detection task. More surprisingly, the automatically synthesized program can even beat some of the baselines! For example, the synthesized program achieves higher detection accuracy than Pan-Tompkins (P.) and Engzee (E.). Moreover, it yields a lower error rate than Christov (C.) and Pan-Tompkins (P.). To investigate the reasons behind this, we examine and analyze the synthesized program.

We learned: 1) Armed with the web search tool, the synthesized program implemented a few basic as well as sophisticated signal processing methods, including bandpass filtering and stationary wavelet transformation in preprocessing, and adaptive thresholding in postprocessing. Some selected algorithms are well-known and widely adopted, while others are less likely to be chosen, even by experienced programmers with domain expertise. Unlike narrowly focused, well-defined simple programming tasks, AIoT tasks typically require systematic integration of multiple algorithms and components to achieve optimal system performance, which creates opportunities for _AutoIOT_ to explore more possibilities in automatically synthesizing optimized programs that could outperform not all but some representative baselines.

2) Given a single performance objective, we notice that the LLM carries out extensive optimization, sometimes at the cost of other equally important metrics. For example, when instructed to improve the detection accuracy, the synthesized

program sets a larger tolerance window, which increases the chance of correctly detecting heartbeats (true positives) at the cost of increased false positives. Considering AIoT applications' complexity and multiple competing or even contradicting objectives, user requirement specification needs to be as complete and comprehensive as possible, which necessitates domain expertise and system development experience.

3) We found that the webpages returned by the web search tools are often about general algorithms due to their popularity and higher page rankings. Such popular algorithms, however, may not perform the best in domain-specific tasks. With minimum user intervention by providing specialized algorithms, _AutoIOT_ can synthesize programs accordingly and achieve comparable performance to the baselines.

Fig. 10(b) shows the classification accuracy in two HAR tasks. We observe that _AutoIOT_ outperforms NN and 1D-CNN while underperforms BiLSTM, Conv-LSTM and LSTM-RNN. The main reasons are twofold: 1) HAR tasks require both signal processing and machine learning algorithms, increasing the programming complexity to some extent; 2) Training neural networks requires fine-tuning of a vast array of hyper-parameters (_e.g._, network architecture configurations, epoch number, learning rate, optimizer, and loss function). This significantly amplifies the instability of the generated code and calls for careful fine-tuning to achieve the best performance in practice. As a result, _AutoIOT_ surpasses those baselines adopting simple model architectures (NN and one-dimensional CNN) but falls short against baselines using sophisticated architectures (BiLSTM and Conv-LSTM) with highly optimized hyper-parameters. Although during code improvement, some synthesized programs define a set of configurations and adopt a searching strategy to obtain optimal hyper-parameters, the performance still remains slightly lower than some baselines. This is because the determination of the optimal configurations for machine learning models is typically a trial-and-error process, requiring substantial human effort. Fortunately, we observe that if the user provides a potential search space in advance, the LLM can design a search algorithm to try different hyper-parameter configurations and select the one with the best performance.

For multimodal HAR, the input instruction (A1) includes the basic information of the task, _i.e._, the task target, the
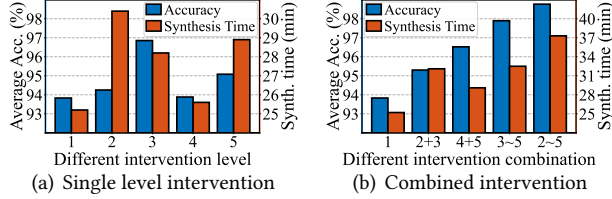
(a) Single level intervention   (b) Combined intervention

Figure 11: Different levels of user intervention.

(a) GPT-4   (b) GPT-3.5

Figure 12: Different numbers of iteration.

(a) Average accuracy & MAE   (b) Wall-clock time & network

Figure 13: Different LLMs. (4 for GPT-4, 3.5 for GPT-3.5, C. for Cohere, A. for Anthropic Claude 2, L. for Llama2-7b, and G. for Gemma-7b.)

