LLM-Based Code Development Model with Active Prompt Adjustment and On-Device Validation

Hong Su[1]

[1]School of Computer Science, Chengdu University of Information Technology

January 28, 2025

LLM-Based Code Development Model with Active Prompt Adjustment and On-Device Validation

Hong Su

***Abstract*—Large language models (LLMs) have been widely adopted to assist in IoT device code development. However, prompts are typically provided by the user and adjusted based on the user's judgment when the generated code fails to function as expected. As a result, users often have to test candidate solutions one by one. In this paper, we propose an on-device development model that dynamically adjusts prompts based on on-device validation results and device parameter retrieval, creating a closed-loop process for code development. The process begins with the user providing requirements and device-specific information to the LLM, which generates both functional and validation code. The generated code is validated directly on the device by the model. If the code fails continuously beyond a specified threshold, the model iteratively refines the prompt by modifying its scope and word order. Furthermore, candidate solutions from the LLM are organized into a tree structure, enabling the efficient reuse of common steps across solutions. The results show that prompt adjustments based on device parameters and validation enhance accuracy, while the tree structure approach improves the efficiency of testing candidate solutions by 25%.**

***Index Terms*—Active Development Model, Large Language Models (LLMs), On-Device Validation, Active Prompt Adjustment**

I. Introduction

The integration of Large Language Models (LLMs) into the development of Internet of Things (IoT) devices has gained significant attention due to the increasing demand for automated and efficient development workflows [1] [2]. Traditionally, when users seek code or solutions for IoT development, they manually extract relevant information from the device, combine it with their requirements, and form structured prompts that are fed into LLMs to generate corresponding code [3] [4]. This process, while powerful, often faces several challenges that hinder its effectiveness in practical IoT development scenarios.

One of the primary challenges in using LLMs for IoT development is the user's responsibility to correctly identify and provide the necessary information for the LLM, without the ability to adjust the prompt based on on-device validation and real-time device data. LLMs are highly dependent on the quality of the input prompts, and the results they generate are directly influenced by the clarity, specificity, and completeness of these prompts [5]. If the prompt is vague, incomplete, or misleading, it can lead to irrelevant or incorrect outputs, regardless of how many attempts the user makes [6]. This presents a significant barrier, as users often lack the expertise to formulate the most effective prompts, further complicating the process.

Providing more information in a prompt does not always lead to better results. The scope of the prompt plays a pivotal role in determining the accuracy of the LLM's response [7]. In this context, the scope refers to the range or extent of information included in the prompt; for simplicity, this paper considers the scope as the different words within the prompt. A prompt that is too broad may introduce irrelevant or contradictory details, while one that is too narrow may exclude essential context. Consequently, dynamically adjusting the prompt's scope to align with the specific requirements and constraints of the IoT device, as well as incorporating on-device validation results, is critical for generating accurate and contextually relevant outputs.

For instance, consider the case of a user working with an Ubuntu Qt Python application on a touch-screen device in landscape mode. By default, the onboard keyboard (one of the Ubuntu on-screen keyboards [1]) may be hidden in full-screen mode on Ubuntu 22.04 due to a misalignment of the keyboard with the window. If the user inputs a generic prompt like "Ubuntu Qt Python full screen landscape onboard," the LLM might not provide any useful information. This is because the issue is specific to the device and its configuration, and the prompt may not include the correct parameters to generate a relevant solution. This example highlights a fundamental limitation of the current passive model, where users must manually craft prompts that may not always lead to effective solutions.

To address these challenges, this paper proposes an active approach for IoT device development that integrates LLMs with an on-device parameter collection, prompt adjustment strategies, and on-device validation. In this active development mode, device parameters are dynamically extracted and used to refine the prompts, adjusting the scope iteratively based on the validation results. The system continues to adjust the prompt until a workable solution is identified. Furthermore, when multiple potential solutions are generated, a hierarchical judgment tree structure is used to evaluate their applicability. Solutions are grouped into subtrees based on shared characteristics, with incompatible solutions filtered out early. Finally, a validation step ensures that the generated solution is tested on the actual device, guaranteeing its relevance and functionality.

H. Su is with the School of Computer Science, Chengdu University of Information Technology, Chengdu, China, 610041.
E-mail: suguest@126.com.

[1]https://manpages.ubuntu.com/manpages/focal/man1/onboard.1.html

The term "active" refers to the closed-loop process in IoT device development, where tasks such as code generation, validation, device parameter retrieval, and prompt adjustments are managed automatically by the model. This stands in contrast to traditional models where users are responsible for manually preparing and adjusting prompts, performing validations, and collecting error information. By automating these processes, the proposed model enhances accuracy and ensures the correct solution is identified without relying solely on user judgment.

The contributions of this paper are summarized as follows: (1) Active IoT Device Development Mode: This paper introduces an active development mode that integrates LLMs with real-time device parameters, dynamic prompt adjustment strategies, and on-device validation. Unlike traditional passive models, where users manually craft prompts, our approach automatically adjusts the prompts based on on-device validation results and relevant parameters collected from the IoT device. By fine-tuning the prompt scope, the system ensures that the LLM generates solutions that are more relevant and applicable to the specific IoT device context, improving the quality and applicability of the generated solutions. (2) Impact of Prompt Scope: We highlight the critical impact that the scope of the prompt has on the output quality of generated code. A prompt that is too broad or too narrow can lead to inaccurate or incomplete results. Our approach emphasizes the importance of dynamic scope adjustment through on-device validation, ensuring that the generated solutions are tailored to the IoT device's specific needs and constraints. (3) Hierarchical Judgment Tree: To address the issue of multiple potential solutions being generated, we introduce a hierarchical judgment tree that organizes solutions based on their relevance and applicability to the IoT device. This structure enables efficient evaluation by filtering out incompatible solutions early in the process, ensuring that only the most viable solutions are considered for further development.

The rest of this paper is organized as follows. Section II provides an overview of related work. Section III introduces the model for active on-device code development. Section IV discusses the impact of unrelated prompt words and word order on transformer outputs, highlighting the necessity of active prompt adjustment. In Section V, we present the verification results and perform an in-depth analysis. Finally, Section VI concludes the paper with a summary of our contributions.

