

Jarvis: Zero-code Prototyping of IoT Applications with Composable Hardware-Software Abstractions

WEILONG WANG, Southeast University, Nanjing, China

BORUI LI, Southeast University, Nanjing, China

SHUAI WANG, Southeast University, Nanjing, China

TIAN HE, Southeast University, Nanjing, China

The Internet of Things (IoT) has become an integral part of daily life, enabling seamless interaction between humans and the physical world. However, prototyping IoT applications remains an arduous task, requiring expertise in both hardware and software development. Current low-code and zero-code development approaches fail to address the tight coupling between hardware and software as well as the performance of the generated application, limiting their applicability. We introduce Jarvis, a zero-code prototyping framework for IoT applications that leverages composable hardware-software abstractions. By abstracting physical constraints and introducing a parameter-free self-reflection generation mechanism, Jarvis enables large language models to understand and address the coupling constraints in IoT development with minimized token cost and hallucination. Jarvis also considers the cost and energy efficiency of the resulting IoT prototype, optimizing the performance of the generated solutions. Evaluations demonstrate that Jarvis outperforms state-of-the-art methods, reducing hardware costs by 23.1%–75.5%, saving power consumption by 18.7%–95.2%, and lowering token usage for prototyping by 32.1%–86.2%.

CCS Concepts: • **Computer systems organization** → **Embedded and cyber-physical systems**; • **Hardware** → *Sensor devices and platforms*;

Additional Key Words and Phrases: IoT prototyping, zero-code development, large language model

ACM Reference Format:

Weilong Wang, Borui Li, Shuai Wang, and Tian He. 2025. Jarvis: Zero-code Prototyping of IoT Applications with Composable Hardware-Software Abstractions. *ACM Trans. Internet Things* 6, 4, Article 27 (November 2025), 27 pages. <https://doi.org/10.1145/3762670>

1 Introduction

The **Internet of Things (IoT)** penetrates our daily life in various aspects, such as the smart home [12, 23, 33], healthcare [2, 34], smart city [5, 6], and logistics [17, 22]. It now serves as an

This work is partially supported by the Major Project of Fundamental Research on Frontier Leading Technology of Jiangsu Province (Grants No. BK20222006), the National Natural Science Foundation of China (Grants No. 62302096, 62272098, and U24B20152), the Natural Science Foundation of Jiangsu Province (Grant No. BK20230813), and Zhishan Young Scholar Program of Southeast University (Grants No. 2242024RCB0041).

Authors' Contact Information: Weilong Wang, Southeast University, Nanjing, China; e-mail: wang_wl@seu.edu.cn; Borui Li, Southeast University, Nanjing, China; e-mail: libr@seu.edu.cn; Shuai Wang (corresponding author), Southeast University, Nanjing, China; e-mail: shuaiwang@seu.edu.cn; Tian He, Southeast University, Nanjing, China; e-mail: tianhe@seu.edu.cn. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 2577-6207/2025/11-ART27

<https://doi.org/10.1145/3762670>

indispensable bridge between humans and the physical world [1, 12, 31]. Given these diverse usage scenarios, it is essential for developers and entrepreneurs to implement their ideas and gain first-mover advantages by fast prototyping IoT applications.

Prototyping an IoT application requires comprehensive expertise for both hardware components and software development, presenting a formidable challenge for non-experts [3, 38]. Specifically, to build an IoT application prototype, developers are required to select appropriate hardware components, assemble them correctly to form an IoT platform, and then write application code corresponding to the platform [24]. This process is labor-intensive and time-consuming [7, 16], hindering the innovative advancement of IoT applications.

To accelerate the hardware-software integrated IoT application prototyping, researchers from academia and industry propose software-defined low-code development [7, 15, 16, 24, 26]. These approaches facilitate rapid IoT prototyping by providing a series of programming abstractions for developers to express their application requirements and generating the hardware configuration and software binary automatically. Recently, the emergence of **large language models (LLMs)** brings new opportunities to simplify the development of IoT applications. The surprising task-planning and content-generation ability of LLMs facilitates a zero-code paradigm in IoT application development [12, 23, 30, 45]. As the term “zero-code” suggests, developers are no longer required to do extensive programming. Instead, developers only need to specify their application logic with natural language, and these zero-code development frameworks generate the code of the IoT application accordingly. Two unique characteristics in IoT application prototyping are ignored by existing works. (1) *Tight coupling of hardware and software*. Existing research predominantly emphasizes the software aspects of IoT applications, neglecting the critical aspects of selecting and composing the hardware platform. These hardware decisions are equally, if not more, important than code generation in prototyping IoT applications. (2) *Optimal solution rather than feasible*. Existing works concentrate on generating a valid implementation of the application logic, ignoring the real-world performance requirements for the generated IoT prototype, such as hardware cost and energy efficiency.

In this article, we advocate Jarvis, a zero-code prototyping framework for IoT applications that considers both the hardware platform and the corresponding software. With Jarvis, an IoT application developer specifies his/her prototyping idea in plain text. Upon receiving the idea as input, Jarvis automatically selects the appropriate hardware components from a hardware database that fulfill the idea and illustrates the platform composition with the components. Additionally, Jarvis generates the application logic that runs on top of the hardware platform.

Building such a zero-code prototyping framework for IoT applications faces non-trivial challenges. (1) Though LLMs show their premier ability for common-sense reasoning, the complicated physical constraints (e.g., the electrical characteristics, physical pins, interface protocols) and hardware-software coupling are hard for LLMs to understand. Moreover, evidence [21] shows that LLMs inherently possess a statistical lower bound on the hallucination rate regardless of their architecture. How to facilitate LLMs to comprehend the complicated hardware-software constraints, avoiding the inaccurate generation due to hallucination, is the first challenge, we need to overcome. (2) In real-world IoT prototyping, application requirements commonly accompany performance goals. How to generate a prototype IoT application that meets multiple performance goals rather than merely find a feasible solution is another challenge we need to address.

In response to the challenges mentioned above, the key building block of Jarvis is the *composable hardware-software abstraction*, which is a well-structured declarative configuration that records the physical constraints and software abilities of a hardware component. The meaning of “composable” is two-fold. First, it transforms the hardware component and software logic generation problems in IoT prototyping into the composition and manipulation of textual configurations, which are

easily solvable by LLMs. Second, this abstraction also enables the reuse and composition of multiple generated IoT prototypes to create higher-level and complex IoT applications, allowing for further performance optimization between correlated prototypes. Based on the concept of composable abstraction, Jarvis proposes a *parameter-free self-reflection* technique to alleviate the hallucination problem during the hardware-software integrated generation while minimizing the generation cost. Jarvis introduces a *goal-oriented hardware generation paradigm* that simultaneously considers multiple optimization goals during the generation and finds the Pareto-optimal solutions for IoT prototypes. Additionally, Jarvis utilizes *static sensor inference optimization* and *correlation-aware dynamic optimization* to enhance the performance of the generated prototype.

In summary, the contributions of this article are as follows:

- To the best of our knowledge, Jarvis is the first zero-code prototyping framework for IoT applications that addresses the tight integration of hardware and software.
- To facilitate the zero-code prototyping, we propose a composable hardware-software abstraction, which facilitates the hardware-software integrated generation. Based on this abstraction, we design an LLM-based workflow, including a parameter-free self-reflection and goal-oriented hardware generation to generate an optimized prototype of IoT applications.
- We implement Jarvis and evaluate its performance against eight state-of-the-art zero-code generation frameworks for IoT applications. Evaluation results show that Jarvis compared with the current state-of-the-art methods, the generated applications reduce costs by 23.1%–75.5% and save power consumption by 18.7%–95.2%. Moreover, the token cost used for prototyping reduces by 32.1%–86.2%.

The rest of this article is organized as follows. Section 2 presents the related work with Jarvis. Section 3 presents the background and motivation behind Jarvis. Section 4 provides an overview of the framework, outlining its key components and design principles. Section 5 delves into the methodology and technical design of Jarvis in detail. Section 6 details the implementation of Jarvis. Section 7 describes the experimental setup and provides a comprehensive analysis of the performance compared with existing methods. Section 8 discusses open issues of Jarvis. In Section 9, we conclude this article.

2 Related Work

In this section, we review and analyze existing solutions and research efforts for accelerating the prototyping of software-hardware integrated IoT applications.

2.1 Low-Code Rapid Development

Low-code development helps developers complete the most basic development tasks. For example, EdgeProg [27] proposes an edge-centric integrated programming model for IoT applications based on low-code development. This programming model allows users to write processing logic on edge servers, which then automatically generates application code and splits it into device-side and edge-side application logic to achieve optimal latency.

In addition to programming the software logic of IoT applications, developers must also consider the selection and connection of underlying IoT hardware. TinyLink [7] adopts a top-down approach to design the hardware and software of IoT applications. It provides a unified API for IoT applications to interact with underlying hardware components. Developers write applications in a domain-specific language, specifying the key logic of the application without dealing with the details of specific hardware components. TinyLink takes application code as input and automatically generates hardware configurations as well as executable binary programs for the target hardware platform. AutoLink [26] proposes an on-demand generation scheme for IoT hardware platforms

oriented toward multiple performance metrics. Users need to input functional requirements and constraints, and the system automatically completes hardware selection, greatly simplifying the developer's workload. TinyLink 2.0 [16] further integrates cloud-based code development based on TinyLink, incorporating the development features of the cloud computing layer, edge computing layer, and IoT device layer for IoT applications. Its programming language enables developers to express diverse IoT application logic in an easy-to-use manner.

These low-code development methods reduce the barriers and costs of IoT application development. However, their flexibility is still limited because they still require domain-specific knowledge for IoT applications when performing low-code development.

2.2 Zero-Code Rapid Development

Traditional approaches rely on customized interaction commands or well-annotated datasets, which limit their applicability. Recently proposed zero-code development paradigm directly generates executable code using application logic expressed in natural language.

ChatIoT [12] first achieves zero-code generation of **trigger-action programs (TAP)** through large language models. ChatIoT parses user requirements and multi-modal information of the processing scenario, and then uses agents to create and launch service models that are not locally available, generating code for the corresponding IoT services. This approach paves the way to barrier-free application development. Sasha [23] proposes a smart home assistant, parses ambiguous user commands through large language models, and generates corresponding control commands to respond to user requests for smart home operations. Unlike the previous two works, SAGE [31] dynamically constructs an LLM prompt tree to control the sequence of execution actions, deciding what action to take next, whether a particular action is successful, and when to terminate the process. The action set of SAGE enhances the capabilities of LLMs and supports some key requirements of smart home assistants.

Although the above works have achieved zero-code development, they focus on whether the application functions are completed and do not consider power consumption and cost during application development. Moreover, they are based on specific domain languages or tasks and are limited to software-level device invocation, failing to achieve true universality and adaptability.

3 Background and Motivation

In this section, we present the background of prototyping IoT applications and zero-code generation. Then, we show a motivating example of our hardware-software integrated zero-code programming.

