IoTGen: Towards LLM-Driven IoT Hardware PCB Generation

Qinpei Luo
University of California, San Diego
qpluo@ucsd.edu

Ruichun Ma*
Microsoft Research Asia
ruichunma@microsoft.com

Xinyu Zhang
University of California, San Diego
xyzhang@ucsd.edu

Lili Qiu
Microsoft Research Asia
The University of Texas at Austin
lili@cs.utexas.edu

Abstract

With the rapid growth of IoT and AIoT applications, the demand for customized hardware is surging, yet PCB design for IoT devices remains a heavily manual, GUI-centric process that requires significant expertise in circuit and tools. This creates a widening gap between the accelerated software development and the difficulty of realizing corresponding hardware, making PCB-based hardware a critical bottleneck for system innovation. We present *IoTGen*, an LLM-driven agentic system that generates IoT PCB designs directly from natural-language demands. *IoTGen* introduces semantic-rich programming abstractions that capture schematic construction, enabling a domain-specialized language model to synthesize schematics with code. It further integrates a semantic component retrieval algorithm that combines LLM understanding with vector-based search, and an LLM-guided hybrid layout procedure that coordinates auto-routing tools to produce PCB layouts suitable for fabrication. We evaluate *IoTGen* on a dataset of diverse IoT designs, achieving a high component matching accuracy of 90%, a schematic generation success rate of 82%, and significant layout improvement. Case studies further demonstrate its capability to generate IoT PCB designs while substantially reducing the human effort and domain expertise required for hardware development. We release *IoTGen* to facilitate further research.

CCS Concepts

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

Keywords

Large Language Model, Internet of Things, Computer-aided design

ACM Reference Format:
Qinpei Luo, Ruichun Ma, Xinyu Zhang, and Lili Qiu. 2026. *IoTGen*: Towards LLM-Driven IoT Hardware PCB Generation. In *The 24th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '26), June 21–25, 2026, Cambridge, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3745756.3809239

*Ruichun Ma is the corresponding author.

This work is licensed under a Creative Commons Attribution 4.0 International License.
MobiSys '26, Cambridge, United Kingdom
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2027-7/2026/06
https://doi.org/10.1145/3745756.3809239

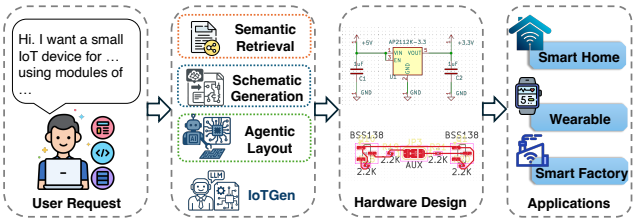


Figure 1: Illustration of *IoTGen*. *IoTGen* generates IoT hardware PCB designs for various applications, using a semantic retrieval algorithm, a specialized language model for schematic generation, and an agentic layout procedure.

1 Introduction

With the growing demand for Internet of Things (IoT) devices and, more recently, Artificial Intelligence of Things (AIoT), the need for customized hardware solutions is increasing rapidly. Printed circuit boards (PCBs) are the backbone of almost all electronic hardware, serving as the physical instantiation of hardware design and the substrate that composes electronic components, chips, and circuits. Researchers, engineers, and students routinely need customized PCBs to meet their specific use cases, e.g., experimenting with new prototypes, developing new products, and learning by building actual devices. IoT devices in particular require frequent hardware customization, such as integrating specific new sensors, wireless connectivity modules, power management circuits, and micro-controllers.

Despite the demand, there remains a significant gap between a design requirement and PCB designs that can be fabricated. Today's IoT PCB design workflow is still GUI-centric and manual, demanding substantial domain expertise to develop the design and operate Electronic Design Automation (EDA) tools (Section 2.1). This difficulty in creating hardware on demand limits the pace and breadth of IoT innovation. At the same time, we see a growing mismatch between how easily we can develop software and how hard it remains to design hardware. Large language models (LLMs) have transformed software development, enabling developers to move from intent to code with unprecedented speed. Recent research work has explored the use of LLMs for IoT and embedded programming, showing promising results for generating simple application code [37, 38, 49]. In contrast, hardware aspects remain largely untouched by such automation, emerging as a critical bottleneck for end-to-end IoT system development.

This paper asks: *How can we enable IoT hardware development automation?* Specifically, we aim to achieve automated PCB generation for IoT systems from high-level design intent. Inspired by how LLMs revolutionize software development, we seek to automate the PCB design workflow using LLMs specialized for hardware design. Our key insight is to introduce *semantic-rich programming abstractions* that expose the PCB design workflow as a software-defined process. These abstractions turn hardware design into a generative AI problem: Given a specification of the desired IoT device, an agent can plan, search, and synthesize the corresponding PCB. This also makes IoT hardware design an integral, programmable part of the IoT development workflow rather than a specialized, isolated task.

We present *IoTGen*, an LLM-driven system that generates IoT PCB designs from natural language descriptions (Figure 2). We focus on IoT devices as it frequently requires customization for integrating diverse sensors and communication modules. The core of our approach is a set of programming abstractions for PCB schematic design generation, which makes the design process programmable and casts the problem as a generative AI task. Given a description of the desired hardware design, *IoTGen* identifies the necessary circuit components, creates the corresponding schematic design, and prepares the final PCB layout for fabrication. Though *IoTGen* can significantly speed up the design process, it cannot guarantee full correctness. To ensure robustness and safety, users can confirm at each design stage that the generated PCB aligns with actual needs.