II. Related Work

The integration of Large Language Models (LLMs) with Internet of Things (IoT) device development has gained increasing attention in recent years, driven by the need for automated, efficient development workflows. This section reviews the existing literature on LLMs in IoT development, prompt engineering, validation strategies, and solution evaluation frameworks.

A. LLM-based Code Generation

1) LLMs in IoT Development: Several studies have explored the use of LLMs for code generation and development support in IoT systems. For example, LLMs such as OpenAI's GPT and Google's BERT have been employed to automate the generation of code snippets for embedded systems (e.g., Arduino, Raspberry Pi) and IoT devices [8], [9]. These studies focus on the ability of LLMs to generate programming code based on user input, aiming to streamline the development process. However, a common limitation of these systems is that they heavily rely on user-crafted prompts, which often require technical expertise, and fail to dynamically adapt to the unique constraints and parameters of specific IoT devices.

2) Prompt Engineering and Dynamic Adjustments: Prompt engineering is a key challenge when using LLMs for IoT device development. Several works have addressed this issue by proposing methods for dynamically adjusting prompts based on context and feedback. Some approaches leverage meta-prompting or iterative prompt refinement to improve the quality of LLM outputs [10], [11]. However, these studies primarily focus on static or pre-defined adjustment strategies and do not incorporate on-device validation or dynamic scope adjustments based on real-time device parameters.

In contrast, our approach emphasizes an active method where the scope of the prompt is dynamically adjusted based on the IoT device's parameters and feedback from on-device validation. This real-time feedback loop enables more precise and relevant outputs, a feature that is often missing in the existing methods.

3) Solution Evaluation and Decision Trees: The problem of selecting the most relevant solution from multiple candidates generated by LLMs has been addressed in various ways. Some works use heuristic-based filtering to rank potential solutions based on predefined criteria [12]. Others leverage decision trees or rule-based systems to guide the selection of solutions [13]. These approaches typically rely on a static evaluation framework that does not adapt based on the specific context of the IoT device.

In contrast, our approach introduces a hierarchical judgment tree that evaluates solutions based on their applicability to the IoT device. This structure categorizes solutions into subtrees, allowing for efficient identification of the most relevant solutions and filtering out incompatible ones early in the process. The hierarchical tree is guided by real-time feedback from the device, ensuring that the most promising solutions are prioritized.

B. On-Device Validation

On-device validation is an important aspect of ensuring that generated code works in real-world IoT environments. Existing works on validation often focus on simulation-based environments where generated solutions are tested against predefined benchmarks [14]. While these methods ensure correctness in a controlled environment, they lack the real-time feedback required for optimizing prompts and adjusting solutions based on specific device constraints.

Some studies, like those by [15] and [16], attempt to validate solutions in the context of the hardware by testing them on actual devices after the initial code generation. However, these approaches are typically passive, relying on the user to manually identify and test potential solutions, which can be time-consuming and error-prone. In our work, we propose an active IoT device development mode, where the system automatically adjusts the prompts and solutions based on on-device validation results, significantly reducing the need for manual intervention.

C. Conclusion

While there has been significant progress in using LLMs for IoT development, many existing approaches still face challenges related to the correct formulation of prompts, validation of generated solutions, and selection of applicable solutions. Our proposed active IoT device development mode addresses these challenges by incorporating active prompt adjustment, on-device validation, and a hierarchical solution evaluation framework. By integrating these strategies, our approach aims to enhance the efficiency, accuracy, and relevance of LLMs in IoT device development.

III. Active On board code development model

A. Motivation

Currently, when users need to develop code for IoT devices, they must manually extract relevant device information, transform it into prompts, and input it into a Large Language Model (LLM). However, this approach presents several challenges.

(a) The user must provide accurate prompt information to the LLM, which requires an understanding of the specific data the LLM needs. However, users often fail to supply sufficient or precise information, or they may provide misleading words, which leads to incorrect results regardless of how many attempts are made. As LLMs rely on the prompts provided by users, their output is determined by the attention mechanism of the transformer model, meaning the quality of the results directly depends on the quality of the input prompts, as shown in (1).

For example, in a Python Qt project running on an Ubuntu device, the on-board keyboard might not appear in full-screen landscape mode. If the user simply inputs the broad description "Python Qt project in an Ubuntu" as the prompt, the LLM will likely produce no useful information.

$$\text{result} \xrightarrow{\text{related to}} \{\text{prompt1}, \text{prompt2}, \ldots\} \tag{1}$$

(b) Scope of Prompts: Simply providing more information does not necessarily lead to better results. As illustrated in (1), the scope of the prompt (i.e., the information provided) needs to be carefully adjusted. Users often lack the knowledge of how to modify the prompt's scope to obtain the correct result. On-device information queries and validation can help in this process by dynamically adjusting the prompt's scope using algorithms or models. For example, removing specific terms (e.g., "Qt" or "Python") from the prompt may better align it with the device's requirements and improve the accuracy of the generated response.

(c) Verification of Solutions: While LLMs can suggest multiple solutions, they do not verify whether a solution works on a specific IoT device. Users are left to manually test each proposed solution, even though some may be incompatible with the device. As shown in Equation (2), LLMs often provide several possible solutions for the same prompt, but not all of them are viable.

$$prompt \xrightarrow{produce} \{\text{Solution}_1, \text{Solution}_2, \ldots,\} \tag{2}$$

Thus, we propose an active IoT device development model that incorporates LLMs, collects device parameters, forms prompts, dynamically adjusts the prompt scope, and validates solutions until a workable solution is found. Additionally, if there are multiple potential solutions, we organize them into a decision tree. Solutions that share a common subtree can be tested sequentially, while unrelated solutions can be tested independently.

B. Active LLM-Based Development Model

The active LLM-based development model aims to optimize IoT device code development by dynamically adjusting prompts and validating solutions on the device. This approach reduces the need for user interaction, enhances the efficiency of code development, and improves the accuracy of results by iteratively refining prompts based on validation feedback. The process is illustrated in Figure 1, and the corresponding model is called **active LLM-based development model**.