3.1 Background

3.1.1 IoT Application Prototyping. An IoT application includes its hardware platform and corresponding software code. IoT hardware platforms are tightly coupled to their applications [8, 18, 35]. This coupling makes it difficult for general-purpose platforms to address application-specific needs. Developers have to know the aspects of an IoT system, including the requirements of the target application as well as the hardware and software components. They typically follow two methods. (1) *Bottom-up approach* [11, 36, 43]. Developers first build an appropriate hardware platform consisting of appropriate microcontrollers and different peripherals and then writes application software that runs on the platform. (2) *Top-down approach* [7, 26, 27]. Developers first write the application code as if there exists a virtual platform that possesses all hardware components, and a development system generates the optimal hardware configuration for the IoT platform as well as the code runs on the platform.

We take the development of a smart home monitoring application as an example. The development requirements might include: automatically opening the door when a person

approaches the room, adjusting the indoor temperature to a comfortable level, automatically closing the door when the person leaves, and turning off indoor appliances if there is no one in the room for an extended period. We decouple the entire development process into the following steps: (a) Analyze the development intent (or functional requirements), for example, detecting when someone approaches/leaves, automatically opening/closing doors, timing the indoor occupancy status, and controlling the status of indoor appliances. (b) Identify the peripherals corresponding to the service functions. For example, a camera is used to detect when someone approaches/leaves, a small motor is used for automatically opening/closing doors, a timer is used for timing the indoor occupancy status, and a remote control for appliances is used to control the status of indoor appliances, etc. (c) Express the requirements using machine language (code). For example, the code corresponding to the requirements is compiled into languages such as C or Python to implement the functional requirements. (d) Burn the compiled code onto a development board or service device. For example, these requirements are compiled and burned into a TelosB. The code controls the peripherals to ultimately achieve the development requirements.

3.1.2 Zero-Code Code Generation. LLM-powered autonomous agents are designed to tackle complex and diverse tasks, often by breaking them down into multiple stages or subtasks. Various agent architecture designs have been proposed in the literature [37]. For instance, **Chain-of-Thought (CoT)** [40] is a prominent prompting technique that enables agents to perform complex reasoning through sequential planning and acting. In CoT implementations, multiple CoT demonstrations are embedded in the prompt to guide the agent’s reasoning. Alternatively, zero-shot CoT [25] showcases the reasoning capabilities of LLMs by simply adding the phrase “think step by step” to the prompt. Another approach extends CoT by adopting a tree-like reasoning structure, where each intermediate step has its own set of sub-steps [41]. These methods do not incorporate feedback from the environment, i.e., results of actions already taken, into the planning process. Subsequent work addresses this limitation; for example, ReAct [42] integrates observations from the environment, e.g., results of API calls or tool usage, received after taking an action. These observations are then considered in subsequent reasoning steps. Human feedback also helps refine the agent’s plan by providing additional details or preferences.

Another crucial aspect of agent design is the use of external tools for action execution, allowing agents to extend beyond their internal knowledge. APIs are the most common type of tool, with LLMs like Gorilla [28] and ToolLLM [29] specifically trained for API usage. In addition to APIs, external knowledge bases, e.g., document databases, are leveraged as tools to acquire specific information or expert knowledge [13].

3.2 Motivating Experiment

Through our investigation of existing works, we find that existing zero-code development efforts fall flat in the following two scenarios.

First, existing works primarily focus on software-level generation [12, 23, 31] and do not address the *highly coupled nature of software and hardware in IoT*. A practical IoT application development platform must pay attention to this coupling issue to achieve coordinated operation between software and hardware [7, 16]. Second, existing zero-code development primarily focuses on functional metrics and overlooks the performance side, such as power consumption and cost, of IoT applications. In real-world IoT application development, the performance metrics are critical indicators. Hence, this article focuses on not only generating valid but also performance-optimized applications. This is an even more challenging goal than existing zero-code IoT development methods.

We use a simple multi-device coordination program as an example: “When a person is detected in the room and the indoor brightness is insufficient, turn on the light.” We develop applications using

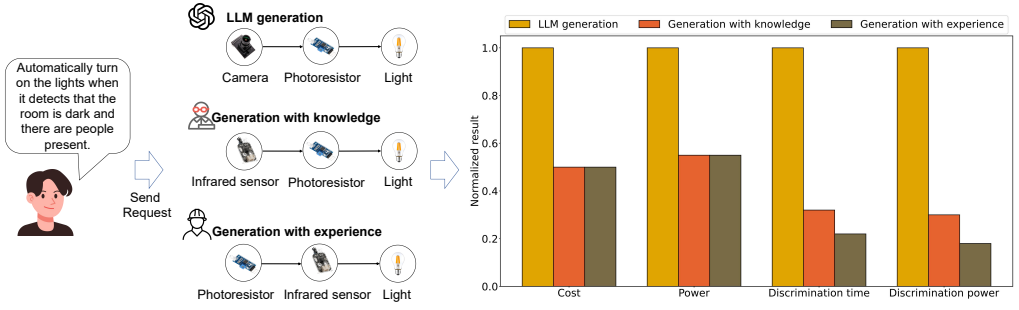


Fig. 1. Different approaches to IoT application development.

the three most common methods in the real world: (1) *LLM generation*: generating IoT applications using an LLM, (2) *Generation with knowledge*: manually generating IoT applications based on domain knowledge as a reference, (3) *Generation with expertise*: manually generating IoT applications by developers with extensive experience. We compare and analyze the power consumption, cost, trigger discrimination time and power for applications generated by the three different approaches. The trigger discrimination time and power refer to the minimum time and power required to determine whether to trigger an “Action”. The experimental results are normalized based on the results obtained using the LLM.

As shown in Figure 1, we find that the power consumption and cost of IoT applications generated by LLMs are always the highest. This is because LLM-based applications primarily aim at fulfilling functional requirements without considering the cost and power consumption of the application. Additionally, with a vast device space, LLMs struggle to effectively filter and select the optimal configuration. Both (2) and (3) are able to generate applications with relatively lower power consumption and cost. They gradually reduce the device search space based on their execution capabilities and functional requirements, ultimately finding the optimal configuration. From the left side of Figure 1, it is seen that although (2) and (3) select the same device configuration, the order of device invocation is different. For multiple devices with no data dependency on each other, (3) adjusts the invocation priority of different devices based on lower power consumption and latency. Therefore, the applications developed by (3) have lower trigger discrimination power consumption and time.

In summary, it is necessary to design a zero-code generation paradigm that facilitates such development expertise and ensures that this expertise is as accurate and comprehensive as possible. This paradigm enables the development of accurate and low-power IoT applications.

4 System Overview

In this section, we introduce the overall architecture of our work, as shown in Figure 2. It includes the application generation phase and the application optimization phase.

4.1 Generation Phase

The first goal of our work is to ensure the high accuracy of the developed applications. Considering the highly complex problem of software-hardware integrated IoT applications, we define a new software-hardware integrated abstraction by combining existing software [10] and hardware [7] abstraction concepts (Section 5.1). Based on the software and hardware information, we construct a software-hardware correlation graph to address the coupling constraints between software and hardware during the development process. Taking into account the computational token

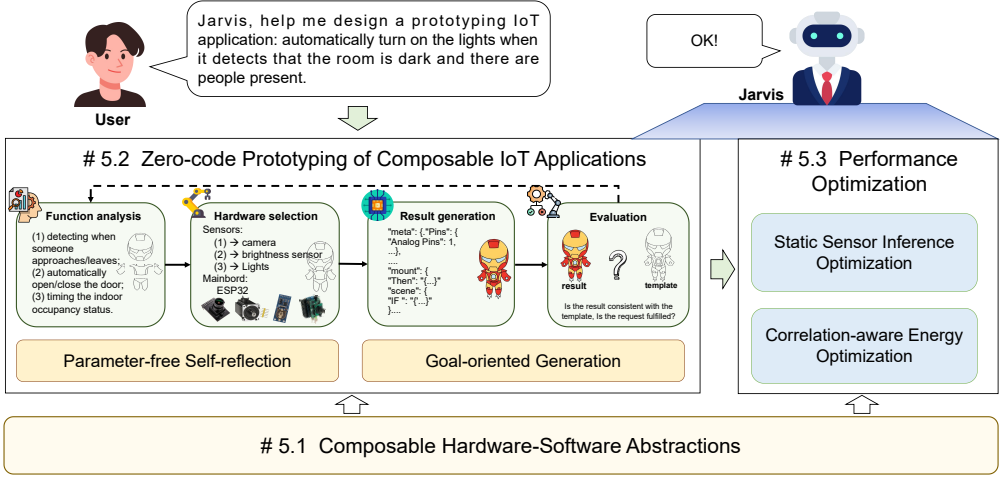


Fig. 2. The system overview of Jarvis.

cost incurred during application development, we decompose the entire application into multiple modules, each equipped with a dedicated LLM agent to handle the corresponding problems (Section 5.2). Given the strong coupling between software design and hardware selection in integrated software-hardware development scenarios, we design the modules to operate in a collaborative mode, ensuring consistency between software and hardware development. This approach enables the agent to capture functional requirements while reducing the cost and improving the accuracy of application development. For example, for the requirement of “detecting the presence of a person,” we filter device information based on the result of “function analysis” from Figure 2. This not only reduces the token cost for subsequent agent operations but also minimizes information redundancy.

Secondly, still targeting our primary objective, we address the accuracy of each module through a dual-adaptive reflective algorithm. The first layer enhances the accuracy of results through multiple rounds of internal self-reflection, while the second layer further improves accuracy by leveraging external result consistency. The adaptive algorithm determines the optimal number of reflections to prevent problems such as result hallucinations [9, 20] and high development costs associated with excessive reflection iterations (Section 5.2).

4.2 Optimization Phase

Targeting our second objective of reducing the power consumption and cost of the developed applications. We map the device selection problem to a multi-objective optimization problem and construct the static sensor inference optimization module to find the optimal solution in terms of power consumption and cost through a multi-objective optimization algorithm. As mentioned in the motivation section, the execution order of devices affects the overall peak power consumption and time of the application. Therefore, we abstract the device execution logic into a search tree using the idea of search tree pruning. By rapidly pruning the optimal sub-nodes, we ultimately achieve the best execution order for application device invocation and implement static application optimization (Section 5.3).

Considering the logical rules and other information present in IoT scenarios, we design the correlation-aware energy optimization module to implement both periodic and real-time execution of the generated IoT applications. We construct a logical cache of information for the module, which includes information obtained from various devices over a period of time, as well as the logical

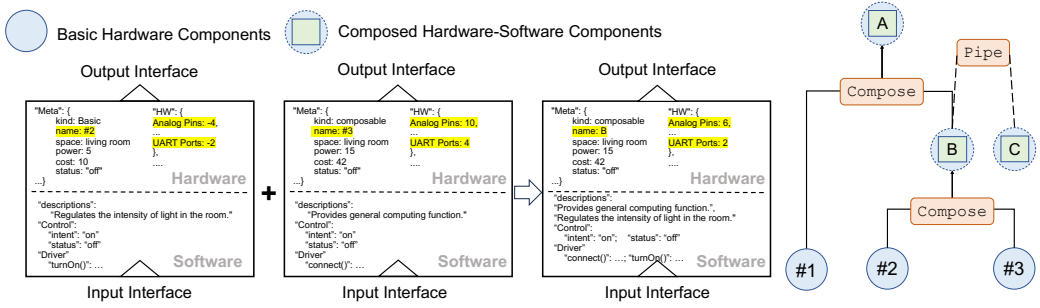


Fig. 3. The composable hardware-software abstraction in Jarvis.

relationships between different pieces of information (Section 5.3). By reusing data information from the same device across different applications, the cost of data acquisition is reduced. Based on data mining algorithms, potential correlations between different applications are identified. These relationships are then abstracted into new applications and dynamically updated into the triggering rules of relevant applications, thereby optimizing overall power consumption and latency. For example, to determine if there is anyone indoors, one immediately calls upon the camera to collect data for identification, or uses the cached data collected by the camera for identification, or even uses data collected by other devices for logical reasoning to make the determination.