dataset specifications, and the output format. Based on that, we create two additional variations: one with a GPU-memory-constrained requirement (A2) and another with a high accuracy requirement (A3). We then feed the instructions into *AutoIOT* and measure the accuracy and inference time of the synthesized programs, with results shown in Fig. 10(c). By analyzing the three synthesized different programs, we observe that: 1) All the synthesized programs adopt a similar workflow as the baseline system, *i.e.*, they first construct three encoders to extract effective features from the three modalities, then concatenate these features and feed them into a classifier for activity recognition. This implies that benefiting from our CoT-based problem-solving paradigm, *AutoIOT* recognizes the workflow and architecture as effective and standard for handling multimodal data-related tasks, which is consistent with most of the existing methods [59]. 2) *AutoIOT* can adjust the generated code to fulfill different requirements. The second program consumes less memory than others due to the resource constraint requirement, resulting in lower accuracy but reduced inference time (Fig. 10(d)). On the other hand, the third program adopts a more complex and larger model architecture, requiring more GPU memory and incurring a longer inference time. Such differences validate the capabilities of *AutoIOT* in accurately understanding and processing natural language-based user requirements. These observations further demonstrate the effectiveness of *AutoIOT* in ensuring the correctness of user requirement understanding and the generated code, benefiting from our automatic self-improvement component.

6.2.2 Communication cost. & wall-clock time. We select heartbeat detection as an example and measure the total communication cost with wall-clock execution time of *AutoIOT* and direct LLM inference as done in Penetrative AI [79]. Specifically, ECG data is first down-sampled and segmented into multiple windows and then embedded into the prompt for LLMs' inference. Experiment results show that *AutoIOT* requires 8MB of network traffic mainly for prompt transmissions, while [79] consumes more than 50MB mainly for sensor data transmissions. Besides, *AutoIOT* takes 25 minutes to complete the task with a dramatic reduction in inference time compared to [79], which needs to send and process all windowed signals with the remote LLM serving.

6.3 Sensitivity Analysis

Different levels of user intervention. To show how *automated program synthesis* improves user experiences, we
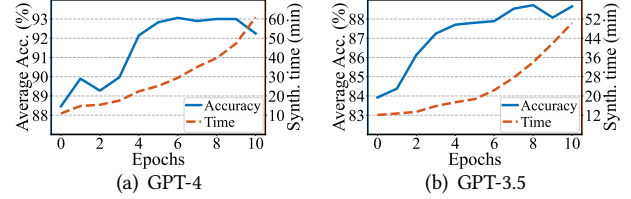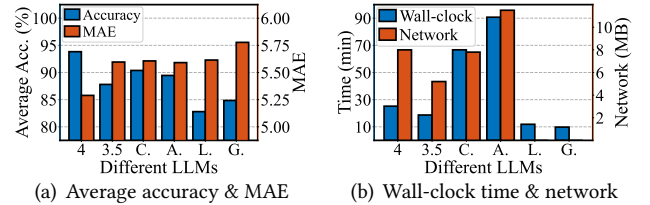
evaluate the *AutoIOT*'s performance under five different levels of user intervention: 1) No intervention; 2) Intervention with user-provided domain knowledge; 3) Intervention with user-specified algorithms for program synthesis; 4) Intervention with user-based debugging; 5) Intervention with user-decided algorithm modification for code improvement. Fig. 11(a) shows the performance of *AutoIOT* with different levels of user intervention. When the user manually instructs the LLM to generate code according to specific hand-picked algorithms (*e.g.*, designed by experts or research papers), the average accuracy can be improved. This user-in-the-loop process becomes particularly advantageous when users have a higher level of domain expertise in AIoT, enabling them to design or select more advanced and robust algorithms. But it leads to increased synthesis time as the LLM has to revise outputs until the user is satisfied. Fig. 11(b) shows the performance with different user intervention combinations. We see that increased user involvement in the program synthesis process correlates with higher accuracy. However, this heightened engagement leads to significantly longer synthesis time and extra user overhead. Note that *AutoIOT* may not always be able to fix bugs and finish program synthesis tasks. In this case, user intervention with minimum effort is still a must. Thus, *AutoIOT* allows users to provide detailed instructions necessary for program synthesis by the LLMs.