Definition 1: **Active LLM-based development model** refers to a model which automatically collects information from IoT devices, forms it into query prompts for LLMs, and generates code. When a solution is proposed, it is tested on the IoT device via the on-device interface. The validation results are fed back as adjustments to the prompt. If the verification fails, the model will either try another solution or modify the prompts before feeding them back into the LLM. If the verification succeeds, the process is complete. This validation process guides the prompt adjustments, enabling the model to automatically modify the prompts to solve the issue as effectively as possible.

Active LLM-based development model has two key features, active prompt adjustment and active validation.

Active Prompt Adjustment: Active Prompt Adjustment refers to the dynamic modification of the input prompts based on ongoing feedback from the IoT device. In the context of LLM-based IoT device development, this process involves iteratively refining the prompts provided to the large language model (LLM). Initially, a prompt is generated based on the user's requirements and device-specific information. However, as the model generates code and runs validation, the prompt is adjusted to improve the accuracy and efficiency of the generated code. This iterative adjustment ensures that the LLM receives the most relevant and precise information at each step, leading to more accurate solutions over time.

Active Validation: Active Validation refers to the process of on-device, immediate validation of the generated code. This feature requires the LLM to generate executable code that can be tested directly on the IoT device. Active Validation involves continuously assessing the performance of the generated solution and providing immediate feedback that guides prompt refinement. The feedback loop ensures that the generated code is constantly optimized based on real-world results from the device, thereby improving the overall quality and relevance of the solution.

1) Active Prompt Adjustment Process Based on On-Device Validation: The proposed method leverages active prompt adjustment and on-device validation to iteratively refine input prompts. This process ensures that all necessary device-specific information is retrieved and that all possible prompt combinations (in terms of scope and order) are explored. The feedback loop is formalized as follows:

(1) **Initial Prompt:** The user provides an initial query or task description, which is fed into the LLM. This initial prompt includes both user-defined requirements and any preliminary device-specific information retrieved via the device interface.

(2) **Validation and Feedback:** The LLM generates a potential solution based on the current prompt. This solution is then validated by executing it on the IoT device, and the outcome of this validation determines the next steps:
- **Failure (¬Success):** If the solution does not meet the requirements, the system retrieves any additional device-specific information required by the LLM and adjusts the prompt accordingly:
  - Modify the scope of the prompt by adding or removing terms.
  - Change the order of the prompt components to explore alternative input structures.
  - Refine the phrasing of the prompt to better align with the device context.
- **Success (Success):** If the solution meets the requirements, the process is terminated, and the solution is accepted as valid.

(3) **Prompt Adjustment:** The system dynamically adjusts the prompt using a combination of scope, order, and phrasing refinements:
- **Scope Adjustment:** Iteratively expand or narrow the prompt's scope by adding relevant device-specific information retrieved from the device interface or removing irrelevant terms.
- **Order Adjustment:** Iteratively permute the order of the prompt components to explore all possible input sequences.
- **Phrasing Refinement:** Alter the phrasing of the prompt to enhance clarity and better align with the context of the IoT task.

2) Formal Description of the Process: The iterative process can be described as:

$$\text{Prompt}_i \xrightarrow{\text{LLM}} \text{Solution}_i \xrightarrow{\text{Validate}} \{\text{Success}, \neg\text{Success}\}$$

If ¬Success, the next prompt is adjusted as follows:

$$\begin{aligned}\text{Prompt}_{i+1} = {} & \text{AdjustScope}(\text{Prompt}_i) \\ & + \text{AdjustOrder}(\text{Prompt}_i) \\ & + \text{IncorporateDeviceInfo}(D) \qquad (3)\end{aligned}$$

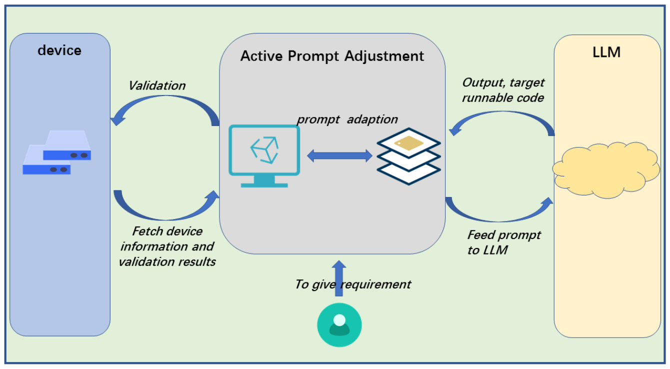

Fig. 1. Active Prompt Adjustment Process

where D represents the device-specific information retrieved from the IoT device interface.

3) Adjustment Algorithm for Prompt Refinement: In the above process, three aspects of the prompt are adjusted: the scope of the prompt, the order of the prompt, and the device information. The adjustment of the scope is done iteratively, i.e., the scope is either increased or decreased step by step. In each adjustment, the order of the prompt is also modified iteratively, meaning all possible orders are explored. Meanwhile, the change in device information is based on the information required by the Language Model (LLM). This iterative adjustment process is formalized in Algorithm 1.

4) Brief Proof of Convergence and Correctness of the Prompt Adjustment Algorithm: To prove that if there exists a prompt capable of producing the correct code, then the proposed methods of active prompt adjustment and validation will ensure that the model eventually generates the correct code, we outline the proof as follows:

a) Proof Outline:: Let P denote the set of all possible prompts, and let $p_{\text{correct}} \in P$ represent the specific prompt that generates the correct code for the given IoT task.

b) Step 1: Assumption of Information Availability: When a requirement or error information is submitted, the model can retrieve any necessary device-related information through the device interface. This ensures that all relevant information required for generating p_{correct} can be accessed. Thus, the system has access to all possible information needed to construct the correct prompt.

c) Step 2: Prompt Scope and Order Adjustment: The algorithm iteratively adjusts both the scope and order of the prompt to ensure all possible combinations of prompts are explored:

- The scope is dynamically modified by adding relevant device-specific information or removing irrelevant terms.
- The order of prompt components is permuted to account for variations in how the LLM interprets and prioritizes different input structures.
- This iterative process ensures that every possible combination of prompt scope and order is eventually tested.