5 Main Design

Jarvis aims to enable users to directly generate low-cost and low-power IoT applications utilizing natural language interfaces, rather than requiring traditional coding or the selection of hardware configurations. In this section, we present how Jarvis enables users to directly generate IoT applications in a token-efficient manner. We first describe the overall workflow of IoT applications generation, including the inputs and outputs of Jarvis. Then, we dive into the details of the context-aware prompting.

5.1 Composable Hardware-Software Abstractions

Inspired by [7, 10], we recognize that software or hardware abstraction can accelerate IoT application prototyping. However, these approaches primarily rely on domain expertise and rule-based construction of formatted abstractions, making them unsuitable for no-code application generation scenarios. This limitation arises because large language models naturally handle textual input and output operations but struggle with formatted computational tasks. To address this, Jarvis proposes a composable hardware-software abstraction method that transforms component selection and configuration into textual operations, enabling accurate IoT prototype generation. The specific design and detailed description are as follows.

5.1.1 Hardware-Software Integrated Abstractions. Inspired by TinyLink [7] and dSpace [10], to facilitate LLMs with the ability to generate IoT prototypes, the basic abstraction structure of each hardware component is shown on the left-hand side of Figure 3. In each abstraction, Jarvis records both the metadata, physical hardware characteristics (e.g., analog pins in the figure). Through these abstractions, LLM agents achieve a precise understanding of the physical space. The descriptions of these two abstractions are shown in Figure 3. Their specific definitions and descriptions are as follows.

(1) Basic hardware components. These are traditional peripherals that are typically standalone devices in their own right. They are combined with other devices to form composable

hardware-software abstractions. As shown in Figure 3, this abstraction includes its software-level functionality and also contains the physical hardware description of the abstraction, such as the number of interfaces.

(2) Composed hardware-software components. These are collections of devices that usually perform tasks that cannot be accomplished by a single device alone. This combination of devices is considered a special type of device, as it shares similarities with traditional peripherals, such as the ability to connect with other new devices and the possession of hardware physical information and software functionality. As shown in Figure 3, it is a new abstraction composed of other abstractions. It is combined with other abstractions to form new ones. The resulting abstraction can both reuse the functionalities of its components and develop new functionalities based on the existing ones.

5.1.2 Composing Abstractions. By “composable”, we indicate the abstraction of different hardware is able to be composed together. For example, in Figure 3, basic components #2 and #3 are composed together to form a new hardware platform, B. Considering the characteristics of the above abstractions and real physical devices, we comprehensively examine whether these abstractions are combined, how they are combined, and how the relationships after combination are represented. We also introduce two declarative commands, as follows, to express the relationship and data flow between components.

(1) `Compose(#2, #3)`: Our abstraction is centered around the concept of `Compose`. Consistent with the real physical space, we determine whether devices are composed based on the compatibility of their physical constraints. To help the LLM better understand this compatibility, we compare the interface types and quantities from the previous abstractions, thereby judging whether devices are composed. Once two or more components are composed, their abstractions are also textually composed together, as shown in Figure 3, which facilitates the efficient hardware composition with LLMs.

(2) `Pipe(B, C)`: IoT applications typically consist of triggers and actuators, which are essentially two abstractions in the digital space. If the behavior of two triggers or actuators is correlated, e.g., when the light is on in B, the light will not be lit in C. At this time, we pipe the two triggers of B and C to attach the two results for further optimization.

It is worth noting that this composable hardware-software abstraction for each hardware component is easy to obtain. The software part of the abstraction is found in open-source smart home projects such as Home Assistant. The hardware part is obtained with the help of a hardware database or directly analyzed from the datasheet of the hardware component [19].

5.2 Zero-Code Prototyping of Composable IoT Applications

This section describes how to generate efficient and accurate composable hardware-software abstractions or an IoT application prototype based on user requirements. To ensure Jarvis can precisely generate these abstractions and application prototypes that meet user needs, we define an overall workflow of zero-code prototyping. Considering the unique power consumption and construction cost challenges in IoT scenarios, along with personalized user requirements, we design an efficient generation paradigm with a parameter-free self-reflection method and a goal-oriented hardware configuration generation method. The specific descriptions and detailed designs are presented below.

5.2.1 Overall Workflow of Zero-Code Prototyping. As shown in Figure 4, the overall workflow of IoT applications generation mainly includes the input, the output, and the intermediate processes. It includes four modules: functional analysis module, hardware selection module, result generation module, and result evaluation module.

Input: The input to Jarvis includes three parts, (1) the IoT applications generation request using natural language from the user; (2) additional information (i.e., the context of device) about the

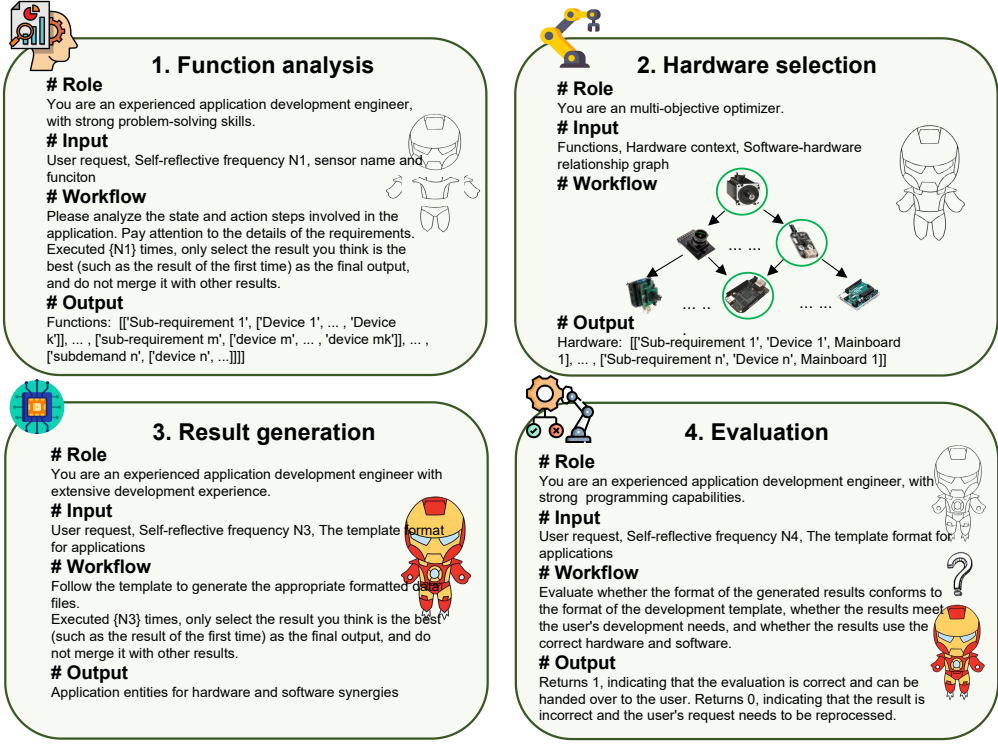


Fig. 4. The workflow of IoT applications generation of Jarvis.

peripherals and mainboards in the IoT scenarios, as well as the correlation between devices; (3) the hardware and software entity information template for the application. The prompting device context is mainly device metadata, including device name, type, power consumption, input interface type and number, output type and number, and cost description as shown on steps 2-4 of Figure 4. For example, {"Device Name": "Camera Module (OV7670)", "type": "peripheral", "Power Consumption": 0.5, "Analog Pins": 0, "UART Pins": 0, "Analog Ports": 0, "UART Ports": 1, "Cost (USD)": 15, "Description": "Provides visual capture for image processing."}.

Output: Jarvis generates a full-process demand design consistent with user requests. In many IoT application frameworks, it is common to implement application logic in the format of TAP with "IF <triggers> WHILE <conditions> THEN <actions>" [7, 12, 16, 26]. This format stipulates that upon the occurrence of triggers and the satisfaction of conditions, a sequence of actions is executed. When user requests are overly complex, this format becomes too redundant. To address this, we simplify the aforementioned format. Instead of using the traditional format, Jarvis generates the full-process application service in a list format. Jarvis analyzes the required functions and peripherals based on user descriptions, and constructs the functions into propositions for generating triggers, conditions, and actions. Finally, it deploys the peripherals used in the entire logic to the mainboards. It selects the mainboard and generates IoT code. We define the newly developed application as a new entity of hardware, which also has the capability to be combined with other devices. Our final output includes the hardware (peripherals and mainboards) entity combination scheme, the software control files, and the description of the new application entity as a combinable device.

For example, when a user states, *Automatically turn on the lights when it detects that the room is dark and there are people present.*, the final output is step 3 or 4, as shown in Figure 4. The result of

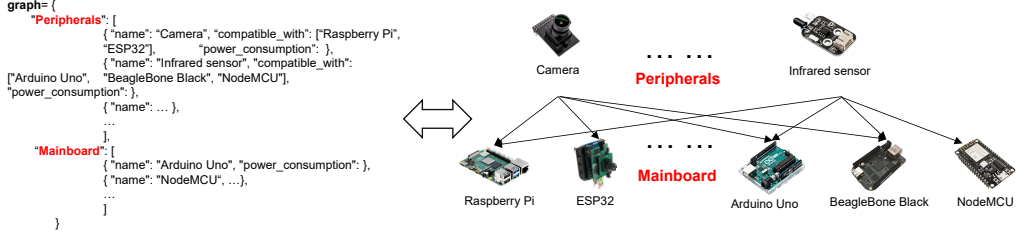


Fig. 5. The software and hardware relationship graph of Jarvis.

step 2 includes the full-process demand list L : [“Detect dark environment”, “BH1750 Light Sensor”, “STM32 Nucleo”], [“Detect presence of people”, “HC-SR501 Motion Sensor”, “STM32 Nucleo”], [“Turn on lights”, “Light (Lamp)”, “STM32 Nucleo”]. The final result is shown in step 3 or 4. We present the physical device information of the combined application (hardware) and its internal execution logic (software) in JSON format. It serves as a standalone device added to the universal device list, providing services for subsequent application requirements. The software generation method is compatible with current low-code development tools like [7, 26, 27].