Different numbers of improvement iterations. We vary the number of epochs for code optimization from 0 to 10 and evaluate the impact on the synthesized programs. As shown in Fig. 12, with more improvement epochs, the accuracies/synthesis time of the GPT-3.5/-4 generated programs gradually increase. However, after around 5 epochs, the marginal gain of average accuracy starts to diminish while the synthesis time increases dramatically. This is because, with longer conversation history, LLMs may fail to recall past context information and tend to generate inconsistent responses [75]. Therefore, we empirically set the epoch number to five.
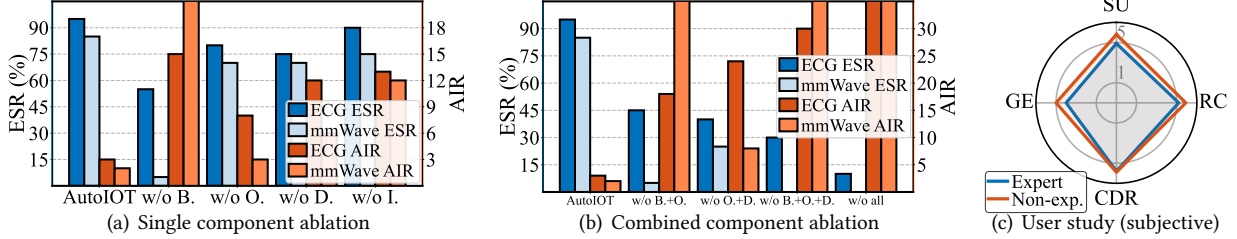
Figure 14: (a) & (b): Ablation study. B. for background knowledge retrieval, O. for algorithm outline generation, D. for detailed design generation, and I. for code improvement. (c): User study on subjective metrics.
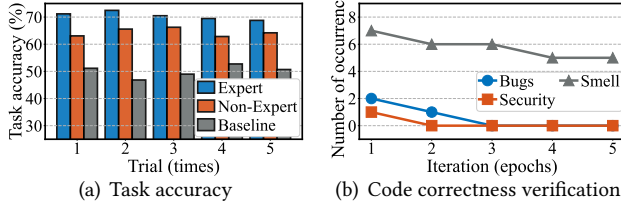

Figure 15: User study (Objective Evaluation).

Impact of Different LLMs. We select the following LLMs for comparison: GPT-4 [16], GPT-3.5 [18], Llama2-7b [72], Cohere [10], Claude 2 [11], and Gemma-7b [70]. Llama2 and Gemma are locally deployed in our lab. We select R-peak detection as an example with the Christov algorithm as the baseline. Fig. 13(a) shows that GPT-4 performs the best. Given the knowledge retrieved by the same tools, LLMs still need strong language understanding and reasoning capabilities to comprehend AIoT tasks and synthesize programs. Experiment results indicate that GPT-4 might have superior performance in language understanding and reasoning capability for this specific task. Although Llama2-7b and Gemma-7b achieve relatively lower accuracy, these two local models offer much faster response speeds.

6.4 Ablation Study

In the ablation study, we use two metrics to evaluate the code quality: 1) *Execution success rate* (ESR): the proportion of the code that can be executed successfully for the first time. 2) *Average iteration round* (AIR): the average number of improvement iterations required to achieve 80% accuracy. **Background knowledge retrieval.** To explore the influence of the *background knowledge retrieval* module, we disable the web search tool and the knowledge database. We then instruct the LLM to synthesize 20 different programs with no user intervention. Fig. 14(a) shows the ESR and AIR of two AIoT tasks. We see that the ESR of heartbeat detection drops slightly while the ESR of mmWave-based HAR exhibits a significant drop. This is because the mmWave-based HAR uses a newly published dataset, which has not been seen by the LLM. Therefore, the LLM does not know the dimensionality of the dataset and only randomly configures the hyper-parameters of the neural network. Additionally, both applications require larger numbers of iterations to improve the accuracy of synthesized programs. We note that

mmWave-based HAR even fails to achieve the expected accuracy (thus, its AIR is marked as infinite). A similar phenomenon is also observed in Fig. 14(b). The experiment results indicate that the background knowledge retrieval module plays a pivotal role for the LLM to retrieve up-to-date domain knowledge to augment the program synthesis process.

Chain-of-thought. We evaluate the contribution of the algorithm outline generation step and the detailed design generation step during the CoT prompting, respectively. As shown in Fig. 14(a), when only one step is enabled, we observe a slight drop in ESR and an increase in AIR for both applications. When we disable both steps, the ESR drops significantly as shown in Fig. 14(b). Without the explicit guidance specified in the two steps, the LLM cannot synthesize executable programs and presents null functions with placeholders. Therefore, the CoT method with detailed instructions emulating the software development lifecycle plays a crucial role in helping LLMs synthesize executable programs. **Code improvement.** The *code improvement* module contains automated debugging and code optimization. Since automated debugging is essential to ensure the executability of synthesized programs, we only conduct an ablation study on the code optimization step. With the program after the debugging step, we evaluate the *code improvement* module by directly instructing the LLM to modify the program over several iterations without providing the compiler or interpreter feedback. As shown in Fig. 14(a), without the *code improvement* module, the ESR is almost unaffected while the AIR increases significantly. This is because generating an executable program with no syntax errors is seldom influenced by this module. However, without feedback from the compiler or interpreter, more iterations are required since the LLM does not know which step of the algorithm should be modified or improved. Besides, due to the laziness of LLMs [52], synthesized programs tend to adopt simple and popular algorithms. This necessitates the *code improvement* module, which progressively directs the LLM to explore more advanced algorithms to improve the synthesized program.