There are several key challenges we need to address: (1) As the core of the design process, PCB schematic design captures the logical connectivity and is typically created through a GUI, requiring deep knowledge of design principles and EDA tools. Besides, current commercial LLMs *cannot* generate usable schematic design due to the lack of representations (Section 3.3). To this end, we propose a set of semantic-rich programming abstractions that expose schematic operations as a software-defined workflow. LLMs can then reason over these abstractions and generate a Python code representation that constructs schematic designs. We further train a domain-specific language model to internalize the knowledge of PCB schematic design and generate high-quality schematic designs. (2) Even for experienced designers, searching for suitable circuit components for a target design is challenging. There are a massive and fragmented component ecosystem, spanning RLC elements, connectors, sensors, and wireless modules across manufacturers, footprints, specifications, and prices. Existing component search engines assume strong prior expertise to search by manufacturer, part number, and other detailed attributes. To address this, we propose a novel component search algorithm that combines the semantic understanding of LLMs with vector-based retrieval to find the most suitable circuit components. (3) PCB layout requires both high-level planning to decide spatial relationships and detailed optimization of routing topology, as it determines the physical arrangement of components and copper traces between them. Purely automated routers often perform poorly without good high-level guidance, especially on complex designs. To balance these aspects, we propose a hybrid approach that uses LLMs for layout planning, while leveraging existing auto-routers for detailed wire routing.

We evaluate *IoTGen* through both quantitative metrics and qualitative case studies in Section 4. For a testing dataset of 2481 schematic designs, *IoTGen* achieves 82% success rate in schematic generation, and 90% in component search. Our programming abstraction for schematic generation significantly outperforms using existing schematic file formats by 32% of netlist accuracy. Building on this abstraction, our fine-tuned domain-specific language model surpasses commercial models with 11.5% accuracy on schematic generation, with a substantially smaller model size. Moreover, case studies, including a multi-sensor module and a CO2 monitoring device, demonstrate the practicality and effectiveness of the end-to-end workflow.

We make the following Contributions:

- We present *IoTGen*, the first system that generates IoT PCB designs directly from natural-language requests, bringing the generation capabilities of large language models to the IoT hardware domain.
- We introduce semantic-rich programming abstractions that bridge code generation and PCB schematic design. And we train a novel domain-specialized language model that utilizes such code representation for schematic generation.
- We develop a semantic-based component search algorithm and an LLM-guided hybrid PCB layout approach, enabling an end-to-end automated workflow from intent to PCB design.
- The evaluation results show *IoTGen* has superior performance in component search, schematic generation, and PCB layout. On top of that, we conduct two case studies that verify the feasibility of the proposed system on generating actual IoT devices. We release the code, dataset, and model of *IoTGen*[1] to facilitate further research.

2 Background and Related Works

2.1 PCB Design Background

Tools. Designing IoT devices begins with translating functional requirements into concrete electronic circuitry, and this process is almost universally mediated by computer-aided tools such as Electronic Design Automation (EDA) software. In practice, engineers rely on both proprietary tools—Altium Designer [2], Cadence Design Systems [4], EAGLE [10]—and open-source solutions like KiCad [21]. Although these tools serve the same purpose and support similar schematic elements and editing workflows, they adopt very different schematic file formats. Throughout this paper, we adopt the *KiCad* schematic format as our running example because it is open-source and widely accessible in the IoT hardware community.

Design workflow. From an IoT designer's perspective, the process typically starts with selecting appropriate circuit components, such as sensors, MCUs, radios, and power regulators. These components are then placed on the schematic canvas, each represented by a *symbol*, to create a spatially sensible schematic diagram, after which the engineer connects pins with wires to form the underlying electrical network. This step is crucial because it determines both the functional composition of the IoT device and the validity of its electrical behavior. Yet it is also tedious—dozens of symbols may appear in a small IoT node, and hundreds of pins need careful wiring.

[1] https://github.com/luoqinpei/IoTGen.git

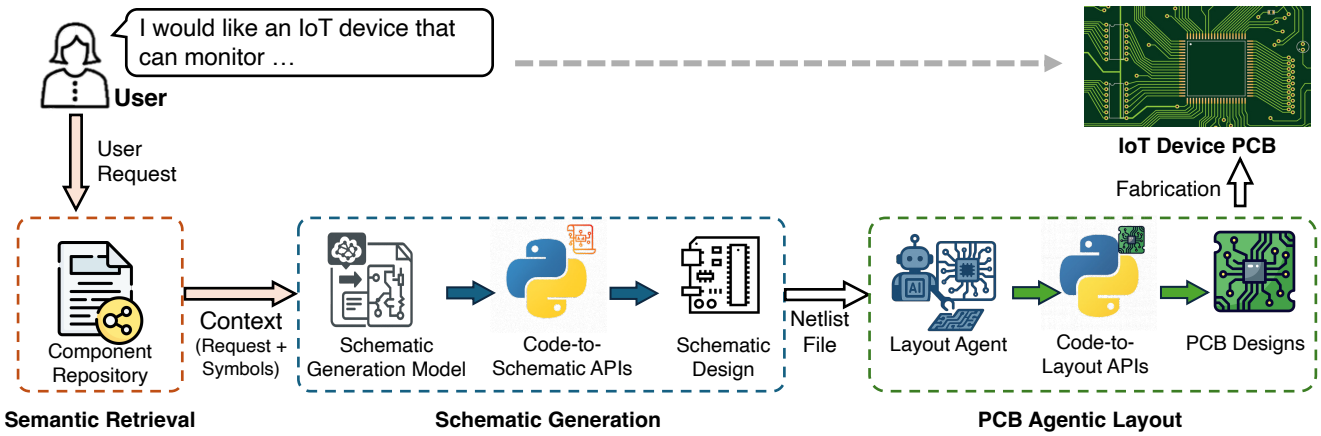


Figure 2: *IoTGen* Overview. Given a natural-language request, *IoTGen* enriches the context with semantic retrieval for circuit components, then uses our schematic generation model to create schematics via the programming abstractions, and invokes an agentic layout procedure to obtain PCB designs that can be fabricated

Once the schematic is completed, the design process transitions to PCB layout and wiring, where the logical circuit is translated into its physical form. In this stage, engineers place each footprint—physical land patterns corresponding to schematic symbols—onto a board outline and arrange them on the board. After components are positioned, routing connects the corresponding pads with copper traces across multiple layers. Modern EDA tools assist with auto-routing, but achieving a high-quality layout for IoT boards often requires iterative refinement, balancing electrical integrity, power delivery, antenna performance, space constraints, and other possible requirements specific to the design purpose.