Context-aware prompting in Jarvis: One major problem with zero-shot prompting is that the initial user request may not include enough information to generate an IoT application [12]. Therefore, in practice, we inevitably incorporate user feedback. In Jarvis, we divide the IoT application generation task into four distinct subtasks: analyzing the application’s functional request, selecting peripherals based on that request, configuring hardware compilation for those peripherals, and generating the final application code. We design four corresponding dialog agent modules: a functional analysis module, a hardware selection module, a result generation module, and a result evaluation module. This subdivision of tasks directly reflects how humans conceptualize IoT application generation from user requests. These four agents connect with each other and generate low-power and low-cost IoT applications accurately while using tokens efficiently. To address the dual challenges of ensuring generation accuracy and optimizing system parameters, we developed an efficient generation framework with parameter-free self-reflection. This method enables collaborative operation among the four agents, which precisely generates low-power and cost-effective IoT applications while maintaining high token efficiency.

For each module agent, we create the system messages shown in Figure 4. These messages instruct the LLM to act as an intelligent assistant in the IoT domain and help users complete their tasks. We define a specific mission for each agent and outline the basic steps needed to fulfill it, emphasizing a step-by-step solution approach. Through this iterative reasoning, we design a series of operations that allow the agents to interact with users.

Unlike traditional application development, our work involves implementing integrated hardware and software IoT development. To accommodate diverse development needs across different users, we design a goal-oriented hardware configuration generation method. We complete the development based on the hardware and software information of the devices, as well as the correlation information between them. As shown in Figure 5, we utilize Tinylink [7] and dSpace [10] to construct the correlation graph G between hardware and software.

5.2.2 Efficient Generation with Parameter-Free Self-Reflection. At present, there are many logical architectures in AI agent work [4, 39–42, 44]. For example, CoT [40] reasoning facilitates the decomposition of complex, multi-step problems into constituent cognitive processes, enabling the dynamic allocation of computational resources to more intricate reasoning tasks. In addition, there are some improvement schemes for CoT to enhance the performance of CoT, such as CoT-SC [39],

TOT [41], GOT [4], ReAct [42]. This method effectively improves the accuracy of solving complex problems.

In Jarvis, we adopt COT-SC [39] as the underlying solving framework. This method has better computational performance compared with other COT-related algorithms and lower computing and interaction costs compared with ReAct-like algorithms.

The accuracy of COT-SC is related to the number of its self-consistency checks; the higher the number, the more accurate the results, but the greater the required token cost. To address this, we design an efficient generation with a parameter-free self-reflection mechanism called EP-SR. It consists of two stages: internal self-reflection and external self-reflection.

Firstly, we establish an internal self-reflection method. It abstracts the self-consistency reflection of the COT-SC process to a higher-level thinking without altering its input and output token costs. The Agent completes the self-consistency result selection during the problem-solving process, rather than relying on an external algorithm to choose the final result. For a given application requirement Q , the Agent thinks multiple times within the solution space, which we define as R_{think} :

$$(S_i^1, S_i^2, \dots, S_i^n, R_i)_{i=1}^k = R_{think}(Q), \quad (1)$$

where k is the total number of thinking iterations. The final answer R is chosen based on the most consistent R_i from self-reflection, $R = R_{self}(R_{i=1}^k)$.

The performance of the internal self-reflection is limited by the number of self-consistency iterations. We find an interesting phenomenon: the performance of internal self-reflection is not entirely consistent with the number of self-reflection iterations. This phenomenon is similar to human overthinking and follows a Poisson distribution. We find that for agents in different modules, the same number of self-reflection iterations results in significant performance differences. This phenomenon is related to information transfer.

Therefore, the accuracy of the self-reflection count k is critically important. However, testing and validating the optimal value of k for each individual problem incurs substantial computational costs. Given the dynamic nature and complexity of these problems, determining the optimal k through exhaustive testing and validation is impractical. To address this, we have developed an algorithm that automatically determines the optimal k without requiring any testing or validation. This approach enables efficient generation through parameter-free self-reflection.

Since the core of our work is multi-agent collaboration, based on information transfer theory, we know that there is information attenuation during the information transfer process. The performance of each module's agent is related to the number of functions included in the user's task and the number of self-consistency iterations, following the Poisson distribution rule. This article designs a self-reflection iteration k selection algorithm without training and parameters, combining information transfer theory and the Poisson distribution rule. The calculation formula is shown as follows:

$$k = \text{round}\left(a + \sin\left(\frac{\pi}{2} e^{-a} \frac{a^\beta}{\beta!} \beta\right)\right), \quad (2)$$

where $\text{round}(\cdot)$ represents the rounding function. a represents the number of Agents, which defaults to 4. β represents the total number of task decompositions, which defaults to 4.

Secondly, we adopt COT-SC as our external thinking method, and execute external self-reflection. As we mentioned earlier, the accuracy of application generation depends on the complexity of the application requirements. Considering that there is a tradeoff between the number of candidate results generated by COT-SC and the token cost. The accuracy of application generation typically relies on the number of candidate results. We design an adaptive COT-SC candidate generation algorithm for EP-SR. As shown in Algorithm 1, the inputs are the application requirements Q

ALGORITHM 1: Candidate Result Generation Algorithm for EP-SR

Input: Application requirements requirement Q , the number c of functions of Q .
Output: The final IoT application r' .

- 1 Initialize an empty set of outcome candidates $R = []$.
- 2 Initialize a probabilistic basis for generating a new candidate result $P = c$.
- 3 **while** $Ture$ **do**
- 4 $r \leftarrow COT(Q)$
- 5 $R.append(r)$
- 6 $p \leftarrow \min\{P, 1\}$
- 7 Generate a random number $m, m \in [0, 1]$.
- 8 **if** $m \leq p$ **then**
- 9 $P \leftarrow \frac{P}{2}$
- 10 **else**
- 11 break
- 12 $r' \leftarrow$ the result with the highest number in R .
- 13 **return** The final IoT application r' .

and the number c of functions of Q from the function analysis module of the framework. The output is the final IoT application r' . Steps 1 and 2 are for initializing the candidate result set and determining the probability of generating the next candidate result. Step 4 involves executing COT [40] to generate candidate results. Steps 6–11 are for deciding whether to continue generating new candidate results and obtaining the final candidate result set based on the decision. Steps 7–8 involve selecting the most frequently occurring candidate result from the candidate result set R as the final result r' and returning it.

5.2.3 Goal-Oriented Hardware Configuration Generation. In this part, our primary objective is to further reduce the cost and power consumption of the application while ensuring accurate application generation. To achieve this goal, we propose a goal-oriented hardware configuration generation mechanism, called GHCG. On the one hand, we employ a multi-objective optimization algorithm to find the optimal application configuration from a static selection perspective. On the other hand, we dynamically adjust the application configuration based on the impact of device invocation on power consumption and results during the actual execution of the application. This work is located in the hardware selection section of the framework.

Problem Definition. The problem we address in this part is to select the optimal solution in terms of power consumption and cost from candidate peripherals and mainboards. This is inherently a multi-objective optimization problem. We define the problem formulation as follows:

$$\text{Find the values of } d_i \ (\forall i \in S) \text{ and } d'_i \ (\forall i \in M) \quad (3a)$$

$$\arg \min \left(\sum_{i=1}^k d_i d'_i (x_i, y_i) + d_i d'_i (x'_i, y'_i) \right) \quad (3b)$$

$$\text{s.t.} \quad \text{Requirement Constraints} \quad (3c)$$

$$\text{Hardware Constraints} \quad (3d)$$

where d_i represents the result of selecting this peripheral. If this device is selected, d_i is 1, otherwise, it is 0. S represents all optional peripherals. M represents all optional mainboards. L represents the output of the Functional Analysis Module, which contains the optional devices related to the user requirements. k represents the number of functions required by the user. x_i and y_i denote the power

consumption and cost of the peripheral d_i , respectively. x'_i and y'_i denote the power consumption and cost of the development board d'_i , respectively.

Requirement Constraints. The selected device must be able to satisfy the user's requirement, i.e., $d_i \in L$, L denotes the set of required functions and peripherals obtained from the functional analysis module.

Hardware Constraints. We map the interface relationships between devices to hardware constraints based on the device definitions in TinyLink [7]. i.e., $\langle d_i, d'_i \rangle \in G$. G represents the association relationship between the sensor or controller and the development board. We use the four generic types of interfaces (Analog Pin, UART Pin, Analog Port, UART Port) as constraints:

$$\text{AnalogPins}(d'_i) \geq \text{AnalogPins}(d_i) \quad (4a)$$

$$\text{UARTPins}(d'_i) \geq \text{UARTPins}(d_i) \quad (4b)$$

$$\text{AnalogPorts}(d'_i) \geq \text{AnalogPorts}(d_i) \quad (4c)$$

$$\text{UARTPorts}(d'_i) \geq \text{UARTPorts}(d_i) \quad (4d)$$

$$\text{AnalogPins}(d'_i) \geq \sum_{d_i \in S, G(d_i)=d'_i} \text{AnalogPins}(d_i) \quad (4e)$$

$$\text{UARTPins}(d'_i) \geq \sum_{d_i \in S, G(d_i)=d'_i} \text{UARTPins}(d_i) \quad (4f)$$

$$\text{AnalogPorts}(d'_i) \geq \sum_{d_i \in S, G(d_i)=d'_i} \text{AnalogPorts}(d_i) \quad (4g)$$

$$\text{UARTPorts}(d'_i) \geq \sum_{d_i \in S, G(d_i)=d'_i} \text{UARTPorts}(d_i) \quad (4h)$$

$$\forall d_i \in S, \forall d'_i \in M, \forall \langle d_i, d'_i \rangle \in G \quad (4i)$$

Goal-driven exploration of optimal application configuration. The Pareto solution set P_S is obtained in the above way, and users choose to use the application configuration scheme according to their needs. We provide an optimal solution-solving scheme. We take the current Pareto solution set as a new solution space and use ideal points and Euclidean distance to find the optimal solution. We first need to define the ideal point and then compute the Euclidean distance between each solution and the ideal point. The ideal point I is defined as the optimal value of the objective functions: $I = (X^*, Y^*)$. X^* is the minimum possible value that minimizes the total power consumption. Y^* is the minimum possible value that minimizes the total cost.

For each solution o in the set of Pareto optimal solutions, compute its Euclidean distance $D(o, I)$ from the ideal point I by the following formula:

$$D(o, I) = \sqrt{(X_o - X^*)^2 + (Y_o - Y^*)^2}, \quad (5)$$

where X_o represents the total power consumption of solution o and Y_o represents the total cost of solution o . The optimal solution o^* is the solution that maximizes the Euclidean distance from the ideal point I in the set of Pareto optimal solutions: $s^* = \arg \min_{s \in P_S} D(o, I)$. This means that the optimal solution o^* is as close as possible to the ideal point in terms of total cost and total power consumption, while still satisfying constraints.