6.5 User Study

To investigate the utilities of *AutoIOT*, we conduct a user study (N=20) by inviting 5 expert and 15 non-expert users, whose detailed background information is listed in Table 1. The expert users are PhD students and professors with work
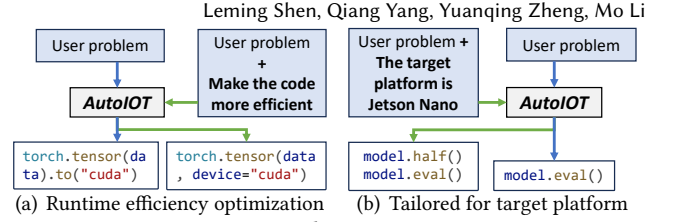
Leming Shen, Qiang Yang, Yuanqing Zheng, Mo Li

Table 1: Participant information of user study (N=20)

Category	Background Information
Gender	Female (45%), Male (40%), Prefer not to say (15%)
Age	Under 18 (10%), 18-30 (75%), 30-39 (10%), 40 and older (5%)
Education	Bachelor (20%), Master (15%), Doctoral (60%), Others (5%)
English	Beginner (5%), Intermediate (25%), Advanced (60%), Fluent (10%)
Expertise	Expert (25%), Non-expert (75%)

(a) Runtime efficiency optimization   (b) Tailored for target platform

Figure 16: Further experiments.

or research experience in the IoT field and have developed many IoT applications. We select human activity recognition using RFID data (XRF55 dataset [73]) as the IoT application, where a 1D Conv-based ResNet18 is the baseline.

Objective Evaluation. We first repeatedly measure the average task accuracy (classification accuracy) after executing the synthesized programs, with the results shown in Fig. 15(a). We see that the programs synthesized by the two groups of users outperform the baseline across multiple trials. Besides, the programs synthesized for experts typically perform better than those for non-experts. The main reason is that experts tend to provide more information in the specifications (*e.g.*, the dataset format, the training workflow). Consequently, *AutoIOT* can synthesize programs with more advanced algorithms and detailed specifications for expert users. Next, we use SonarQube [13] to verify the correctness of the generated code, including bug/logic errors, security issues, and code smells [14] after every improvement iteration. Code smells are not bugs but bad coding styles (*e.g.*, variable name mismatching regular expression) or potential weaknesses (*e.g.*, package version incompatibility). From Fig 15(b), we observe that several bugs and one security issue present initially are ultimately fixed by *AutoIOT*. *AutoIOT* may not be able to address all code smells, as they are closely related to coding styles [50]. Applying code smell correction to the retrieved algorithms and programs can be promising for *AutoIOT* to iteratively detect and fix code smell-related issues.

Subjective Measurement. We ask the users to execute the synthesized programs and rate *AutoIOT* based on four subjective metrics: 1) *System Utility* (**SU**) measures the user's overall satisfaction with *AutoIOT*'s performance; 2) *Requirement Coverage* (**RC**) evaluates how well the user requirements are fulfilled by *AutoIOT*; 3) *Code & Documentation Readability* (**CDR**) measures the clarity and structure of the code and documentation; 4) *Generation Efficiency* (**GE**) accesses how acceptable the waiting time is for synthesizing the final program. All the above metrics are rated by the users on a scale from 1 (not at all) to 6 (more than expected). As shown in Fig. 14(c), the average GE of both user groups reaches 4.5, indicating that the waiting time for *AutoIOT* to synthesize the final program is acceptable for most users. Additionally, we find that non-experts tend to give higher scores (SU, RC, and CDR) than experts. Further examination reveals that non-experts tend to under-specify their requirements. Surprisingly, LLMs can sometimes provide comprehensive responses to meet their requirements. This ability is rooted in LLMs' extensive training on diverse datasets and retrieved online information, enabling them to infer and bridge the gaps with relevant information [32, 69].

7 DISCUSSION

Runtime Efficiency. *AutoIOT* focuses on generating functionally correct code for IoT data processing. While the runtime efficiency of the programs is not optimized, the users can specify the extra requirements in the prompts. Fig. 16(a) shows an example that given the user specification, the generated code adopts CUDA optimization and thereby achieves similar runtime efficiency compared to expert-optimized programs. This is because *AutoIOT* can retrieve and learn from hand-crafted optimizations via various online sources.