Schematic content. In the context of IoT device design, each schematic file encodes the fundamental building blocks of the circuit. It contains three main types of items: (1) component symbols representing physical elements such as microcontroller chips, sensors, resistors, and capacitors; (2) power symbols such as VCC and GND, which denote supply rails essential to every IoT board; and (3) net labels, which declare named electrical nets. Labels that share the same name are treated as electrically connected, and they often specify the input/output interfaces of modular IoT subsystems, allowing separate functional blocks to integrate cleanly through shared labels. Each component symbol includes one or more pins—either identified by pin IDs or human-readable names. For instance, an IoT microcontroller features pins such as VCC and GND for power, and GPIO1, GPIO2, etc., for interfacing with sensors, while an LED symbol uses "A" (anode) and "K" (cathode). Power symbols and net labels typically expose a single pin to support wire connections, and the engineer draws wires between pins and labels to define the circuit's electrical connectivity.

Design automation. Despite the importance of schematics, existing design automation tools primarily concentrate on later stages, in particular PCB layout [26, 42, 43] and routing [23, 39, 50], because these tasks are comparatively more structured. As a result, schematic design remains manual and non-scalable.

2.2 Generative AI for Hardware Designs

There have been many recent works on generative AI-based hardware designs. Built on hierarchically abstracted Boolean logic representations, e.g., Verilog and VHDL, Fu et al. [13], Thakur et al. [41], Wu et al. [45] explore language model-based digital IC design. For analog IC design, CktGNN [9] first represents the circuit topology as graph structures, formulating a graph generation task to enable the design of various topologies. AnalogCoder [22] presents a training-free LLM agent for designing analog circuits through Python code generation, while LaMAGIC [5] fine-tunes a masked language model. AnalogGenie [14] presents a dataset and develops a sequence-based graph representation to enable LLM pre-training for novel circuit topology generation.

These prior works target the specific design tasks with mature programming abstractions, while our task of PCB and schematic generation remains unexplored, lacking key programming abstractions and corresponding system design. Recent works [19, 47] have made some progress on conversion from circuit diagram images to netlists, and Matsuo et al. [28] focuses on netlist-to-schematic translation, which is different from our task of generating PCB designs from user requests. PCB-Bench [24] provides a benchmark for commercial LLMs regarding their performance on the task of PCB placement and layout, which only covers the last step of the whole design pipeline. Several commercial products like *Flux.ai* [12] are moving towards this direction, but so far provide limited, unreliable, and slow services as they typically rely on general-purpose commercial LLMs without domain-specific system designs.

2.3 LLMs for IoT

Recent studies regarding generative AI for IoT focus on programming sensor systems. *AutoIoT* and *GPIoT* [37, 38] propose an LLM-based automated program generator to perform AI-driven IoT tasks like heartbeat detection. [6, 34, 36] pay emphasis on sensor data processing, using LLM to retrieve semantic understanding from raw sensor data to build AI-driven personal assistants. *LLM-CoSen* [11]

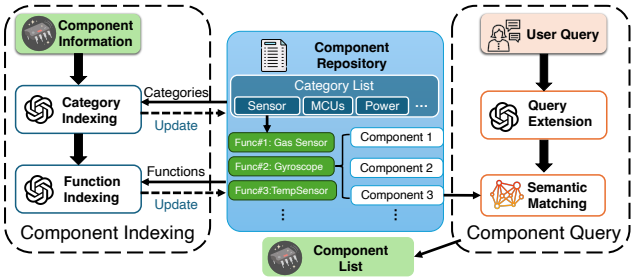


Figure 3: Semantic Indexing and Query of Circuit Components. We construct a structured repository to enable accurate and efficient semantic-based component query.

considers collaborative sensing with LLM, making efforts to mitigate inference error in LLM-enabled sensor systems. [15, 25] tackle the issues of sensor system triggering and management. Other works leverage LLM to help with the software design part with Arduino for embedded systems [49] However, these works only focus on sensor data or the software design part of IoT systems, without a solution for enabling LLM to design IoT hardware. To the best of our knowledge, *IoTGen* is the first system that enables large language models to generate PCB designs for IoT hardware.

3 *IoTGen* Design

3.1 System Overview

The end-to-end workflow of *IoTGen* is shown in Figure 2. At its core, *IoTGen* integrates semantic component retrieval, LLM-driven schematic synthesis, and an agentic PCB layout engine into a unified design pipeline. Given a natural-language request, the system first performs *semantic retrieval* over a curated component repository to extract device-level specifications and symbol information. This enriched context is then fed into the *schematic-generation model*, which employs code-to-schematic APIs to construct a valid circuit diagram and corresponding netlist. Using the produced netlist, *IoTGen* activates its *PCB agentic layout*, a specialized design agent that iteratively places, routes, and evaluates PCB layouts through code-to-layout APIs. The resulting PCB design is exported to fabrication files, enabling the generation of a manufacturable IoT prototype.

We note that, although the modules of *Semantic Retrieval* and *PCB Agentic* rely on commercial LLM APIs, they are model-agnostic and the performance is expected to be improved with more advanced models in the future.

3.2 Semantic Retrieval

Given the specific request, the first step is to select appropriate circuit components. This procedure typically relies on the designer's expertise, manually searching online to determine the component list, which is inefficient and complex. To address this, we build a structured component repository, making it more efficient to use LLMs for semantic component search. On top of that, we design a hybrid method of LLM and vector search, which takes the user request as input and retrieves PCB components from the repository.

Challenges. The original *KiCad* component repository presents several challenges that motivate a structured, LLM-friendly indexing pipeline. First, the repository suffers from **complex organization**, mixing manufacturer-based and function-based groupings. For instance, ESP32 devices appear in both "MCU_Espressif" and "RF_Module", which makes the structure difficult for an LLM to interpret. Second, effective component selection requires **high-level semantic interpretation**, since similarity-based retrieval methods (e.g., cosine similarity as in [49]) often misrank entries; for example, the request "An LED powered by 3.3V" may be incorrectly matched to "LED Driver" due to naïve token-level similarity. Third, the repository contains a **large volume of component data** of over 20K devices with extensive metadata, which far exceeds the context window of modern LLMs and necessitates hierarchical, reasoning-driven retrieval.