d) Step 3: Iterative Feedback Process: The model iteratively refines the prompt based on validation feedback:

Algorithm 1 Prompt Adjustment Based on Validation Feedback

```
1: Input: Initial prompt $P_0$, IoT device interface $D$, and validation feedback $V$
2: Output: Refined prompt $P_{\text{final}}$
3: $P \leftarrow P_0$
4: while Validation fails ($V$) do
5:     if Device information is required by the LLM then
6:         Retrieve required device information via the interface $D$
7:         $P \leftarrow \text{IncorporateDeviceInfo}(P, D)$
8:     end if
9:     if Scope needs adjustment then
10:        for each step in scope adjustment (increase or decrease) do
11:            $P \leftarrow \text{AdjustScope}(P)$
12:            if Validation succeeds ($V$) then
13:                Break
14:            end if
15:        end for
16:    end if
17:    if Order needs adjustment then
18:        for each permutation of the current prompt order do
19:            $P \leftarrow \text{AdjustOrder}(P)$
20:            if Validation succeeds ($V$) then
21:                Break
22:            end if
23:        end for
24:    end if
25: end while
26: Return $P_{\text{final}}$
```

- If the generated code fails validation, the scope is adjusted iteratively, either increasing (to include missing critical details) or reducing (to eliminate irrelevant noise) the prompt.
- Similarly, the order of prompt components is adjusted to explore alternate interpretations by the LLM.
- This feedback-driven refinement guarantees the exploration of all possible prompts within P.

e) Step 4: Inductive Hypothesis: Let S_k represent the set of prompts that have undergone k adjustments. We hypothesize that if $p_{\text{correct}} \in P$, the iterative process will eventually reach p_{correct} after a finite number of adjustments.

f) Step 5: Convergence:

- If p_{correct} lies within the scope of the current prompt, the iterative process will refine the prompt to isolate and retain the critical elements required for correctness.
- If p_{correct} is outside the current scope, the iterative expansion of the prompt ensures that missing critical details are eventually included.
- By exploring all permutations of scope and order, the algorithm systematically converges on p_{correct}.

g) Step 6: Termination: The process terminates once the validation step confirms the correctness of the generated code. Since the model iteratively refines the prompt to explore all possible configurations, it is guaranteed to produce the correct prompt and code, provided p_{correct} exists.

h) Conclusion: The proposed algorithm ensures:

1) **Information Completeness:** All relevant device-related information required to construct the correct prompt is accessible through the device interface.
2) **Exhaustive Exploration:** By iteratively adjusting the scope and order of the prompt, the algorithm explores all possible prompt configurations in P.

Thus, if a correct prompt p_{correct} exists, the model will eventually produce the correct code. Conversely, if no correct prompt exists, the generated code will be unrelated to the device's information, violating the attention mechanism of the transformer, which inherently relies on input relevance.

C. Tree Structure for Parallel Acknowledgement

In many problem-solving scenarios, multiple candidate solutions may be generated by the LLM. These solutions often share common steps but may diverge at certain points as they progress towards the final resolution. To optimize the search for the most effective solution, we organize these solutions into a tree structure. Each branch of the tree represents a distinct candidate solution, with the nodes corresponding to intermediate steps. Common steps are grouped together at shared nodes, facilitating the parallel exploration of different solution paths.

When a solution fails at a certain point, the model identifies the steps shared with other candidate solutions. By skipping the re-execution of these common steps—a process referred to as the shared step method—the model saves time and resources. In contrast, the reset step method involves restarting from the beginning of each new solution. Leveraging shared progress enables the model to avoid redundant computations, thereby improving overall efficiency.

Consider the general structure of two solutions, so_1 and so_2, as follows:

$$\text{so}_1 : \{s_1, s_2, s_3\}, \quad \text{so}_2 : \{s_1, s_2, s_3'\}$$

Here, steps s_1 and s_2 are common to both solutions, while the remaining steps are different. If a failure occurs at step s_3 in solution so_1, the model can revert to the shared steps $\{s_1, s_2\}$ and continue with the divergent steps $\{s_3'\}$ from so_2, as shown in Algorithm 2.

1) Time and Resource Efficiency: By avoiding redundant computations, specifically those associated with shared steps, the model can save both time and computational resources. The time saved can be quantified as follows:

$$\text{Time}_{\text{saved}} = \text{Time}_{\text{so}_1} - \text{Time}_{\text{Common}(\text{so}_1,\text{so}_2)}$$

where $\text{Time}_{\text{so}_1}$ represents the total time to execute solution so_1, and $\text{Time}_{\text{Common}(\text{so}_1,\text{so}_2)}$ is the time spent on the common steps. By focusing on the divergent steps, the model minimizes unnecessary computations, thus improving overall efficiency.

Algorithm 2 Solution Reversion and Divergence

```
1: Input: Two candidate solutions $so_1$ and $so_2$, with steps $s_1, s_2, s_3$ and $s'_1, s'_2, s'_3$, respectively.
2: Output: Updated solution with minimal re-execution of steps.
3: if Failure occurs in solution $so_1$ then
4:     Revert to the common steps between $so_1$ and $so_2$, i.e., $\text{Common}(so_1, so_2)$
5:     Proceed with the remaining divergent steps from solution $so_2$, i.e., $\text{Divergent}(so_2)$
6: end if
7: Return the updated solution after reversion and divergence.
```

D. Active Query Information through On-device Interface

The integration of the IoT device with the Language Model (LLM) requires an effective and responsive communication channel that allows the LLM to query and interact with the device's environment. This interaction is essential for refining the generated code and diagnosing any issues during execution. The IoT device provides this communication capability through an On-device interface, which acts as a bridge between the LLM and the device itself.