Workflow Generalizability. Existing IoT applications can be primarily categorized into four types based on the specific stages of the IoT workflow they address: 1) data collection applications rely on sensors (*e.g.*, IMU and radar) to gather raw information from the environment; 2) data transmission applications enable seamless communication across IoT networks to cloud or edge systems; 3) data processing applications typically adopt advanced algorithms or AI models to analyze and process IoT data; 4) decision-making and actuation applications parse the processed data to perform task automation or actuator control. In this paper, *AutoIOT* is designed primarily for IoT data processing tasks by offering end-to-end solutions with executable programs. The prompts designed in *AutoIOT* are not limited to specific data patterns (time-series or high-dimensional) and processing pipelines (sequentially or parallelly). For other IoT application workflows, users can specify the requirements in the prompt. For example, users can first instruct *AutoIOT* to develop a WiFi data collection application. Then, the synthesized program can be deployed on WiFi-related devices to capture WiFi data. Next, users can further request *AutoIOT* to process and analyze the collected WiFi data to perform HAR or other tasks. Moreover, some IoT applications may require executing multiple programs simultaneously for cross-device communication or synchronization. To tackle such applications, a possible solution can be deploying multiple *AutoIOT* agents with advanced collaboration mechanisms for enhanced cross-device and cross-program interaction. We leave the comprehensive generalization to other types of IoT tasks beyond pipelined data processing for future work.

Real-World Deployment. We show that *AutoIOT* can synthesize functionally correct IoT programs in Python with data processing accuracy as the main performance metric. *AutoIOT* is not limited to specific target IoT platforms. For

example, regarding those resource-constrained MCU-class devices [63, 74], various practical requirements (*e.g.*, power management and network protocols) of the IoT device can be factored into *AutoIOT*'s prompts for program synthesis. Fig. 16(b) shows an example where a user explicitly specifies Jetson Nano, which has limited GPU resources, as the target platform. *AutoIOT* will then adopt half-precision training and inference for the AI model to save GPU memory. Moreover, users can even provide *AutoIOT* with handbooks of some dedicated IoT devices for reference.

Memorization Issue. In *AutoIOT*, sometimes the synthesized programs' performance cannot be improved, even after multiple improvement iterations. This is because LLMs may forget the previous context, resulting in inconsistent code improvement suggestions during iterations [22]. In such cases, our user-in-the-loop optimization allows users to optionally provide instructions that can help the LLM improve the synthesized programs. For instance, users can explicitly instruct the LLM to use the Pan-Tompkins algorithm for ECG data processing. Additionally, we can further upgrade *AutoIOT* by adopting existing memorization enhancement methods, such as context compression via RAG [67] and iterative summarization [68]. By integrating these orthogonal approaches with *AutoIOT*, each self-improvement iteration can contribute more positively to the performance of the generated code with enhanced context consistency.

Privacy Concerns. In *AutoIOT*, only user requirements are transmitted to the cloud LLM for processing. We proactively instruct *AutoIOT* to treat user configuration data (*e.g.*, the local file path) as program input. As such, during local execution, users can directly input the private configurations in the console (§ 4.3), rather than pre-defining them in the prompts for cloud LLMs. To further mitigate privacy concerns, one possible solution can be deploying a local LLM to handle all code generation and debugging tasks [64, 66].

8 RELATED WORK

Integration of LLMs with AIoT. Existing integration methods have two main types: 1) Prompt-based methods embed raw sensor data into tailored prompts and instruct LLMs to perform various AIoT tasks. HARGPT [43] and LLMSense [58] embed textualized sensor data into prompts to show the proficiency of LLMs in comprehending IoT sensor data. They require transmitting raw sensor data to LLM servers, suffering similar issues as Penetrative AI. 2) Fine-tuning-based methods retrain LLMs with labeled datasets containing sensor data inference examples. LLM4TS [21] fine-tunes LLMs using sensor data with labels for time-series data prediction. However, these works demand high compute and memory resources. In contrast, *AutoIOT* explores a new approach to automatically synthesizing programs for AIoT applications without extra overheads.

Table 2: *AutoIOT* vs SOTA code LLMs

Type	Name	Mod. Gen.	Sys. Gen.	RAG	Auto.	Auto Debug & Impro.
Code Gen.	DeepSeek Coder [34]	✓	✗	✗	✗	✗
	CodeLlama [62]	✓	✗	✗	✗	✗
	WizardCoder [54]	✓	✗	✗	✗	✗
Task Auto.	AutoGPT [12]	✓	✓	✗	✓	✗
	MetaGPT [37]	✓	✓	✗	✓	✗
	AutoIOT	✓	✓	✓	✓	✓