5.3 Performance Optimization

Unlike the previous sections, which primarily focus on application generation, this part mainly discusses how to manage and optimize the generated applications. This mechanism consists of

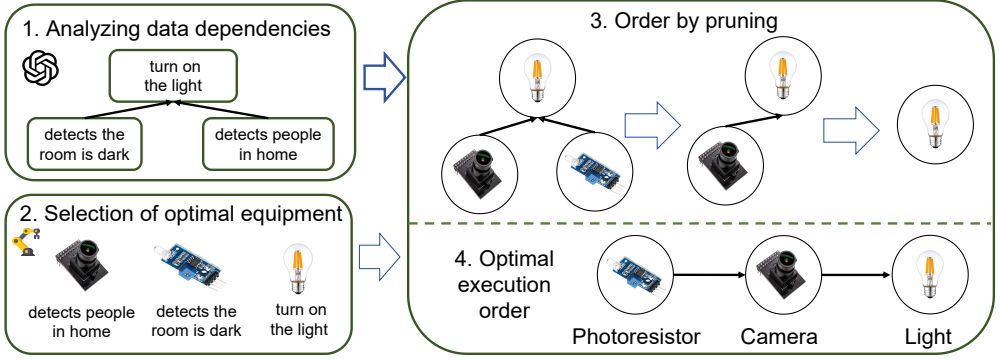


Fig. 6. Multi-stage device sequencing strategy.

two main parts: static sensor inference optimization and correlation-aware energy optimization. The static sensor inference optimization section focuses on optimizing the composable hardware-software abstractions within application prototypes. The correlation-aware energy optimization section analyzes functional correlations between different application prototypes or composable hardware-software abstractions to enable direct functional inference without runtime execution, thereby optimizing sensing power consumption. The specific designs are as follows.

5.3.1 Static Sensor Inference Optimization. Based on the motivation discussed earlier, we know that the execution order of devices affects the minimum triggering power consumption and time of the application. For example, the requirement is: *When a person is detected in the room and the light intensity is insufficient, automatically adjust the indoor brightness.* The final result is: *detects people in the home (camera), detects that the room is dark (photoresistor), and turns on the light (light).* With power consumption as the priority, there are two device invocation or data collection routes: (1) Camera → Photoresistor → Light; (2) Photoresistor → Camera → Light. There are significant data dependencies between devices. For example, in this requirement, the “light” must be placed after both the “camera” and the “photoresistor.” Although the LLM Agent effectively extracts dependencies between data, it is unable to generate an optimal device invocation plan using this strategy. Instead, it typically generates the two strategies mentioned above (1) and (2) randomly. The light is turned on if both the camera detects a person (i.e., Camera == True) and the photoresistor detects insufficient light (i.e., Photoresistor == True). When there is no person in the room or the indoor brightness is sufficient, the optimal execution strategy should be to invoke the photoresistor and then terminate directly. This is because the power consumption of the photoresistor is lower than that of the camera.

To identify such an optimal execution order, we design a multi-stage device sequencing strategy, as shown in Figure 6. Stage 1, the LLM Agent extracts data dependencies. Stage 2, based on the results from Stage 1, a data dependency tree is constructed, where the final triggering condition (e.g., turning on the light) serves as the tree node. The tree nodes are then populated with device information from the hardware selection module of Jarvis, with each node containing the device name and its power consumption. In stage 3, rapid pruning is performed, where the current smallest leaf node is extracted and added to the device invocation sequence list each time. Iteration of Stage: this process is repeated until the tree is empty, and the device invocation sequence list contains devices from the hardware selection module of Jarvis.

5.3.2 Correlation-Aware Dynamic Energy Optimization. Unlike the static sensor inference optimization discussed earlier, in this section, we primarily focus on the correlation between different

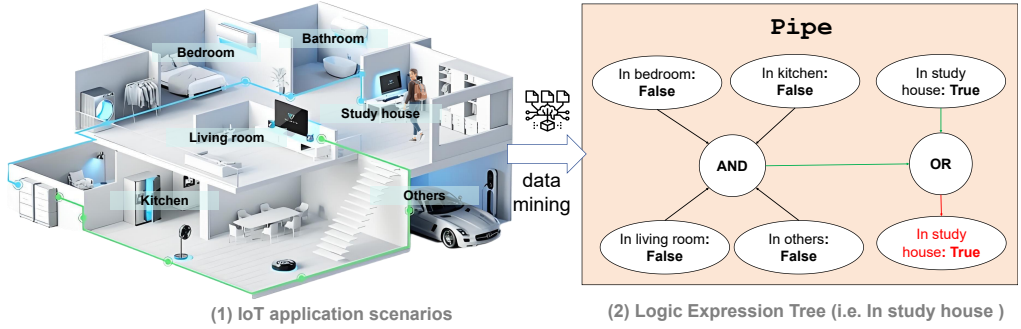


Fig. 7. IoT information logic expression tree of Jarvis (i.e., in study house).

applications and devices. It refers to the acquisition and construction of pipes in composable hardware software abstractions.

In IoT scenarios, there are many mutually exclusive logics and association rules. For instance, at a given moment, an entity (i.e., a person or a device) is in one specific location and cannot be in multiple locations at the same time. A device or system is in one particular state at a time and cannot be in multiple states simultaneously. We assume that within a time interval R , devices and information states are at the same moment. We define a data cache area C , where the application requirements share data information. For IoT application requests, they are uncompleted not only through peripherals but also by directly using the corresponding data in C within S time. When C does not have completely corresponding data, the context rules learned by the rule miner infer the required results from other data in C . The inference rules are shown in Figure 7.

As shown in Figure 7, a logic expression tree for a tuple $\langle r = v \rangle$ is a tree representation of the Boolean expression that implies the value v for the attribute r . r represents the requirement of the service, which the application generated earlier based on user requirements. v represents the result or status of the requirement. For example, Figure 7 shows a logic expression tree for the tuple $Studyhouse = True$. It is generated by combining various rules that imply $\langle r = v \rangle$ directly, or indirectly via transitive relationships with other rules. A logic expression tree is a Boolean AND-OR tree with the following properties: (1) A non-leaf node represents an AND or OR operation on the values of its child nodes; (2) a leaf node represents a tuple. An AND node evaluates to True if all of its child nodes recursively evaluate to True, while an OR node evaluates to True if any of its child nodes recursively evaluate to True. The logic expression tree for the tuple $\langle r = v \rangle$ is designed such that it evaluates to True if and only if the tuple holds according to the context rules.

We construct the logic execution tree by integrating both temporal and spatial information. We consider the triggering states v of different applications r under various time windows W and build the first layer of logical rules using association rule mining methods. For example, for the application of monitoring people, it is frequently triggered during the day but rarely at night. Considering the exclusivity of spatial information, we construct the second layer of logical rules based on the triggering states r of different applications in different areas within the same time window. For example, at 7 p.m., if everyone is in the living room, then the bedroom is definitely unoccupied. By integrating the above rules based on both temporal and spatial information, we build the logic execution tree to reduce unnecessary device invocation overhead during the execution phase.

After mining these rules, we automatically generate a new application or trigger condition C_a and add or update this condition to the associated applications. Like other devices, it also has the capability to be combined. This operation, in practice, is realized with the pipe primitive of our composable abstraction. For example, the new application we mine is C_a "Person's Location". For the application,

“Turn on the light automatically when there is a person in the room and the space is dark,” its rule `IF camera == True and Photoresistor == True THEN Light = True` is automatically modified to `IF Ca == True and camera == True and Photoresistor == True THEN Light = True`. Each time the application is executed, it directly derives the result using the cached data or logical information of C_a, thereby effectively reducing execution power consumption and latency.

6 Implementation

In this section, we provide a detailed introduction to the specific implementation plan of Jarvis, with the following implementation details.

Programming Language. We specify the programming language used in Jarvis in the two parts: the language used in implementing the Jarvis framework, and the language used in the generated code by Jarvis. The Jarvis framework is implemented in Python. The generated IoT application of Jarvis includes two parts: the composable abstraction of each generated IoT platform and the interoperation logic between multiple devices. Specifically, the composable abstraction of each generated device describes the hardware composition and software ability of an IoT platform, as shown in Figure 3 in our manuscript. Within the composable abstraction of a generated device, the code implementation of its software ability is generated in TinyLink domain-specific language [7], which is a state-of-the-art platform-agnostic IoT programming language. The interoperation logic between multiple devices is expressed as a trigger-action program, which is similar to existing zero-code generation frameworks such as ChatIoT [12].

System Architecture. Jarvis includes a generation and an optimization subsystem. The generation subsystem receives the user input, maintains the hardware database, and generates the hardware and software configuration. According to existing benchmarks of LLMs such as BigCodeBench [46] and our empirical study, we observe that the closed-source LLMs such as GPT-4o and Claude Sonnet exhibit good reasoning and code generation ability. Hence, our current implementation of the generation subsystem operates in a local-remote hybrid approach. A local part receives the generation request, maintains the database, and the reasoning and generation mainly use these LLM services via remote APIs. It is worth noting that Jarvis is also adaptive to local LLM deployment, such as the Qwen series. The optimization subsystem mainly operates in both offline and online phases. During offline generation, it optimizes sensor inference. After the IoT hardware platform is generated, it performs energy optimization by deciding how a request is processed by the IoT devices.

Deployment Environment. Jarvis is deployed on a consumer-level server equipped with an i9-13900H×20 processor, 16 GiB RAM, and Ubuntu 22.04 LTS. We use GPT-3.5-turbo and GPT-4o-mini for the generation of the IoT application. To ensure the stability and reproducibility of generated results, we set the temperature parameter to 0 and the maximum token length to 512.

IoT Hardware Component. We use a real-world smart home scenario in our current implementation and evaluations. It contains the following room types: living room, study room, bathroom, bedroom, kitchen, and others. In each room, there are various IoT development boards and peripherals for Jarvis to select. Specifically, the available development boards for each room include: Raspberry Pi 4, Raspberry Pi Zero, Arduino Uno, Arduino Mega 2560, Arduino Nano, ESP32 DevKitC, ESP8266 NodeMCU, STM32 Nucleo, BeagleBone Black, and BeagleBoard. Peripherals include: camera, motion sensor, microphone, temperature and humidity sensor, light, infrared sensor, light sensor, small motor, and sound sensor.

7 Performance Evaluation

In this evaluation section, we first introduce the setup, including the datasets, baselines, and evaluation metrics. We evaluate the performance of Jarvis through a comparative study in terms of multiple metrics.

7.1 Evaluation Setup

In this section, we provide a detailed description of the selected benchmark, baseline methods, and performance indicators.

(1) Benchmark. Similar to existing zero-code generation platforms [12, 23], we use the natural language description of TAP as the input requests. Due to the unavailability of publicly accessible datasets specifically designed for user requests on IoT applications, we generate a set of IoT application development requests with the assistance of the GPT-4 model. To better accommodate real-world daily usage, we extend the evaluation benchmarks that mainly consists of one-step TAP descriptions with various multi-step TAP descriptions that include more complex and realistic tasks. For one-step and multi-step TAP descriptions, we further classify them into *easy*, *medium*, and *hard* tasks according to their generation difficulties, i.e., the number of triggers and actions.

(2) Baseline. To evaluate the performance improvement of our design, we compare Jarvis with 6 types of baselines.