1) Interface to IoT Device — On-device Interface: The On-device interface is a crucial component that facilitates communication between the LLM and the IoT device. It must be capable of accepting queries from the LLM, executing the corresponding instructions, and returning relevant results. Depending on the device's capabilities, the On-device interface can take various forms:

1) **Runnable Interface:** In cases where the LLM generates commands that are directly executable on the IoT device, such as shell scripts or system commands, the On-device interface must be able to run these commands. For example, in an Ubuntu-based system, the interface could execute bash scripts generated by the LLM, providing immediate feedback to the model about the outcome of the commands.
2) **Generated Interface:** In some situations, the LLM may need to generate code that the IoT device is not immediately capable of executing. For instance, if the device does not have a built-in command execution environment, the On-device interface can provide a mechanism for the LLM to generate executable code. The device can then run this code to collect data or perform specific operations. If the device lacks this capability, the LLM can assist in developing the necessary functionality, or the system may implement additional software layers to enable this communication.

The On-device interface not only serves as the communication channel for running commands but also plays a key role in querying device-specific information. Once the device-specific data is collected, it is sent back to the LLM for further refinement of the prompt, ensuring that the generated code is accurate and effective.

2) Providing Complete On-device Environment Information for Issue Identification: For effective problem diagnosis and resolution, it is essential that the On-device interface is capable of gathering detailed environmental information from the IoT device. This includes real-time system data such as device status, hardware resources, software configurations, error logs, and network conditions.

A potential solution involves forming shell scripts or specialized system queries that collect and provide the required data to the LLM. These scripts can be automatically generated based on the prompts and executed by the On-device interface. The collected data can then be returned to the LLM for analysis, enabling it to identify issues, refine the code, and suggest corrective actions.

For instance, if the LLM detects that a particular functionality is failing to execute as expected, it can query the device for specific environmental conditions that might be influencing the performance. These could include memory usage, CPU load, or available storage space. By acquiring this comprehensive set of environmental data, the LLM can make informed adjustments to the code, ensuring that it aligns with the actual device environment.

This process also supports a dynamic feedback loop, where the LLM continually receives and processes environment-specific data to optimize the code generation process. This capability significantly enhances the model's ability to diagnose issues accurately and adapt the code to meet the device's specific requirements.

The On-device interface, by providing continuous and detailed environmental feedback, allows for more precise identification of problems and ensures that the generated code is well-suited to the IoT device's real-world conditions.

IV. Impact of Unrelated Prompt Words and Word Order on Transformer Outputs

In this section, we aim to demonstrate that both the presence of unrelated prompt words and the order of prompt words can significantly affect the output of Transformer models. This underlying behavior supports the need for active prompt adjustment to optimize performance.

A. Dependence on Prompt Words

Transformers process input sequences using embeddings, self-attention mechanisms, and feedforward layers. The model's output is a function of the input tokens, represented as embeddings in a high-dimensional space.

1) Prompt Words Representation: Each input word w_i in a sequence $\{w_1, w_2, \ldots, w_n\}$ is mapped to an embedding $\mathbf{e}_i \in \mathbb{R}^d$, where d is the embedding dimension. This embedding includes positional information as shown in the following equation:

$$\mathbf{e}_i = \text{WordEmbedding}(w_i) + \text{PositionalEncoding}(i)$$

2) Self-Attention Mechanism: The Transformer model uses a self-attention mechanism to compute the relevance of each word to every other word in the sequence. The attention score between words w_i and w_j is calculated as:

$$\text{Attention}(w_i, w_j) = \text{softmax}\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_k}}\right)$$

where:

- $\mathbf{q}_i, \mathbf{k}_j$ are query and key vectors derived from the embeddings $\mathbf{e}_i$ and $\mathbf{e}_j$,
- d_k is the dimensionality of the key vector.

The attention mechanism ensures that the model's output depends on the relationships between all input words, as encoded in the attention weights.

3) Output Computation: The output for each word w_i is computed as a weighted sum of the value vectors $\mathbf{v}_j$, scaled by the attention scores:

$$\mathbf{z}_i = \sum_{j=1}^{n} \text{Attention}(w_i, w_j)\mathbf{v}_j$$

Thus, the output is intrinsically dependent on the input sequence, including both the individual tokens and their order.

B. Impact of Related Words on Correct Results

Transformers are sensitive to semantic relationships between words due to their reliance on self-attention. Related words contribute to meaningful attention patterns, while unrelated words can disrupt these patterns.

1) Definition of Related Words: Let w_i and w_j be two words in the input sequence. We define a *relatedness function* $R(w_i, w_j)$ as the semantic similarity between w_i and w_j, which is measured as:

$$R(w_i, w_j) = \text{cosine}(\text{Embedding}(w_i), \text{Embedding}(w_j))$$

$R(w_i, w_j)$ is high if w_i and w_j are semantically related, meaning they are closer in the embedding space.

2) Effect of Related Words:

- **Attention Distribution**: For related words, $\text{Attention}(w_i, w_j)$ is high, leading to a greater contribution of $\mathbf{v}_j$ to the output $\mathbf{z}_i$.
- **Coherence**: Related words create coherent attention patterns, allowing the model to better understand context and generate more accurate results.

3) Effect of Unrelated Words:

- **Noise in Attention**: For unrelated words, $\text{Attention}(w_i, w_j)$ is low or misaligned, which introduces noise in the output $\mathbf{z}_i$.
- **Degradation**: A high proportion of unrelated words dilutes the meaningful relationships between related tokens, leading to a degradation in the model's performance.

4) Impact of Word Order: In addition to the semantic relationships between words, the order of the words in the input sequence also affects the output. Since Transformers use positional encoding to capture the order of words, the rearrangement of words can change the attention distribution and, consequently, the output.

- **Order Sensitivity**: If the order of related words is changed, the attention mechanism may shift, leading to incorrect associations and reducing the coherence of the generated output.
- **Disruption of Context**: Changing the order of words, even if they are semantically related, may confuse the model, causing it to misinterpret the context or generate less accurate results.

Thus, the order of the words plays a crucial role in determining the effectiveness of the attention mechanism.

5) Quantitative Impact of Relatedness and Order: Consider an input sequence S with n tokens, where r are related tokens and u are unrelated tokens ($r + u = n$). The model's ability to generate correct outputs depends on both the relatedness ratio $\frac{r}{n}$ and the order of the words. The effective accuracy can be expressed as:

$$\text{Accuracy}(S) \propto \frac{r}{n} \cdot f(\text{order})$$

where $f(\text{order})$ represents the impact of the word order on the model's performance. As the proportion of unrelated words increases or the word order becomes more disordered, the accuracy decreases due to increased noise and disrupted attention patterns.