Code LLMs. Recent advances in code LLMs [34, 62] have demonstrated the potential to revolutionize software development. They can produce functionally correct code across various programming languages and input compiler output into LLMs for debugging and program refining [29, 88]. However, they require high computing resources with carefully selected datasets (StarCoder [49] needs an 815GB dataset and 512 A100 80GB GPUs). Worse still, they are unaware of the latest advances in highly specialized domains and cannot generate comprehensive solutions for complex IoT tasks. *AutoIOT* draws strength from these works and addresses specific technical challenges in IoT program synthesis, which require ever-evolving domain knowledge that the above LLMs have not yet assimilated. Table 2 compares *AutoIOT* with SOTA code LLMs. LLM-based code generation synthesizes functionally correct programs for a well-defined module, whereas LLM-based task automation generates a complete automation chain from requirements to solutions. *AutoIOT* lies at the intersection of these two approaches, with the objective of code generation and task automation. Additionally, RAG and automated code improvement jointly improve the code generation process in *AutoIOT*.

9 CONCLUSION

We propose *AutoIOT*, an LLM-driven automated natural language programming system for AIoT applications. Our system features three novel technical modules: background knowledge retrieval, automated program synthesis, and code improvement, transforming natural language descriptions into executable programs. Our experiments demonstrate the competitive performance of *AutoIOT* in synthesizing programs for a variety of AIoT applications, with comparable performance in challenging AIoT tasks and sometimes outperforming some representative baselines. This showcases the strong potential of exploiting the embedded common knowledge of LLMs to evolve AIoT application development.