- COT [40]: Based on GPT, it requires the LLM to think step-by-step within the system messages.
- COT-SC [39]: Building on COT, it generates multiple results and then obtains the optimal result through a result similarity mechanism. The default number of self-consistent candidate results is set to 3.
- ReAct [42]: Based on GPT, it allows the LLM to interact with users and the system through actions during the reasoning process.
- SAGE [31]: An agent architecture based on ReAct and a large number of external tools, using multi-agent collaboration to complete tasks.
- Sasha [23]: An architecture based on LLM for user intent parsing, and application generation through multi-agent interaction.
- ChatIoT [12]: A multi-agent collaboration architecture based on ReAct, with a design to reduce token costs.
- JarvisLite: Based on the design of Jarvis, but it excludes composable abstraction, efficient generation with a parameter-free self-reflection mechanism (EP-SR), combined static and dynamic optimization mechanism (SDIO), and the **goal-oriented hardware configuration generation mechanism (GHCG)**.

(3) Performance indicators. Starting from the actual needs of IoT application development, we mainly conduct the following performance evaluations: (a) Function accuracy: Whether the functions generated by the application meet user requirements. (b) Overall accuracy: Whether the generated IoT application has accurate functions and an accurate application specification document. (c) Power consumption per Execution: The total power consumption of the product from triggering to a single execution, including the power consumption of the used devices. (d) Cost of product configuration: The total hardware cost required to configure the application. (e) Token count: The token cost required to generate the application, which represents the primary computational cost and consumption of the designed framework. (f) Trigger discrimination time: The time taken to determine whether to execute an action. (g) Trigger discrimination power consumption: The power consumption associated with determining whether to execute an action.

7.2 Main Results

We first focus on the accuracy of the application. As shown in Tables 1 and 2, the methods based on multi-stage or multi-agent approaches have significant advantages in terms of accuracy. This is because the highly coupled nature of software and hardware in IoT applications makes the solution space extremely complex. Moreover, our accuracy consistently outperforms other solutions in these cases. Jarvis maintains significant advantages even in more complex task scenarios. On the

Table 1. The Function Accuracy and Overall Accuracy of the Generated IoT Application on One-Step TAP Descriptions

Method	GPT-3.5-turbo						GPT-4o-mini					
	Function ACC.			Overall ACC.			Function ACC.			Overall ACC.		
	Easy	Medium	Hard	Easy	Medium	Hard	Easy	Medium	Hard	Easy	Medium	Hard
COT	81.3%	80.2%	30.4%	0%	0%	0%	60.8%	91.1%	100%	0%	0%	0%
COT-SC	83.2%	57.4%	41.3%	0%	0%	0%	100%	95.6%	88.7%	0%	0%	0%
ReAct	78.4%	80.6%	62.3%	0%	0%	0%	52.1%	91.9%	76.5%	0%	0%	0%
SAGE	20.8%	40.4%	62.4%	0%	0%	0%	21.3%	70.6%	100%	0%	0%	0%
Sasha	41.2%	40.6%	41.7%	0%	0%	0%	38.5%	71.1%	100%	0%	0%	0%
ChatIoT	98.6%	82.6%	78.5%	0%	0%	0%	100%	81.1%	92.3%	0%	0%	0%
JarvisLite	91.3%	80.1%	83.2%	0%	0%	0%	100%	63.1%	58.4%	0%	0%	0%
Jarvis	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%

Table 2. The Function Accuracy and Overall Accuracy of the Generated IoT Application on Complex Multi-Step TAP Descriptions

Method	Function (ACC.)			Overall (ACC.)		
	Easy	Medium	Hard	Easy	Medium	Hard
COT	82.3%	58.1%	19.6%	0%	0%	0%
COT-SC	83.1%	41.3%	38.5%	0%	0%	0%
ReAct	78.5%	74.1%	63.8%	0%	0%	0%
SAGE	41.2%	32.1%	28.6%	0%	0%	0%
Sasha	19.6%	26.8%	41.2%	0%	0%	0%
ChatIoT	100%	85.8%	73.8%	0%	0%	0%
JarvisLite	91.2%	84.5%	83.2%	0%	0%	0%
Jarvis	100%	100%	100%	100%	100%	100%

one hand, we construct a software-hardware correlation graph, which effectively helps the LLM understand and utilize the coupling between software and hardware. On the other hand, the solution process of Jarvis employs a reflection mechanism based on design, conducting multiple rounds of reflection to ensure the accuracy of the results. Additionally, Jarvis includes a loop evaluation mechanism, which further guarantees the accuracy of the application.

As shown in Tables 1 and 2, other solutions generally have an accuracy of nearly zero in the final application evaluation. This is because they fail to generate accurate application configuration files, especially in terms of detailed interface counts. In contrast, we cache and update device details in advance during the device selection and configuration phase. Furthermore, during the final configuration file generation phase, we conduct a comprehensive evaluation once again to ensure the accuracy of the configuration file.

Power consumption and cost are important considerations in IoT application development. As shown in Figures 8 and 10, our approach significantly outperforms other solutions in terms of power consumption and cost. As shown in Figures 9 and 11, Jarvis maintains significant advantages even in more complex task scenarios. On the one hand, other works in this area primarily focus on the functionality of task completion. In traditional IoT development, power consumption and cost are extremely important criteria. On the other hand, we abstract the device selection and configuration problem into a multi-objective optimization problem. By solving this problem, we obtain the optimal application generation solution within the current solution space.

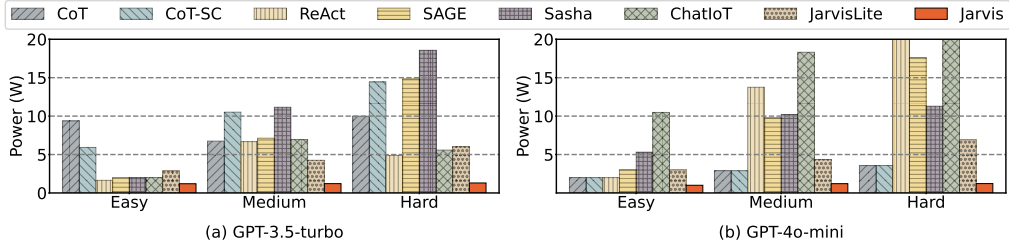


Fig. 8. The power consumption of the generated IoT application on one-step TAP descriptions.

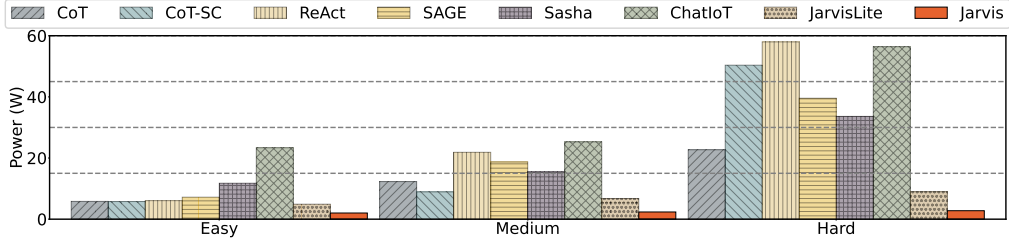


Fig. 9. The power consumption of the generated IoT application on complex multi-step TAP descriptions.

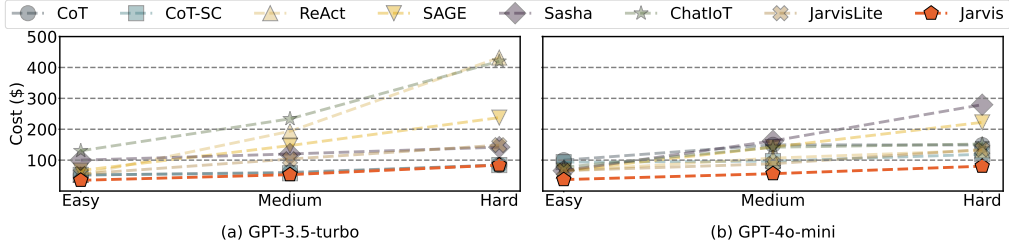


Fig. 10. The configuration cost of the generated IoT application on one-step TAP descriptions.

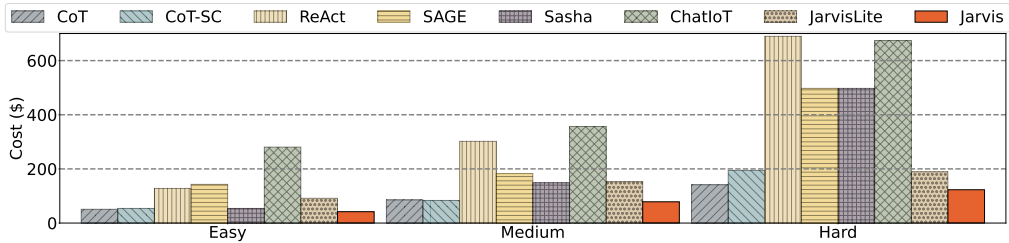


Fig. 11. The configuration cost of the generated IoT application on complex multi-step TAP descriptions.

The cost of computing and development of the application is mainly reflected in the token usage. As shown in Figures 12 and 13, Jarvis always uses the fewest tokens. Jarvis requires significantly fewer tokens than other methods, even in more complex task scenarios. Although ChatIoT also filters some information through context, its approach is based on character-matching models. On the one hand, it affects accuracy, and on the other hand, it ineffectively locates the exact devices,

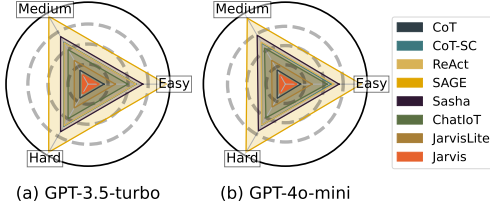


Fig. 12. The number of tokens consumed in generating the IoT application on one-step TAP descriptions.

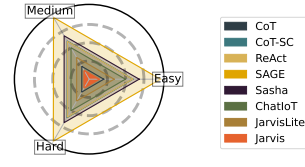


Fig. 13. The number of tokens consumed in generating the IoT application on complex multi-step TAP descriptions.

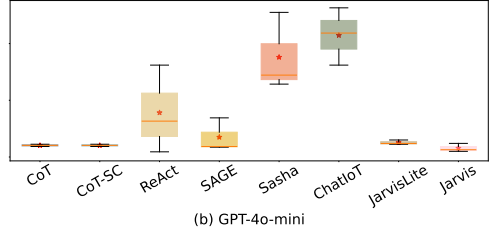
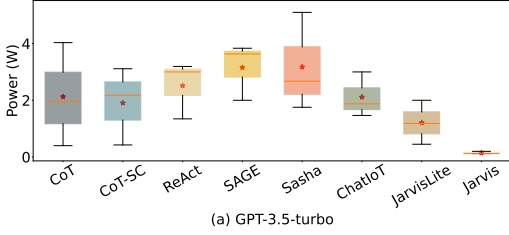


Fig. 14. Trigger discrimination time of the generated IoT application.