Structured component repository. To address the above issues, one intuitive solution is to re-index all components according to a unified and LLM-friendly organizational scheme. Thus, we propose a four-layer structure as "*Category–>Function–>Lib ID–>component ID*", where *Lib ID* and *component ID* are defined in *KiCad* library. Here, we define *category* as a high-level domain reflecting the component's broad electrical role (e.g., Power, Analog, RF, Sensor, Passive), and *Function* as a semantically coherent subgroup within a *category* describing the component's specific operational role (e.g., within Power: Regulator, Battery Management, Gate Driver).

A remaining challenge is determining the appropriate set of categories and functions. Methods like hierarchical clustering [31] can automatically build a tree-like structure from the data in an unsupervised way. However, due to the semantical complexity of each record of the component, it is challenging and unreliable to divide them merely depending on the distance in the latent space after tokenization.

Adaptive clustering. To address this, we introduce a hierarchical clustering approach to do the component indexing as illustrated in Fig. 3. We begin by extracting raw component information from the `.kicad_sym` files and enriching them using online resources such as datasheets with LLM assistance. We then maintain a curated repository consisting of a category list and function lists corresponding to each category. For each component, the LLM agent is provided with its information as well as the current lists. If the component semantically matches an existing category or function, it is assigned accordingly; otherwise, the agent proposes a new category or function, which is subsequently added to the repository. This iterative process yields a coherent, semantically structured library covering all *KiCad* components.

Semantic-based matching. As discussed above, directly querying the entire component library with an LLM is both time- and cost-intensive, while performing a straight vector search over the repository using natural language queries is prone to errors. To tackle this issue, we first call the LLM agent for query extension according to the taxonomy of the re-organized component repository. *IoTGen* prompts the LLM to analyze the circuit-level requirements and infer a structured list of *search keys*, each containing the function name, possible footprint, and relevant technical specifications. These predictions are grounded in the taxonomy and function hierarchy of the organized component repository, ensuring that the

Algorithm 1: Component Query

Input: q (user request)
Output: S_{final} (final selected components)
Load Taxonomy $\mathcal{T}$, Repository $\mathcal{R}$, LLM Agent$\mathcal{A}$;
Search Keys:$\mathcal{K} \leftarrow \mathcal{A}$.Query_Extension$(q, \mathcal{T})$;
$S_{\text{candidate}} \leftarrow \emptyset$;
foreach *Key* $k \in \mathcal{K}$; // Semantic Matching
do
  $S_{\text{matched}} \leftarrow \mathcal{R}$.hybrid_search$(k)$;
  $S_{\text{candidate}} \leftarrow S_{\text{candidate}} \cup S_{\text{matched}}$;
$S_{\text{final}} \leftarrow \mathcal{A}$.Realign$(q, S_{\text{candidate}})$;
return S_{final};

LLM's reasoning is constrained by the available library structure rather than only the natural-language query. Using the generated keys, we perform a semantic matching, including a hybrid vector search over the component repository and LLM realignment. The hybrid vector search combines full-text search (FTS) using SQLite's built-in FTS5 engine and approximate nearest neighbor (ANN) retrieval over sentence-transformer embeddings [35]. FTS evaluates keyword and semantic relevance at the text level, while ANN measures embedding similarity in the latent space. The two scores are fused to produce a ranked list of candidate components for each function. The union of all candidate results forms a raw candidate pool. To ensure that the final output respects the user's high-level design intent, we invoke the LLM again to realign and refine the candidates. Given the user query and the candidate pool, the LLM selects the most relevant to produce the final component list. The entire pipeline is outlined in Alg. 1 and summarized in Fig. 3.

User confirmation. After retrieving the related components, we allow the users to make an informed confirmation. A major source of failure is that a vague user request can correspond to multiple potential designs. By providing an easy-to-understand functional description of components, we let users decide whether this matches their expectations. If not, users can revise the input request and restart the workflow.

3.3 Schematic Generation

To achieve the challenging but critical step of schematic generation, we introduce programming abstractions that explicitly encode both the semantics of schematic-editing operations and the underlying rationale governing component placement and electrical interconnections. Building on these abstractions, we construct a large dataset and train a specialized large language model capable of synthesizing schematic designs directly from user-specified requirements.

3.3.1 ***Code Representation for Schematic Designs.*** Although existing LLMs claim to have the ability to generate PCB schematic designs when asked in a prompt, they fail to produce valid schematic files that can be parsed by EDA tools, e.g., *KiCad* . We argue that the main reason is that the representation of PCB schematics is not learning efficient for LLMs. Section 3.3 compares existing representations and our proposed code representation. When using text-based representation, e.g., raw *KiCad* schematic content in symbolic expression format, LLMs struggle to capture the format of the schematic due to the presence of excessive version-specific formatting details and redundant information that are irrelevant to the schematic's functionality. When using LLM and image generation, e.g., GPT-5, the generated schematic images contain distorted components and show random format, making it infeasible to convert them into valid schematic files that can be leveraged for downstream design.

To tackle the challenges above, our goal is to find a new representation that is suitable for this task and learning efficient for LLMs. Our approach is inspired by the observation that human engineers typically follow a systematic process when designing schematics, which can be abstracted into a series of editing operations backed by clear rationales. Specifically, engineers first place central components that represent the core components of the circuit, then arrange other components around them based on their functional relationships. Finally, we connect the pins according to the semantic connections between circuit component pins, e.g., VCC pins are connected to power source components, and GND pins are connected to ground components. We summarize two key insights from the above process: (1) the schematic design can be abstracted as a series of editing operations, including adding components, placing labels, and connecting pins; (2) the placement of components and labels is typically relative to a local reference point based on functional correlations; the wire connections follow clear rationales based on the pin names that encode the pin functionalities concisely.