REFERENCES

[1] 2001. https://www.physionet.org/content/mitdb/1.0.0/
[2] 2009. https://psutil.readthedocs.io/en/latest/.
[3] 2018. https://github.com/bartkowiaktomasz/har-wisdm-lstm-rnns.
[4] 2018. https://github.com/akshaykulkarni07/activity_recognition.
[5] 2018. https://github.com/bartkowiaktomasz/har-wisdm-bidirectional-lstm-rnns.
[6] 2020. https://github.com/coloriz/HAR-WISDM_ar.
[7] 2022. https://github.com/langchain-ai/langchain.
[8] 2022. https://github.com/AthanJohn/Human-Activity-Recognition-WISDM.
[9] 2023. https://github.com/assafelovic/gpt-researcher.
[10] 2023. https://www.cohere.ai.
[11] 2023. https://www.anthropic.com.
[12] 2023. https://github.com/Significant-Gravitas/AutoGPT.
[13] 2024. https://www.sonarsource.com/
[14] 2024. https://rules.sonarsource.com/python/RSPEC-2316/
[15] 2025. https://platform.openai.com/account/limits
[16] 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).
[17] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
[18] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *NeurIPS* 33 (2020), 1877–1901.
[19] Dongqi Cai, Yaozong Wu, Shangguang Wang, Felix Xiaozhu Lin, and Mengwei Xu. 2023. Efficient federated learning for modern nlp. In *ACM MobiCom*. 1–16.
[20] Jiani Cao, Chengdong Lin, Yang Liu, and Zhenjiang Li. 2022. Gaze tracking on any surface with your phone. In *ACM SenSys*.
[21] Ching Chang, Wei-Yao Wang, Wen-Chih Peng, and Tien-Fu Chen. 2023. Llm4ts: Aligning pre-trained llms as data-efficient time-series forecasters. *arXiv preprint arXiv:2308.08469* (2023).
[22] Juo-Tung Chen and Chien-Ming Huang. 2023. Forgetful large language models: Lessons learned from using LLMS in robot programming. In *Proceedings of the AAAI Symposium Series*.
[23] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).
[24] Ivaylo I Christov. 2004. Real time electrocardiogram QRS detection using combined adaptive threshold. *Biomedical engineering online* 3, 1 (2004), 1–9.
[25] Kaiyan Cui, Leming Shen, Yuanqing Zheng, Fu Xiao, and Jinsong Han. 2024. Talk2Radar: Talking to mmWave Radars via Smartphone Speaker. In *IEEE INFOCOM*.
[26] Kaiyan Cui, Qiang Yang, Yuanqing Zheng, and Jinsong Han. 2023. mmRipple: Communicating with mmWave radars through smartphone vibration. In *ACM IPSN*.
[27] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).
[28] Mingzhe Du, Anh Tuan Luu, Bin Ji, and See-Kiong Ng. 2024. Mercury: An Efficiency Benchmark for LLM Code Synthesis. *arXiv preprint arXiv:2402.07844* (2024).
[29] Shukai Duan, Nikos Kanakaris, Xiongye Xiao, Heng Ping, Chenyu Zhou, Nesreen K Ahmed, Guixiang Ma, Mihai Capota, Theodore L Willke, Shahin Nazarian, et al. 2023. Leveraging Reinforcement Learning and Large Language Models for Code Optimization. *arXiv preprint arXiv:2312.05657* (2023).
[30] Alexander Dunn, John Dagdelen, Nicholas Walker, Sanghoon Lee, Andrew S Rosen, Gerbrand Ceder, Kristin Persson, and Anubhav Jain. 2022. Structured information extraction from complex scientific text with fine-tuned large language models. *arXiv preprint arXiv:2212.05238* (2022).
[31] Willem AH Engelse and Cees Zeelenberg. 1979. A single scan algorithm for QRS-detection and feature extraction. *Computers in cardiology* 6, 1979 (1979), 37–42.
[32] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines* 30 (2020), 681–694.
[33] Ming Gao, Lingfeng Zhang, Leming Shen, Xiang Zou, Jinsong Han, Feng Lin, and Kui Ren. 2023. Exploring practical acoustic transduction attacks on inertial sensors in MDOF systems. *IEEE TMC* (2023).
[34] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y 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).
[35] Pat Hamilton. 2002. Open source ECG analysis. In *Computers in cardiology*. IEEE, 101–104.
[36] Yuze He, Chen Bian, Jingfei Xia, Shuyao Shi, Zhenyu Yan, Qun Song, and Guoliang Xing. 2023. VI-Map: Infrastructure-Assisted Real-Time HD Mapping for Autonomous Driving. In *ACM MobiCom*. 1–15.
[37] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352* (2023).
[38] Ningning Hou, Yifeng Wang, Xianjin Xia, Shiming Yu, Yuanqing Zheng, and Tao Gu. 2025. MoLoRa: Intelligent Mobile Antenna System for Enhanced LoRa Reception in Urban Environments. In *ACM SenSys*.
[39] Ningning Hou, Xianjin Xia, Yifeng Wang, and Yuanqing Zheng. 2024. One shot for all: Quick and accurate data aggregation for LPWANs. *ACM ToSN* 32, 3 (2024), 2285–2298.
[40] Ningning Hou, Xianjin Xia, and Yuanqing Zheng. 2023. Jamming of LoRa PHY and countermeasure. *ACM ToSN* 19, 4 (2023), 1–27.
[41] Kai Huang and Wei Gao. 2022. Real-time neural network inference on extremely weak devices: agile offloading with explainable AI. In *ACM MobiCom*. 200–213.
[42] Ritu Jain and Ugrasen Suman. 2015. A systematic literature review on global software development life cycle. *ACM SIGSOFT Software Engineering Notes* 40, 2 (2015), 1–14.
[43] Sijie Ji, Xinzhe Zheng, and Chenshu Wu. 2024. HARGPT: Are LLMs Zero-Shot Human Activity Recognizers? *arXiv preprint arXiv:2403.02727* (2024).
[44] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A Code Evolution Framework via Large Language Models. *arXiv preprint arXiv:2306.02907* (2023).
[45] Vignesh Kalidas and Lakshman Tamil. 2017. Real-time QRS detector using stationary wavelet transform for automated ECG analysis. In *IEEE BIBE*. 457–461.
[46] Jennifer R Kwapisz, Gary M Weiss, and Samuel A Moore. 2011. Activity recognition using cell phone accelerometers. *ACM SigKDD Explorations Newsletter* 12, 2 (2011), 74–82.
[47] Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. CodeChain: Towards Modular Code Generation Through Chain of Self-revisions with Representative Sub-modules. arXiv:2310.08992 [cs.AI]
[48] Sunjae Lee, Junyoung Choi, Jungjae Lee, Munim Hasan Wasi, Hojun Choi, Steve Ko, Sangeun Oh, and Insik Shin. 2024. Mobilegpt: Augmenting llm with human-like app memory for mobile task automation.

In *ACM MobiCom*.

[49] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[50] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2019. Deep learning based code smell detection. *IEEE TSE* (2019).

[51] Jianwei Liu, Wenfan Song, Leming Shen, Jinsong Han, Xian Xu, and Kui Ren. 2021. Mandipass: Secure and usable user authentication via earphone imu. In *IEEE ICDCS*.

[52] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *TACL* 12 (2024), 157–173.

[53] Xin Liu, Daniel McDuff, Geza Kovacs, Isaac Galatzer-Levy, Jacob Sunshine, Jiening Zhan, Ming-Zher Poh, Shun Liao, Paolo Di Achille, and Shwetak Patel. 2023. Large Language Models are Few-Shot Health Learners. *arXiv preprint arXiv:2305.15525* (2023).

[54] 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).