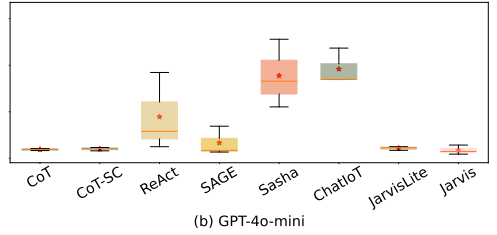
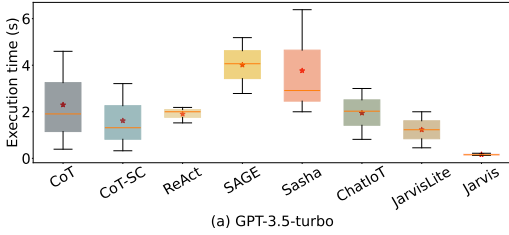


Fig. 15. Trigger discrimination power consumption of the generated IoT application.

resulting in redundant device information. In contrast, Jarvis sends device names and functional descriptions to the LLM, allowing the LLM to select the devices. Ultimately, the multi-objective optimization algorithm designed for local execution performs the optimal device selection, thereby significantly reducing the large token costs associated with uploading additional device information.

As shown in Figures 14 and 15, Jarvis significantly outperforms other algorithms in terms of trigger discrimination time and power consumption. This is because our designed device sequencing strategy ensures that, while maintaining the correctness of application logic, low-power and low-latency triggering devices are prioritized for data collection. Moreover, based on the designed data buffer and logic tree, relevant information is obtained without data collection. This design effectively reduces the power consumption and latency associated with triggering actions.

7.3 Ablation Results

In this section, we conduct ablation studies to demonstrate the necessity of each of our designs. Our ablation designs mainly include four variants: (1) w/o abstraction: The application generation process lacks the abstraction design. (2) w/o EP-SR: The application generation process does not include the efficient generation with a parameter-free self-reflection mechanism. (3) w/o SDIO: After application generation, the optimization mechanism designed is not applied for optimization.

Table 3. The Accuracy and Tokens Consumed for the IoT Applications Generation by Different Ablation Variants on One-Step TAP Descriptions

Method	Function (ACC.)			Overall (ACC.)			Tokens		
	Easy	Medium	Hard	Easy	Medium	Hard	Easy	Medium	Hard
w/o abstraction	100%	100%	100%	51.2%	30.8%	0%	5341	5352	5361
w/o EP-SR	100%	95.7%	100%	100%	89.6%	100%	5579	5589	5592
w/o SDIO	100%	100%	100%	100%	100%	100%	2682	2691	2700
w/o GHCG	100%	100%	100%	100%	100%	100%	2682	2691	2700
Jarvis	100%	100%	100%	100%	100%	100%	2682	2691	2700

Table 4. The Accuracy and Token Consumed for the IoT Applications Generation by Different Ablation Variants on Complex Multi-Step TAP Descriptions

Method	Function (ACC.)			Overall (ACC.)		
	Easy	Medium	Hard	Easy	Medium	Hard
w/o abstraction	100%	86.8%	19.6%	50.2%	16.7%	0%
w/o EP-SR	100%	37.1%	13.7%	100%	34.9%	12.6%
w/o SDIO	100%	100%	100%	100%	100%	100%
w/o GHCG	100%	100%	100%	100%	100%	100%
Jarvis	100%	100%	100%	100%	100%	100%

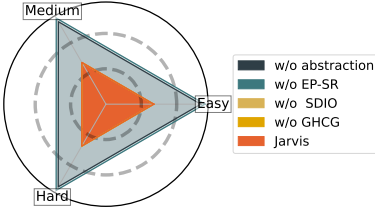


Fig. 16. The number of tokens consumed for the IoT applications generation by different ablation variants on complex multi-step TAP descriptions.

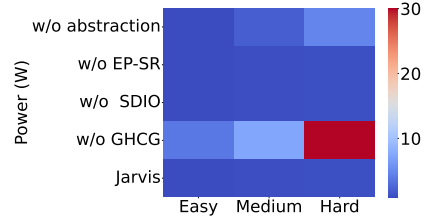


Fig. 17. The power consumption of the IoT applications generated by different ablation variants on one-step TAP descriptions.

(4) w/o GHCG: During the application generation process, the designed objective optimization algorithm is not used.

As shown in Tables 3 and 4, whether in conventional general task scenarios or more complex task scenarios, the accuracy of the final application generated by the w/o abstraction variant performs poorly. This is because the accurate coupling and correlation information of software and hardware in the abstraction enables Jarvis to locate detailed configuration information. On the one hand, it achieves accurate device positioning, and on the other hand, it ensures the accuracy of device information updates. As shown in Table 3 and Figure 16, the w/o abstraction variant requires a higher token cost, both in traditional and complex task scenarios. This is because it needs to submit all information to the LLM to complete device selection and configuration, which significantly increases the token cost. Similarly, the w/o EP-SR variant also has a high token cost. This is because its reflection mechanism has fixed parameters that cannot effectively adapt to different demand scenarios, often generating a large number of redundant solution candidates.

As shown in Figures 17–20, the cost and power consumption of applications generated by the w/o GHCG variant are relatively high. Even in complex task scenarios, the applications developed

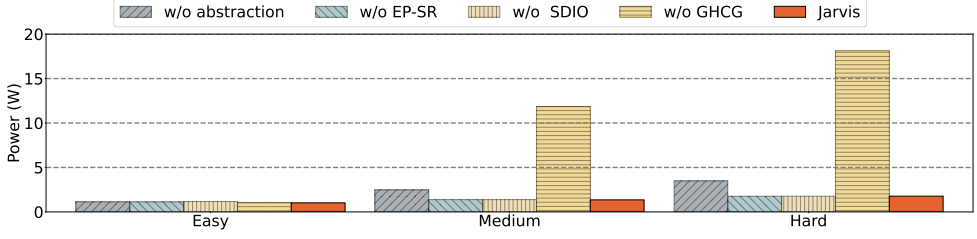


Fig. 18. The power consumption of the IoT applications generated by different ablation variants on complex multi-step TAP descriptions.

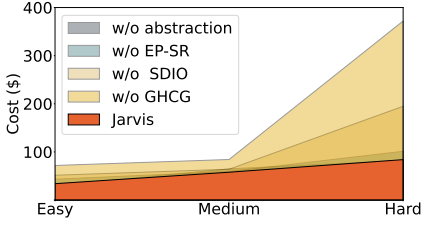


Fig. 19. The cost of the IoT applications generated by different ablation variants on one-step TAP descriptions.

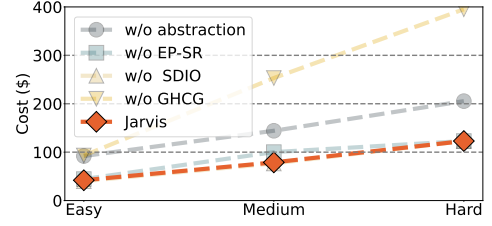


Fig. 20. The cost of the IoT applications generated by different ablation variants on complex multi-step TAP descriptions.

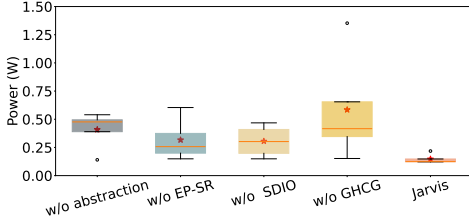


Fig. 21. Trigger discrimination power consumption of different ablation variants.

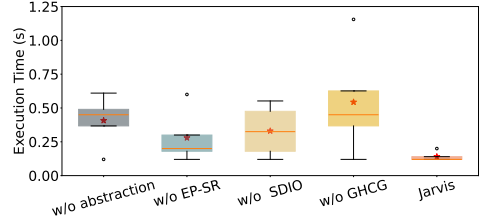


Fig. 22. Trigger discrimination time of different ablation variants.

by the w/o GHCG variant still exhibit higher costs and power consumption compared with other methods. This is because it generates IoT applications solely based on functional objectives without considering power consumption and cost. The w/o abstraction variant does not ensure that the selected device configuration is optimal, as its vast solution space significantly impacts device selection. As a result, the power consumption and cost of the generated application are random and cannot achieve the optimal cost and power consumption.

As shown in Figures 21 and 22, most ablation variants have higher execution power consumption and latency compared with Jarvis. This is because the w/o abstraction and w/o GHCG variants inherently select devices with higher power consumption and execution latency. The w/o SDIO variant does not perform device reordering. As a result, when encountering scenarios where multiple devices are triggered simultaneously, it cannot directly utilize the devices that offer the lowest power consumption and latency.

7.4 Overhead

The overhead of Jarvis primarily comes from three parts: the construction of the abstraction, the search for application devices, and the sorting of application devices. The number of device

Table 5. Overhead Measurement of Jarvis

Overhead (s)	Normal scenarios		Complex scenarios	
	Simple	Difficult	Simple	Difficult
Construction Time	3.01×10^{-5}	3.76×10^{-5}	6.57×10^{-5}	9.71×10^{-5}
Search Time	3.11×10^{-4}	3.64×10^{-3}	3.09×10^{-4}	2.96×10^{-1}
Sorting Time	2.72×10^{-7}	6.03×10^{-7}	3.17×10^{-7}	1.26×10^{-1}

combinations grows exponentially with the increase in device types. Therefore, we define scenarios requiring only 2 device types to complete TAP tasks as normal scenarios, while those requiring more than 2 device types working together are defined as complex scenarios, with the maximum number not exceeding 12. Additionally, since the quantity of each device type significantly affects the overhead of Jarvis, we evaluate the overhead of Jarvis under different numbers of IoT devices being selected. We define *Simple* and *Difficult* generation difficulties. *Simple* indicates fewer than 3 devices per type. *Difficult* indicates more than 3 devices per type, with the maximum quantity capped at 12. As shown in Table 5, the overhead of Jarvis is at the millisecond level. Relative to the large-scale IoT application development, this overhead is acceptable and negligible.

8 Discussion

In this section, we discuss several open problems of Jarvis.

Scalability of Jarvis. Although Jarvis is primarily evaluated on tasks in smart home scenarios, it is scalable to perform zero-code generation in other IoT scenarios such as industrial IoT and vehicular networks. This is because IoT application prototypes fundamentally consist of functional code, hardware peripherals, and computing development boards, which Jarvis is designed to support.

Network Communication Issues. For the network communication aspect, the pipe primitive in our composable abstraction redirects the sensor inference pipeline among multiple devices for energy efficiency. Jarvis generates the inter-device communication using a default code implementation. There are some existing approaches to generate IoT code with customized communication generation, such as IoTPIlot [14]. If developers have more specific requirements towards the inter-device communication, such as to use an alternative protocol or message format, Jarvis is able to seamlessly integrate with existing zero-code code generation solutions by including the function-code agent in IoTPIlot in the generation process of Jarvis. Incorporating such frameworks that include customized communication generation [14, 32] facilitates Jarvis with the generation ability to multi-device collaboration, thereby constructing a large-scale zero-code IoT application system.