Programming abstractions. Based on the observation, we introduce the following APIs:

```
def add_schematic_symbol(component_lib, component_name, x, y, ref,
      value, rotation, mirror)
def add_label(label_pos, label_text, label_ref, label_type,
      text_orient)
def get_pin_location(component_reference, pin_name)
def connect_pins(component_ref_a, pin_name_a, component_ref_b,
      pin_name_b)
def write_out_all_wires()
```

add_schematic_symbol() places a component with the given name from the component list on the assigned location with optional rotating and mirroring operations, meanwhile assigning a unique reference name and an optional value string. *add_label()* places net labels with given text on specific locations and orientation, meanwhile allowing specifying label types (e.g., input, output, bidirectional) and a unique label reference ID. *get_pin_location()* gets the location of a specific pin, queried by component reference name and the pin name. For power components and net labels that have one pin only, the pin name is set as default to '1'. *connect_pins()* connect two pins according to the component reference names and pin names of the two pins. Finally, *write_out_all_wires()* performs an automatic routing and writes out all wires on the schematic of specified connections. Section 3.3 illustrates code snippets and corresponding parts on the schematic.

Spatial relations. LLMs' ability to understand spatial relationships is relatively limited [48]. For our task, the schematic may involve dozens of distinct components/labels and long numerical coordinates, which are hard for LLMs to generate correctly. To handle this issue, we choose to use local coordinate systems rather than

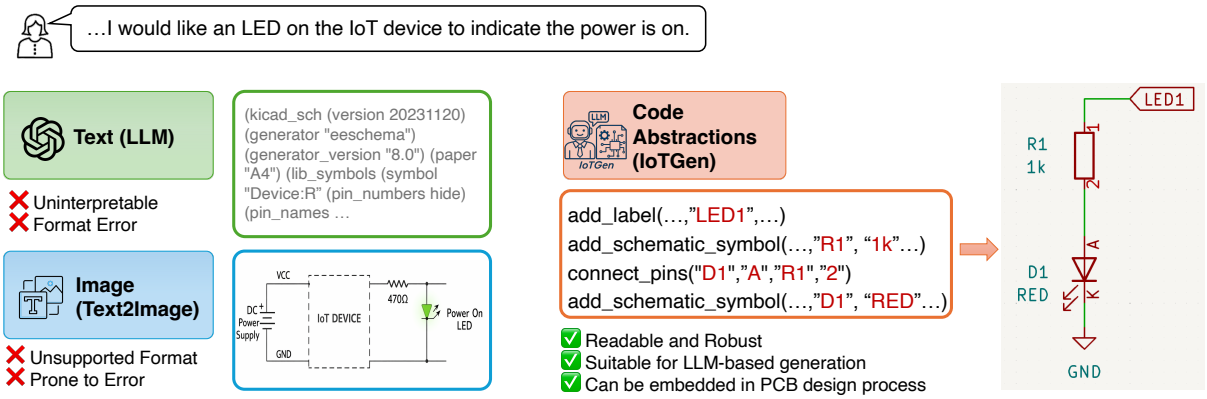


Figure 4: Comparison of generating schematic with different representations. Our proposed code representation builds on our custom programming abstractions, containing rich semantic information to allow effective LLM training.

Table 1: Comparison of different representations.

Repr.	Example

Code-L1	`add_schematic_symbol(... pos_x=center_x_1, pos_y=center_y_1, ...)`
	`add_schematic_symbol(... pos_x=center_x_1 + (-58.42), pos_y=center_y_1 + (17.78), ...)`
	`connect_pins("PWR1", "+1V8", "U2", "VDDIO")`

Code-L2	`add_schematic_symbol(... pos_x=157.48, pos_y=99.51, ...)`
	`add_schematic_symbol(... pos_x=99.06, pos_y=117.29, ...)`
	`connect_pins("PWR1", "+1V8", "U2", "VDDIO")`

Code-L3	`add_schematic_symbol(... pos_x=157.48, pos_y=99.51, ...)`
	`add_schematic_symbol(... pos_x=99.06, pos_y=117.29, ...)`
	`add_new_wire([99.06, 117.29], [114.3, 117.29])`

KiCad	`(lib_symbols(symbol "Device:Q_NMOS_DGS" pin_names(offset 0) hide (exclude_from_sim no) (in_bom yes) (on_board yes) (property "Reference" "Q" (at 5.08 1.27 0) (effects(font(size 1.27 1.27))(justify left))) )) ...`

absolute coordinates in *add_schematic_symbol* and *add_label*. More specifically, we first get the coordinates of anchor points (for components, it is the center component in the block; for labels, it is the pin that it attaches to), then calculate the offsets on the x-y axis, and represent the coordinates with respect to the anchor points.

Table 1 shows the comparison between different schematic representations. Code-L1 is the proposed code representation, while Code-L2 removes relative coordinates and uses absolute coordinates instead. Code-L3 further removes the pin name based wire connection of *connect_pins()*, and utilizes another function *add_new_wire* to directly draw wire segments to build connections. Samples of three different code representations are illustrated in Section 7.1.

3.3.2 ***Training of the Schematic Generation.*** As shown in Figure 2, our goal is to build a model to generate a schematic based on the user request and component information.

Dataset preparation. Thanks to the plentiful open-source PCB designs of IoT devices online, especially from *Sparkfun* [40], we first collect a comprehensive dataset over PCB designs of multiple types of IoT devices. Then we synthesize user requests using external multi-modal LLMs, which take the exported images and netlists of schematics and generate requests from the user's perspective. To model users with different background knowledge levels, we introduce two styles of users' requests to augment the dataset: concise and detailed. For the concise request, we assume the user has little knowledge of PCB design or only cares about high-level functionality; thus, the schematic is described with a summary of its function. For the detailed request, the user request includes the specific circuit components and connections to be included in the schematic. Examples of the two styles of requests can be found in Section 7.2.

We use GPT-oss-20B [1] as the base model and perform supervised fine-tuning for our schematic code generation task. To utilize the reasoning capability of the model, we augment the dataset with chain-of-thought (CoT) reasoning distilled from reasoning models, following the approach of prior work [7, 16, 17]. Specifically, we synthesize the thinking process by calling both the larger reasoning model GPT-oss-120B [1] and GPT-oss-20B itself, prompting them to generate the CoT reasoning that leads to the output from the given request.