[55] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *NeurIPS* 36 (2024).

[56] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. 2022. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005* (2022).

[57] 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*.

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

[59] 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.

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

[61] Pengrui Quan, Xiaomin Ouyang, Jeya Vikranth Jeyakumar, Ziqi Wang, Yang Xing, and Mani Srivastava. 2025. SensorBench: Benchmarking LLMs in Coding-Based Sensor Processing. In *ACM HOTMOBILE*.

[62] 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).

[63] 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 *ACM MobiSys*.

[64] Leming Shen, Qiang Yang, Xinyu Huang, Zijing Ma, and Yuanqing Zheng. 2025. GPIoT: Tailoring Small Language Models for IoT Program Synthesis and Development. In *ACM SenSys*.

[65] Leming Shen and Yuanqing Zheng. 2023. FedDM: data and model heterogeneity-aware federated learning via dynamic weight sharing. In *IEEE ICDCS*.

[66] Leming Shen and Yuanqing Zheng. 2024. IoTCoder: A Copilot for IoT Application Development. In *ACM MobiCom*.

[67] Kaize Shi, Xueyao Sun, Qing Li, and Guandong Xu. 2024. Compressing Long Context for Enhancing RAG with AMR-based Concept Distillation. *arXiv preprint arXiv:2405.03085* (2024).

[68] Shichao Sun, Ruifeng Yuan, Ziqiang Cao, Wenjie Li, and Pengfei Liu. 2024. Prompt Chaining or Stepwise Prompt? Refinement in Text Summarization. *arXiv preprint arXiv:2406.00507* (2024).

[69] Alex Tamkin, Miles Brundage, Jack Clark, and Deep Ganguli. 2021. Understanding the capabilities, limitations, and societal impact of large language models. *arXiv preprint arXiv:2102.02503* (2021).

[70] Gemma Team et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295* (2024).

[71] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[72] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[73] Fei Wang, Yizhe Lv, Mengdie Zhu, Han Ding, and Jinsong Han. 2024. XRF55: A Radio Frequency Dataset for Human Indoor Action Analysis. *ACM IMWUT* (2024).

[74] Kun Wang, Zimu Zhou, and Zhenjiang Li. 2024. LATTE: Layer Algorithm-aware Training Time Estimation for Heterogeneous Federated Learning. In *ACM MobiCom*.

[75] Qingyue Wang, Liang Ding, Yanan Cao, Zhiliang Tian, Shi Wang, Dacheng Tao, and Li Guo. 2023. Recursively summarizing enables long-term dialogue memory in large language models. *arXiv preprint arXiv:2308.15022* (2023).

[76] 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. *NeurIPS* (2022).

[77] 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*.

[78] Lu Wu, Xiaoyun Xie, and Yinglong Wang. 2021. ECG enhancement and r-peak detection based on window variability. In *Healthcare*. MDPI.

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

[80] Huatao Xu, Pengfei Zhou, Rui Tan, Mo Li, and Guobin Shen. 2021. Limu-bert: Unleashing the potential of unlabeled data for imu sensing applications. In *ACM SenSys*.

[81] Hongfei Xue, Qiming Cao, Yan Ju, Haochen Hu, Haoyu Wang, Aidong Zhang, and Lu Su. 2022. M4esh: mmwave-based 3d human mesh construction for multiple subjects. In *ACM SenSys*. 391–406.

[82] Qiang Yang, Kaiyan Cui, and Yuanqing Zheng. 2023. VoShield: Voice liveness detection with sound field dynamics. In *IEEE INFOCOM*.

[83] Qiang Yang and Yuanqing Zheng. 2022. Deepear: Sound localization with binaural microphones. *IEEE TMC* 23, 1 (2022), 359–375.

[84] Qiang Yang and Yuanqing Zheng. 2023. Aquahelper: Underwater sos transmission and detection in swimming pools. In *ACM SenSys*.

[85] Yuqing Yang, Lei Jiao, and Yuedong Xu. 2024. A queueing theoretic perspective on low-latency llm inference with variable token length. In *IEEE WiOpt*.

[86] 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 *ACM MobiCom*.

[87] Shiming Yu, Xianjin Xia, Ziyue Zhang, Ningning Hou, and Yuanqing Zheng. 2024. FDLoRa: Tackling Downlink-Uplink Asymmetry with Full-duplex LoRa Gateways. In *ACM SenSys*.

[88] Li Zhong, Zilong Wang, and Jingbo Shang. 2024. LDB: A Large Language Model Debugger via Verifying Runtime Execution Step-by-step. *arXiv preprint arXiv:2402.16906* (2024).