Experimental studies have validated this relationship, showing that sequences with a higher proportion of unrelated words or poorly ordered words tend to degrade model performance.

C. Conclusion

The output of Transformers relies heavily on the semantic relationships between words and their order within a sequence. Related words enhance performance by forming meaningful connections, whereas unrelated words or changes in word order disrupt attention patterns, leading to diminished results. This evidence underscores the importance of input word relatedness and word order in transformer-based models, emphasizing the need for active prompt adjustments, such as reordering words or altering the prompt scope to optimize results. Additionally, active validation provides feedback on code correctness, further refining the prompt adjustment process.

V. Verification

In this section, we verify the impact of prompt scope changes on the performance of large language models (LLMs) and assess the efficiency of using shared steps across candidate solutions.

A. Impact of Scope Changes on LLM Results

To verify the impact of scope changes in prompts, we use the example of a touch screen keyboard where the onboard

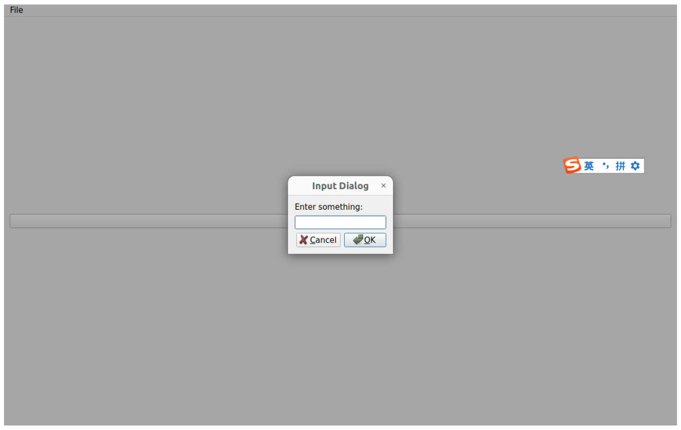

Fig. 2. Touch screen keyboard is not shown

keyboard is not displayed to the user, and the code is generated by the large language model (LLM). The application was developed using Python and Qt in landscape mode, featuring an input box. The corresponding code is shown in Algorithm 3.

The experiments were conducted in an Ubuntu 22.04 environment on a YANMENG industry computer (YM10M-YD) with the following specifications: Intel Celeron N5100 @ 1.1GHz (quad-core CPU), 8 GiB of RAM, and a Goodix Capacitive touchscreen.

Algorithm 3 Virtual Keyboard Integration in a PyQt Application

```
1: function SHOW_INPUT_DIALOG
2:     call SHOW_VIRTUAL_KEYBOARD()    ▷ Display the virtual keyboard
3:     (text, ok) ← QInputDialog.getText(self, ...)    ▷ Show input dialog for keyword
4:     call HIDE_VIRTUAL_KEYBOARD()    ▷ Hide the virtual keyboard
5: end function
6: function SHOW_VIRTUAL_KEYBOARD
7:     call subprocess.Popen(["onboard"])    ▷ Launch the system virtual keyboard
8: end function
9: function HIDE_VIRTUAL_KEYBOARD
10:    call subprocess.run(["pkill", "onboard"], stdout=DEVNULL, stderr=DEVNULL)    ▷ Terminate the virtual keyboard process
11: end function
```

Issues: When the program runs and the user clicks the input dialog, the onboard keyboard (invoked by the code) does not appear, as shown in Figure 2.

We use chatGPT as the LLM to resolve the issue. Three ways during the scope and order of prompt adjustment are chosen as for the analysis, we call it three methods: Method1, Method2 and Method3 with prompt "Qt python project onboard cannot show in landscape Ubuntu 22.04", "onboard can not show in landscape Qt python project Ubuntu 22.04", and "onboard can not show in landscape Ubuntu 22.04" respectively.

The results of the study are presented in Figure 3, which compares three methods across four key metrics: the number of given suggestions, the number of changes made to the Python code, the onboard settings, and the number of suggestions that resolved issues. Each subplot in the figure illustrates the trends for one metric across all three methods.

The diagram labeled "Number of Suggestions that Resolved Issues" represents the number of solutions that successfully address the issues in our project. Only Method 3 provides candidate suggestions that effectively resolve the issues. In contrast, Method 1 and Method 2, despite having more information in the prompt for the LLM, do not contain the correct solutions. This suggests that providing more information may, in some cases, negatively impact the accuracy of the suggestions.

The diagram titled "Onboard Settings" pertains to the onboard settings, with a recommendation to configure other related aspects. If the developer considers additional configurations of onboard settings, they may identify the correct solution. This indicates that exploring LLM generated solutions in a comprehensive manner can yield correct outcomes.

The diagram labeled "Number of Changes Made to the Python Code" illustrates modifications to the Python code. Since the keyboard is not visible in the development environment, errors may arise in the Python code due to its interaction with the interface. This also can introduce an interference factor, as the issue might not be directly related to the Python code itself. It is evident that, at times, the suggestions made result in significant changes to the Python code. This suggests that the prompt itself influences the generated suggestions. Thus, with the prompt scope adjustment algorithm, when iterating, the prompt becomes more suggestive of potential solutions.

Additionally, we observe that the order in which prompts are provided impacts the suggested solutions. Method 1, which begins with "Qt Python Project," consistently suggests changes to the Python code. On the other hand, Method 2, which starts with "Onboard," leads to a different set of suggestions. Method 3, which omits "Qt Python Project," does not suggest changes to the Python code, highlighting the impact of prompt order on solution generation.

The diagram titled 'Number of Given Suggestions' represents the total number of solutions provided by the LLM. Statistical analysis using a one-way ANOVA test indicates that the prompt has minimal impact on the number of suggestions generated. The test results show an F-statistic of 2.1847 and a p-value of 0.1206, which exceeds the common significance level of 0.05. Consequently, we fail to reject the null hypothesis, indicating that the variation in the number of suggestions across the three methods is not statistically significant.