Finally, we construct the supervised fine-tuning dataset $\mathcal{D}$ at the single-schematic granularity, where each entry corresponds to one complete circuit design instance. Each data sample follows a structured triplet format consisting of a system prompt, a user input, and a target assistant output, as illustrated in Listing 1:

Listing 1: Structure of the dataset

```
SYSTEM:{"role": "system", "content":""}
INPUT:{"role": "user", "content":""}
OUTPUT:{"role": "assistant", "content":"", "thinking":""}}
```

The `SYSTEM` field prompts the model regarding the role of a schematic designer and specifies high-level design rules and constraints. The `INPUT` field contains the user's natural-language design request as defined above. The `OUTPUT` field stores the ground-truth response of the code representation of the schematic, while the thinking attribute records the intermediate reasoning process that explains how the design intent is translated into concrete component selection, connection logic, and final code. Following that,

each training sample in $\mathcal{D}$ is represented as

$$x = [x^{\text{sys}}, x^{\text{in}}], \quad y = [y^{\text{think}}, y^{\text{code}}], \tag{1}$$

where x^{sys} denotes the global schematic design constraints, x^{in} denotes the user's natural-language design request, y^{think} denotes the structured intermediate reasoning trace, and y^{code} denotes the final executable schematic generation code.

Model training. We perform *supervised fine-tuning (SFT)* by optimizing the conditional likelihood of the full output sequence given the input. Since the `OUTPUT` field contains both the `thinking` process y^{think} and the schematic code y^{code}, the overall training objective naturally decomposes as

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{think}} + \mathcal{L}_{\text{code}} \tag{2}$$

with

$$\mathcal{L}_{\text{think}} = -\sum_{t=1}^{T} \log p_\theta(y_t^{\text{think}} \mid x, y_{<t}^{\text{think}}), \tag{3}$$

$$\mathcal{L}_{\text{code}} = -\sum_{t=1}^{T} \log p_\theta(y_t^{\text{code}} \mid x, y^{\text{think}}, y_{<t}^{\text{code}}). \tag{4}$$

where p_θ is the autoregressive GPT-oss model parameterized by θ, T is the total number of target tokens including both the `thinking` trace and the final schematic construction code, and $y_{<t}$ represents the previously generated tokens.

This joint optimization encourages the model to learn explicit domain-specific design reasoning (e.g., connection logic, component compatibility, and topology flow), while simultaneously producing fully executable schematic construction code that is consistent with such reasoning.

Instead of fully fine-tuning all model parameters, we adopt *Low-Rank Adaptation (LoRA)* to enable efficient domain specialization for PCB schematic generation. For each weight matrix $W \in \mathbb{R}^{d\times k}$ in the attention and feed-forward layers, the LoRA update is defined as

$$W' = W + \Delta W, \quad \Delta W = AB \tag{5}$$

where $A \in \mathbb{R}^{d\times r}$, $B \in \mathbb{R}^{r\times k}$, and $r \ll \min(d, k)$ is the low-rank dimension.

During training, the base weights W are frozen, and only the low-rank matrices A and B are optimized. The resulting conditional probability distribution becomes

$$p_\theta(y_t \mid x, y_{<t}) = p_{\theta_0+\Delta\theta}(y_t \mid x, y_{<t}), \tag{6}$$

where θ_0 denotes the frozen pretrained GPT-oss parameters and $\Delta\theta$ corresponds to the LoRA adapters. This significantly reduces the memory and training cost while preserving the general language modeling capability of the base model.

Schematic Generation Workflow. The schematic generation model in *IoTGen* can handle the generation of schematics at the functional block level. For a complex user request involving multiple functional blocks, we first call the external LLM API to split it into several sub-tasks of generating corresponding functional blocks, which are assigned with unified global labels. Eventually, these blocks are connected through the labels and merged into the final schematic that meets the user's request. Attributing to our code representation with the relative coordinates, the merge can be easily done by merging code snippets.

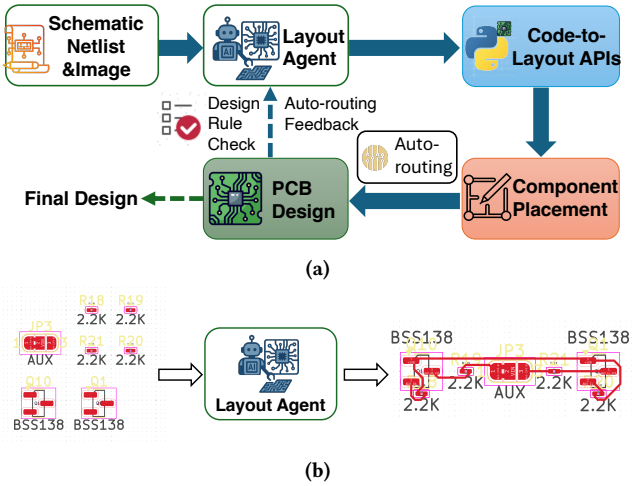


Figure 5: (a) Agentic layout workflow of *IoTGen* (b) A PCB layout example.

User Confirmation. Although the schematic generation module in *IoTGen* can generally produce designs that align with the user's request, inaccuracies may still arise at the level of detailed connections or component configurations. To address this, users are encouraged to review the generated schematic and provide targeted feedback or refined prompts, enabling iterative correction and improvement of the design.

3.4 PCB Agentic Layout

Given the schematic and netlist, the next step is to place the footprint of each component on the PCB board, draw actual metal wires to connect them, and make via holes if necessary. The output is the final board ready to be produced by the factory.

Conventional heuristic. Conventional algorithms in the domain of EDA rely on heuristic methods, such as the genetic algorithm [3], A-star algorithm [3], and maze-solving algorithm [30], which can give reliable wiring when component layout is determined. *Freerouting* [44], a useful tool which combines rip-up and reroute [27], maze routing, negotiated congestion [29], and heuristic optimization, has been widely used in many PCB EDA software, including *Eagle*, *LayoutEditor*, and *KiCad* . However, it still needs a human designer to determine the placement of all components before automatic wiring, which necessitates expertise and experience.