Security Issues. Different from the low-code development framework, the zero-code generation framework of IoT directly generates an executable IoT application, which raises security issues. Specifically, the security issue is twofold in Jarvis. First is the soundness of the generated application, i.e., how the generation application follows the original instructions of developers. Jarvis addresses this problem through the evaluation process at the end of the generation. The second issue is the unseen bugs or vulnerabilities in the generated code. Towards this problem, existing works leverage static and dynamic code analysis to detect code vulnerabilities. Recent advances in LLM-based code understanding and bug detection further enable Jarvis with the ability to address the security issues.

9 Conclusion

Zero-code development for IoT applications is a highly accessible and efficient framework, with significant research focus on smart home scenarios. Existing approaches typically LLMs to interpret

task requirements and then use software protocols to integrate device functionalities, fulfilling user development needs. These methods neglect the coupling constraints between software and hardware, which restricts the applicability of these development techniques. In this work, we present Jarvis, a zero-code prototyping framework for IoT applications that incorporates composable hardware-software abstractions. By abstracting the physical and spatial information of hardware and software, Jarvis creates composable hardware-software abstractions and a digital relationship graph for devices. This empowers LLMs with the ability to comprehend the relationships between software and hardware, thereby addressing the coupling challenges in IoT application development. Jarvis further ensures the accuracy of application development through a multi-round reflection mechanism. Considering the power and cost implications of IoT applications, Jarvis formulates the development process as a multi-objective optimization problem. By identifying Pareto optimal solutions, Jarvis achieves reduced power consumption and cost while maintaining application accuracy. We implemented Jarvis and conducted extensive performance evaluations. The results show that, across various scenarios, compared with state-of-the-art methods, the applications generated by Jarvis achieve cost reductions of 23.1%-75.5% and power savings of 18.7%-95.2%. Additionally, Jarvis reduces token costs during application generation by 32.1%-86.2%, while accurately generating application entities and their configuration files.

References

- [1] Bayan Al Muhandar, Jason Wiese, Omer Rana, and Charith Perera. 2023. Interactive privacy management: Toward enhancing privacy awareness and control in the Internet of Things. *ACM Transactions on Internet of Things* 4, 3 (2023), 1–34.
- [2] Arman Anzanpour, Delaram Amiri, Iman Azimi, Marco Levorato, Nikil Dutt, Pasi Liljeberg, and Amir M. Rahmani. 2020. Edge-assisted control for healthcare internet of things: A case study on ppg-based early warning score. *ACM Transactions on Internet of Things* 2, 1 (2020), 1–21.
- [3] Ons Aouedi, Thai-Hoc Vu, Alessio Sacco, Dinh C. Nguyen, Kandaraj Piamrat, Guido Marchetto, and Quoc-Viet Pham. 2025. A survey on intelligent Internet of Things: Applications, security, privacy, and future directions. *IEEE Communications Surveys & Tutorials* 27, 2 (2025), 1238–1292.
- [4] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 17682–17690.
- [5] Tung-Chun Chang, Chih-Chun Wu, Georgios Bouloukakakis, Cheng-Hsin Hsu, and Nalini Venkatasubramanian. 2025. SmartParcels: Constructing smart communities through human-in-the-loop urban IoT planning. *ACM Transactions on Internet of Things* 6, 2 (2025), 1–31.
- [6] Soteris Constantinou, Andreas Konstantinidis, Panos K Chrysanthis, and Demetrios Zeinalipour-Yazti. 2022. Green planning of IoT home automation workflows in smart buildings. *ACM Transactions on Internet of Things* 3, 4 (2022), 1–30.
- [7] Wei Dong, Borui Li, Gaoyang Guan, Zhihao Cheng, Jiadong Zhang, and Yi Gao. 2020. TinyLink: A holistic system for rapid development of IoT applications. *ACM Transactions on Sensor Networks (TOSN)* 17, 1 (2020), 1–29.
- [8] Biyi Fang, Qiumin Xu, Taiwoo Park, and Mi Zhang. 2016. AirSense: An intelligent home-based sensing system for indoor air quality analytics. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 109–119.
- [9] Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. 2024. Detecting hallucinations in large language models using semantic entropy. *Nature* 630, 8017 (2024), 625–630.
- [10] Silvery Fu and Sylvia Ratnasamy. 2021. dSpace: Composable abstractions for smart spaces. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 295–310.
- [11] Yi Gao, Wei Dong, Kai Guo, Xue Liu, Yuan Chen, Xiaojin Liu, Jiajun Bu, and Chun Chen. 2016. Mosaic: A low-cost mobile sensing system for urban air quality monitoring. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.
- [12] Yi Gao, Kaijie Xiao, Fu Li, Weifeng Xu, Jiaming Huang, and Wei Dong. 2024. ChatIoT: Zero-code generation of trigger-action based IoT programs. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 3 (2024), 1–29.
- [13] Yingqiang Ge, Wenyue Hua, Kai Mei, Juntao Tan, Shuyuan Xu, Zelong Li, Yongfeng Zhang, et al. 2023. Openagi: When llm meets domain experts. *Advances in Neural Information Processing Systems* 36 (2023), 5539–5568.

- [14] Kaijie Gong, Wei Dong, Yingqi Peng, Hao Wang, and Yi Gao. 2024. Poster: Enabling IoT application programming in natural language with IoT pilot. In *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*. 903–904.
- [15] Daniel Graham and Gang Zhou. 2016. Prototyping wearables: A code-first approach to the design of embedded systems. *IEEE Internet of Things Journal* 3, 5 (2016), 806–815.
- [16] Gaoyang Guan, Borui Li, Yi Gao, Yuxuan Zhang, Jiajun Bu, and Wei Dong. 2020. TinyLink 2.0: Integrating device, cloud, and client development for IoT applications. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. 1–13.
- [17] Baoshen Guo, Shuai Wang, Yi Ding, Guang Wang, Suining He, Desheng Zhang, and Tian He. 2021. Concurrent order dispatch for instant delivery with time-constrained actor-critic reinforcement learning. In *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 176–187.
- [18] Mehrdad Hesar, Vikram Iyer, and Shyamnath Gollakota. 2016. Enabling on-body transmissions with commodity devices. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 1100–1111.
- [19] Luke Hsiao, Sen Wu, Nicholas Chiang, Christopher Ré, and Philip Levis. 2020. Creating hardware component knowledge bases with training data generation and multi-task learning. *ACM Transactions on Embedded Computing Systems (TECS)* 19, 6 (2020), 1–26.
- [20] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems* 43, 2 (2025), 1–55.
- [21] Adam Tauman Kalai and Santosh S. Vempala. 2024. Calibrated language models must hallucinate. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*. 160–171.
- [22] Patrick Killeen, Iluju Kiringa, and Tet Yeap. 2022. Unsupervised dynamic sensor selection for iot-based predictive maintenance of a fleet of public transport buses. *ACM Transactions on Internet of Things* 3, 3 (2022), 1–36.
- [23] Evan King, Haoxiang Yu, Sangsu Lee, and Christine Julien. 2024. Sasha: Creative goal-oriented reasoning in smart homes with large language models. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 1 (2024), 1–38.
- [24] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. 2022. Model-driven self-adaptive deployment of internet of things applications with automated modification proposals. *ACM Transactions on Internet of Things* 3, 4 (2022), 1–30.
- [25] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in Neural Information Processing Systems* 35 (2022), 22199–22213.
- [26] Borui Li and Wei Dong. 2020. Automatic generation of IoT device platforms with autolink. *IEEE Internet of Things Journal* 8, 7 (2020), 5893–5903.
- [27] Borui Li and Wei Dong. 2020. Edgeprog: Edge-centric programming for iot applications. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 212–222.
- [28] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis. arXiv:2305.15334. Retrieved from <https://arxiv.org/abs/2305.15334>
- [29] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. arXiv:2307.16789. Retrieved from <https://arxiv.org/abs/2307.16789>
- [30] Dmitriy Rivkin, Francois Hogan, Amal Feriani, Abhisek Konar, Adam Sigal, Xue Liu, and Gregory Dudek. 2025. AIoT smart home via autonomous llm agents. *IEEE Internet of Things Journal* 12, 3 (2025), 2458–2472.
- [31] Leong Yee Rock, Farzana Parveen Tajudeen, and Yeong Wai Chung. 2024. Usage and impact of the internet-of-things-based smart home technology: A quality-of-life perspective. *Universal Access in the Information Society* 23, 1 (2024), 345–364.
- [32] Leming Shen, Qiang Yang, Yuanqing Zheng, and Mo Li. 2025. AutoIoT: LLM-driven automated natural language programming for AIoT applications. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. 1–15.
- [33] Devkishen Sisodia, Jun Li, Samuel Mergendahl, and Hasan Cam. 2024. A two-mode, adaptive security framework for smart home security applications. *ACM Transactions on Internet of Things* 5, 2 (2024), 1–31.
- [34] Nahar Sultana, Farhana Huq, Palash Roy, Md Abdur Razzaque, Md Mustafizur Rahman, Taiyeba Akter, and Mohammad Mehedi Hassan. 2025. Context aware clustering and meta-heuristic resource allocation for NB-IoT D2D devices in smart healthcare applications. *Future Generation Computer Systems* 162 (2025), 107477.
- [35] Hsin-Ruey Tsai, Shih-Yao Wei, Jui-Chun Hsiao, Ting-Wei Chiu, Yi-Ping Lo, Chi-Feng Keng, Yi-Ping Hung, and Jin-Jong Chen. 2016. iKneeBraces: Knee adduction moment evaluation measured by motion sensors in gait detection. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 386–391.

- [36] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. 2017. {FarmBeats}: An {IoT} platform for {data-driven} agriculture. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 515–529.
- [37] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 186345.
- [38] Xin Wang, Zhongwei Wan, Arvin Hekmati, Mingyu Zong, Samiul Alam, Mi Zhang, and Bhaskar Krishnamachari. 2024. Iot in the era of generative ai: Vision and challenges. *IEEE Internet Computing* 28, 5 (2024), 57–64.
- [39] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. arXiv:2203.11171. Retrieved from <https://arxiv.org/abs/2203.11171>
- [40] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.
- [41] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems* 36 (2023), 11809–11822.
- [42] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. arXiv:2210.03629. Retrieved from <https://arxiv.org/abs/2210.03629>
- [43] Lan Zhang, Xiang-Yang Li, Wenchao Huang, Kebin Liu, Shuwei Zong, Xuesi Jian, Puchun Feng, Taeho Jung, and Yunhao Liu. 2014. It starts with igaze: Visual attention driven networking with smart glasses. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*. 91–102.
- [44] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. arXiv:2210.03493. Retrieved from <https://arxiv.org/abs/2210.03493>
- [45] Di Zhao, Tianxu Niu, Bin Song, and Xiaojiang Du. 2025. Service-oriented resource allocation and task scheduling for wi-fi and bluetooth coexistence in smart home IoT systems. *ACM Transactions on Internet of Things* 6, 2 (2025), 1–24.
- [46] Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2025. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *Thirteenth International Conference on Learning Representations*. Retrieved from <https://openreview.net/forum?id=YrcTjllL0>

Received 28 February 2025; revised 19 May 2025; accepted 6 August 2025