In conclusion, both the scope and order of the prompt have a significant impact on the solutions provided. Adjusting the scope and order can enhance the accuracy of the suggestions.

B. *Efficiency of using knowledge tree*

The verification process evaluates two methods—Shared Steps and Reset Steps—by comparing their cumulative and

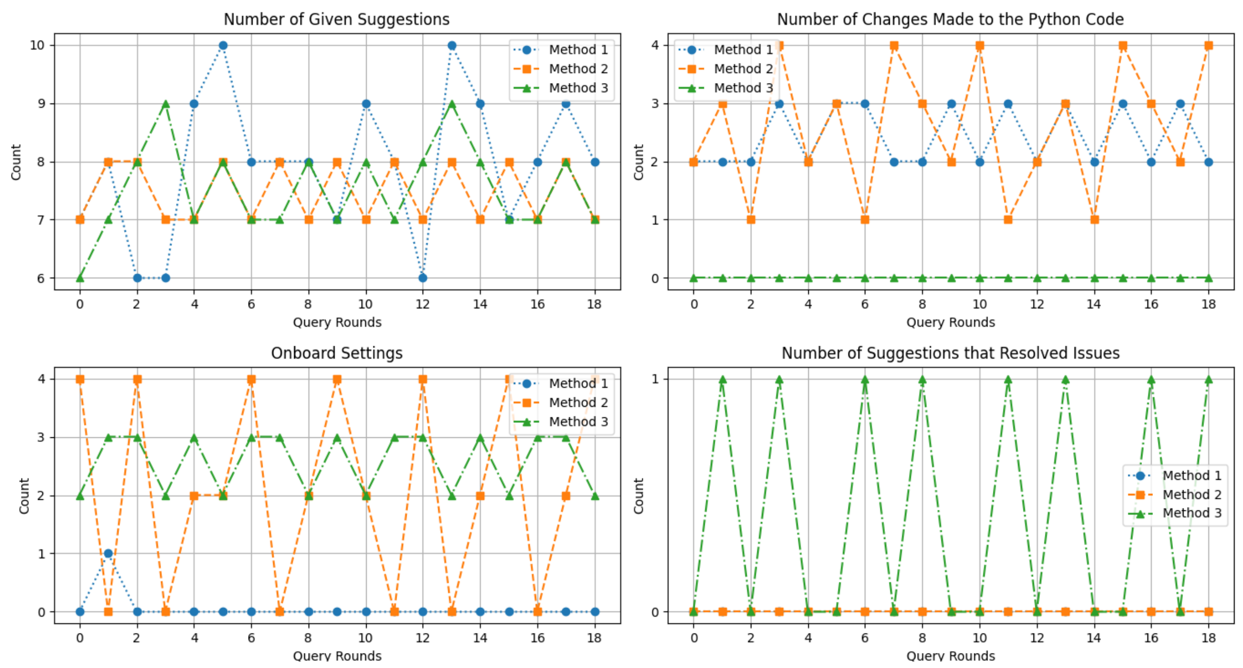

Fig. 3. Comparison of Key Measurements Across Methods with Scope and Order Variations in Prompts

average time efficiency. In the simulation, the path lengths were selected randomly from the set $\{2, 4, 8, 16\}$, chosen to simulate various complexities of real-world tasks, ranging from simpler to more complex scenarios. The correctness of a method was determined probabilistically with a success rate $p = 0.6$, representing a moderate likelihood of success. This value was selected to model realistic conditions where methods may fail but still have a reasonable chance of success.

The transition probabilities for shared steps between methods were defined as {1:1/2, 2:1/4, 3:1/8, 4:1/16}, reflecting a natural decrease in the likelihood of shared steps as the number of steps increases. Each pair is formatted as 'length of shared steps: probability.' For example, '3: 1/8' indicates a probability of $1/8$ that two candidate solutions share 3 steps.

For the time calculation, the total time for each method was computed as follows:

- Shared Steps Method: The total time was calculated by summing the time for common steps (reused from the previous method) and the time for remaining steps (requiring execution from the new method):

$$T_{\text{shared}} = (\text{common steps} \times t_{\text{step}}) + (\text{remaining steps} \times t_{\text{step}})$$

- Reset Steps Method: The total time for the reset method was calculated by summing the time for all steps, starting from the first step after each switch as no common steps are used in the verification:

$$T_{\text{reset}} = \text{total steps} \times t_{\text{step}}$$

Where t_{step} is the time to complete one step.

The verification was performed for 25 independent test rounds. In each round, the methods were tested under identical conditions, ensuring fairness in comparison. The results were analyzed by plotting the accumulated time over test rounds and comparing the average times for both methods. The total time difference between the methods was computed as:

$$\Delta T = \sum T_{\text{reset}} - \sum T_{\text{shared}}$$

The results are shown in Figure 4. From the figure, it is evident that the accumulated time for the Shared Steps method is consistently less than that of the Reset Steps method. The total time for the Reset Steps method was 204 seconds, whereas the Shared Steps method required 164 seconds, resulting in a time saving of 40 seconds. This represents a near 25% improvement over the Reset Steps method. The reduction in accumulated time demonstrates the advantage of reusing shared steps between methods, which allows the Shared Steps method to optimize transitions more effectively.

In addition to time savings, the Shared Steps method also requires fewer revert steps. When a path fails, reverting to shared steps ensures a smoother and more efficient switch to other methods. In contrast, the Reset Steps method requires restarting from the beginning for each new path, leading to unnecessary repetition and increased effort. By minimizing revert steps, the Shared Steps method not only saves time but also aligns better with scenarios where efficient recovery is critical, such as real-time systems or resource-constrained environments.

These findings highlight the practical benefits of the Shared Steps approach in reducing computational overhead and improving operational efficiency. The results provide an indication that leveraging shared progress can be a valuable strategy for optimizing performance in systems with probabilistic outcomes and frequent switching requirements.