Programming abstractions. Aiming to fully automate the PCB layout process and embed it into the whole pipeline, we first introduce the following code-to-layout Python APIs,

```
layout = layout_api()
layout.place_fp(component_ref, x_pos, y_pos, orientation)
layout.auto_routing()
```

where *layout_api* is a Python class, *place_fp* operates on the *KiCad* software to place a footprint according to the assigned component reference, position, and orientation, and *auto_routing* performs auto-wiring based on *Freerouting*.

Multi-modal layout agent. On top of that, we design the multi-modal layout agent to edit the PCB board with a few rounds of iterations. It takes multiple sources of prompts: 1) Netlist file, which depicts all the nets of pins in the PCB board; 2) Image of the schematic design, which provides instructions on the placement on the board; (We remark that this input is important, for example, decoupling capacitors may share the same net of "VCC" and "GND", but they need to place near the corresponding chips.) 3) Feedback from *Freerouting* and design rule check (DRC). When finished auto-wiring, *Freerouting* reports the scores while solving the routing problems, and outputs a JSON summary describing routing statistics such as trace count, via count, net completion, total wire length, bend angles, and any remaining clearance violations, and DRC reports any violations such as shorts, clearance errors, unrouted nets, and footprint conflicts. Given the prompts above, the layout agent evaluates the current PCB layout with corresponding advice on improving it, and generates new versions of code. Through reasonable rounds of iterations, the agent can produce a complete PCB design that mimics the schematic design, which both follow the original user request. As the layout agent takes multi-modal information (image and text) as the input, it is challenging to fine-tune a multi-modal foundation model, which we leave for future work.

User Confirmation. The layout agent in *IoTGen* only takes DRC and *Freerouting* reports as feedback when iteratively improving the PCB layout. However, in practical scenarios, users may have custom needs, such as board size, RF constraints, etc. To this end, users can inject specific requirements, such as "Make the board size as small as possible.", or manually adjust the placement and wiring. As the whole system does not include RF simulation and feedback, the RF constraints injected in prompts cannot be guaranteed, which we leave to future work.

4 Evaluation

4.1 Experimental Setup

4.1.1 Dataset. To evaluate the semantic retrieval of PCB components and train the schematic generation model. We first collect 196 open-source designs from *Sparkfun* [40] as shown in Figure 6. Then, we divide them into similar-sized schematic blocks and form a comprehensive dataset that contains 2481 *KiCad* schematic designs, including code representations, user requests, and thinking processes. The total volume is quadrupled to 9924 samples using two styles of user request and chain-of-thought reasoning from two models as described in Section 3.3.2. The schematics span various types of functionality, covering microcontroller, analog modules, LED, power, storage, battery, USB communication, antenna, connectors, etc, with up to 39 symbols and 48 labels per design. Figure 6 shows some examples of *Sparkfun* devices, from which we grab the schematics in the dataset.

4.1.2 Training Setup. We choose the open-source model by OpenAI with 20 billion parameters *gpt-oss-20b* as the base model and train it with supervised finetuning and LoRA [18]. We adopt a learning rate of 8×10^{-4} with a cosine annealing scheduler and a warmup ratio of 0.01. Only the assistant loss is optimized during training, with the maximum token length of 13312. For parameter-efficient adaptation, we employ LoRA with a rank of 8 and a scaling factor of 16, where all linear layers are selected as target modules. We run training and inference on a single 80 GB Nvidia A100 GPU.

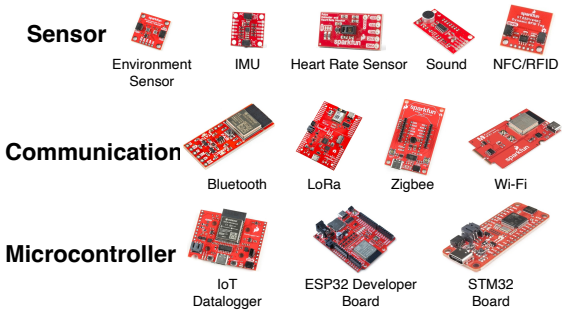


Figure 6: Examples of *Sparkfun* design in our dataset

4.2 Semantic Library Retrieval

We leverage the whole dataset for the evaluation of the module of the semantic library retrieval. The component list used in each semantic is taken as the ground truth.

Evaluation Metrics: We use the following two metrics to evaluate the retrieved components: (1) **Recall Accuracy** is defined as the ratio of correctly retrieved components to the ground-truth components for each user request. We use an LLM as a judge to determine whether two components are matched or functionally equivalent. To ensure the grading accuracy of LLM, we sampled a subset of the results and manually verified that LLM can output the correct judgement. (2) **Token Consumption**. We use the tokenizer in *BERT* [8] to calculate the token consumption in each retrieval process. Due to the large volume of component information and background knowledge needed to identify circuit components, we need commercial frontier LLMs for best performance. Therefore, we consider token consumption as an important metric for component retrieval cost. Locally deploying such models is either unrealistic or expensive.

Baselines: To demonstrate the efficiency of our proposed method, we compare it with the following three baselines: 1) *LLM Only*. This method first prompts the LLM with the whole list of *Lib ID* in *KiCad* component library to pick out the possible candidate list of *Lib ID*, then reads out the full information of all components belonging to these Lib IDs, which are prompted to the LLM again to select the final component list. 2) *IoTGen w/o Indexing*. This method directly asks LLM to analyze the possible functions (power, microcontroller, sensor) of the circuits according to the user request to get a candidate function list, then do the same vector search in *IoTGen* to get the final component list. 3) *Vector Search* directly performs vector search in the repository using the raw user request. We evaluate different methods using different models of *GPT-o4mini* [33], *GPT-5* [32], and *Grok-3* [46] regarding the defined metrics.