VI. Conclusion

In this paper, we presented the LLM-based code development model that actively adjusts prompts to optimize IoT

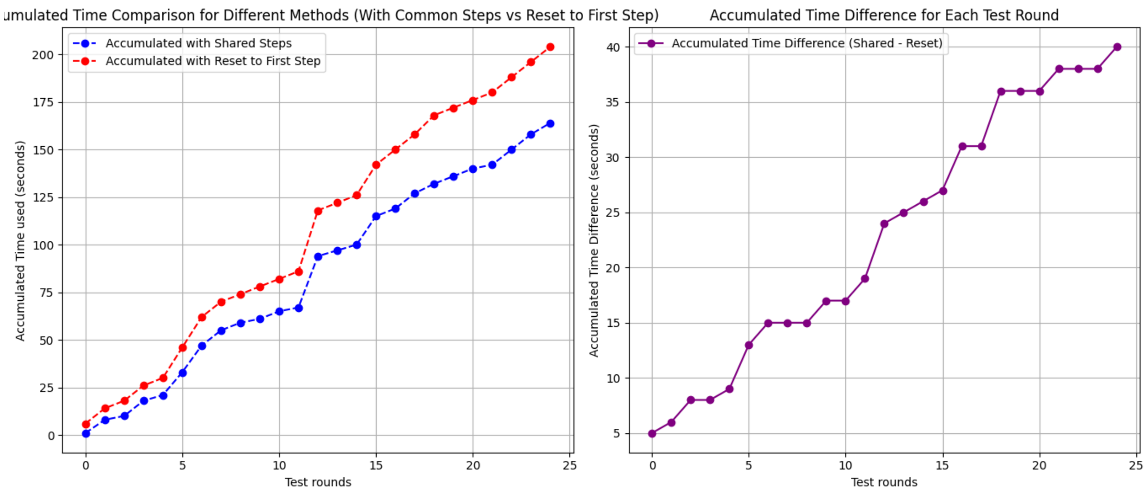

Fig. 4. Accumulated time for shared path method and individual path method

device code generation and validation. By incorporating on-device parameter retrieval and immediate validation feedback, our model iteratively refines the input prompts given to large language models (LLMs), leading to more accurate and efficient code generation. Additionally, we introduced a novel method of organizing candidate solutions into a tree structure, enabling the efficient reuse of common steps and reducing redundant computations. The verification results demonstrate that adjusting prompts based on device parameters and validation significantly improves accuracy, while the tree structure approach enhances the efficiency of candidate solution testing, yielding a 25% improvement in testing efficiency in the simulation. The proposed model presents a promising solution to the challenges of developing reliable and efficient IoT device code, emphasizing the potential for integrating immediate validation and dynamic prompt refinement into future LLM-based development frameworks.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61772352; the Science and Technology Project of Sichuan Province under Grant No. 2019YFG0400, 2018GZDZX0031, 2018GZDZX0004, 2017GZDZX0003, 2018JY0182, 19ZDYF1286, 2020YFG0322, and the R&D Project of Chengdu City under Grant No. (2019-YF05-01790-GX).

References

[1] N. S. Mathews and M. Nagappan, "Test-driven development and llm-based code generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1583–1594.

[2] J. Sauvola, S. Tarkoma, M. Klemettinen, J. Riekki, and D. Doermann, "Future of software development with generative ai," *Automated Software Engineering*, vol. 31, no. 1, p. 26, 2024.

[3] H. Cui, Y. Du, Q. Yang, Y. Shao, and S. C. Liew, "Llmind: Orchestrating ai and iot with llm for complex task execution," *IEEE Communications Magazine*, 2024.

[4] N. Tang, "Towards effective validation and integration of llm-generated code," in *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2024, pp. 369–370.

[5] Y. Zhang, H. Fei, D. Li, and P. Li, "Promptgen: Automatically generate prompts using generative models," in *Findings of the Association for Computational Linguistics: NAACL 2022*, 2022, pp. 30–37.

[6] E. Jahani, B. S. Manning, J. Zhang, H.-Y. TuYe, M. Alsobay, C. Nicolaides, S. Suri, and D. Holtz, "As generative models improve, people adapt their prompts," *arXiv preprint arXiv:2407.14333*, 2024.

[7] L. Wang, X. Chen, X. Deng, H. Wen, M. You, W. Liu, Q. Li, and J. Li, "Prompt engineering in consistency and reliability with the evidence-based guideline for llms," *npj Digital Medicine*, vol. 7, no. 1, p. 41, 2024.

[8] Z. Englhardt, R. Li, D. Nissanka, Z. Zhang, G. Narayanswamy, J. Breda, X. Liu, S. Patel, and V. Iyer, "Exploring and characterizing large language models for embedded system development and debugging," in *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–9.

[9] Z. Tafferner, B. Illés, O. Krammer, and A. Géczy, "Can chatgpt help in electronics research and development? a case study with applied sensors," *Sensors*, vol. 23, no. 10, p. 4879, 2023.

[10] S. Raaj Hiraou, "Optimising hard prompts with few-shot meta-prompting," *arXiv e-prints*, pp. arXiv–2407, 2024.

[11] T. S. Kim, Y. Lee, J. Shin, Y.-H. Kim, and J. Kim, "Evallm: Interactive evaluation of large language model prompts on user-defined criteria," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–21.

[12] M. Chen, Z. Liu, H. Tao, Y. Hong, D. Lo, X. Xia, and J. Sun, "B4: Towards optimal assessment of plausible code solutions with plausible tests," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1693–1705.

[13] S. Gao, C. Gao, W. Gu, and M. Lyu, “Search-based llms for code optimization,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 254–266.

[14] C. Li, J. Sifakis, Q. Wang, R. Yan, and J. Zhang, “Simulation-based validation for autonomous driving systems,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 842–853.

[15] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, “Codet: Code generation with generated tests,” *arXiv preprint arXiv:2207.10397*, 2022.

[16] A. Jayasena and P. Mishra, “Directed test generation for hardware validation: A survey,” *ACM Computing Surveys*, vol. 56, no. 5, pp. 1–36, 2024.

PLACE PHOTO HERE

Hong Su Hong Su Hong Su received the MS and PhD degrees, in 2006 and 2022, respectively, from Sichuan University, Chengdu, China. He is currently a researcher of Chengdu University of Information Technology Chengdu, China. His research interests include blockchain, cross-chain and smart contract.