Results: Figure 7 presents the recall accuracy of different retrieval strategies across multiple LLMs. Our proposed semantic component retrieval method consistently outperforms both *IoTGen w/o Indexing* and *Vector Search*. Its median accuracy exceeds 90%, indicating robust performance across most test cases. In contrast, *Vector Search*

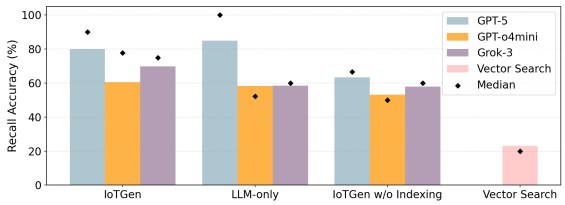


Figure 7: Recall accuracy with different component retrieval methods under various models

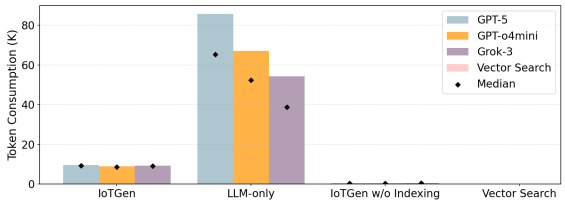


Figure 8: Token consumption with different component retrieval methods. *IoTGen* uses significantly fewer tokens, leading to lower cost and faster speed.

performs significantly worse, largely due to the challenge of high-level semantic interpretation discussed in Section 3.2. Although the proposed method achieves slightly lower accuracy than the *LLM Only* baseline, it requires only about 10% of the tokens consumed by *LLM Only*, as shown in Figure 8. This substantial reduction in token usage arises because *LLM Only* directly reads extensive *KiCad* component metadata from the repository, leading to exceptionally large input lengths.

Benefiting from the well-organized indexing tree, the token consumption of our method remains nearly constant across different models, whereas the cost of the *LLM Only* approach varies considerably with model architecture. Regarding recall accuracy, *GPT-5* delivers the highest overall retrieval accuracy, followed by *Grok-3*, while *GPT-o4mini* exhibits the weakest performance on this task.

4.3 Schematic Generation

To evaluate our schematic generation model, we randomly select 500 samples as the test set, using the remaining as the training set.

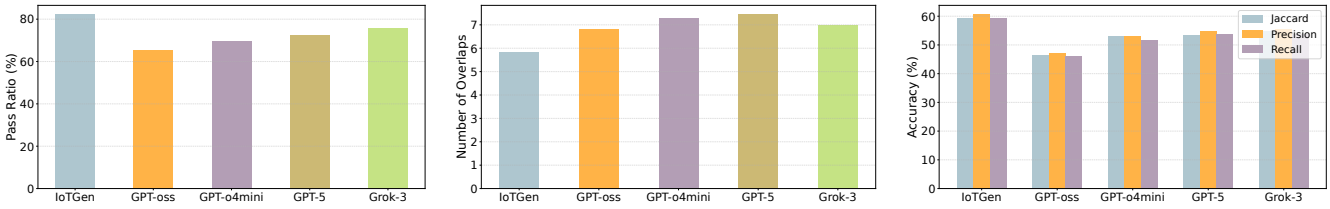
Evaluation metrics. We evaluate the performance of the schematic generation model of *IoTGen* from three perspectives: (1) **Valid Circuits**, determined by the ratio of generated code that can be successfully executed with no Python errors raised, which are typically triggered by incorrect arguments in the function calls, e.g., non-existing symbol reference or pin name, or coordinates out of range. (2) **Spatial Violation**, determined by the numbers of *spatial overlaps* among symbols, labels, and wires. In the code-to-schematic APIs, we assign a bounding box to each object. If there exist bounding box intersections, it is counted as an overlap. To consider the effect of pass ratio on the number of overlaps, we adjust the average number of overlaps by the pass ratio, represented by $\bar{n}_{weighted} = \frac{\bar{n}_{original}}{\text{pass ratio}}$. We use this metric to evaluate the readability of schematic designs and the spatial arrangement capability of different approaches. Having a bounding box violation does not mean the circuit is incorrect. (3) **Netlist Accuracy**, which examines symbol and wire connection accuracy by comparing the output netlist of model-generated schematic designs with the netlist of our ground truth schematic designs in the testing set. We first give the definition of the node $v_i = (\text{symbol}, \text{pin})$. In the netlist file, each node is assigned to a net where nodes from the same net are bound together. We can represent each net as a set $\mathbb{N} = \{v_1, v_2, \cdots, v_n\}$, where n is the total number of nodes in the net. Eventually, the connection revealed in the netlist file can be denoted by a set of nets $\mathbb{G} = \{\mathbb{N}_1, \mathbb{N}_2, \cdots, \mathbb{N}_m\}$, in which m is the total number of nets. Thus, the netlists of generated schematic and ground truth are turned into two sets of nets $\mathbb{G}_{gen}$ and $\mathbb{G}_{gt}$, which are used to calculate *Jaccard*, *Precision*, and *Recall*.

Table 2: Performance comparison of *gpt-oss-20b* fine-tuned with different representations

Method	Valid Circuits ↑	Spatial Violation ↓	Netlist Accuracy (%)↑		
	(Pass ratio%)	(Number of overlaps)	Jaccard	Precision	Recall
Code-L1 (*IoTGen*)	82.40	**5.83**	**59.48**	**60.74**	**59.31**
Code-L2	76.40	5.86	49.96	53.06	49.44
Code-L3	84.80	6.03	55.24	56.18	58.78
KiCad File	38.41	7.12	28.85	28.21	29.40

Baselines. To show the effects of different schematic representations after supervised fine-tuning, we compare the performance of our proposed presentation **Code-L1** with three baselines: 1) *gpt-oss-20b* model fine-tuned with the dataset using Code-L2 representation; 2) *gpt-oss-20b* model fine-tuned with the dataset using Code-L3 representation; 3) *gpt-oss-20b* fine-tuned with the dataset using the raw *KiCad* schematic file. Moreover, using the testing set of the proposed dataset as the benchmark, we compare the performance of our model with *gpt-oss-20b* base model, which is the vanilla model without finetuning, and three other recently released models with much larger model sizes, including *GPT-o4mini*, *GPT-5*, and *Grok-3*. We prompt these unmodified models with descriptions of different schematic representations and give one example usage of the representation.

Effectiveness of code representations. Table 2 reports the performance of finetuning the same base model, *gpt-oss-20b*, with datasets under different schematic representations. Our proposed Code-L1 representation achieves the best performance in terms of spatial violation and netlist accuracy, while maintaining a high pass ratio. Notably, Code-L1 representation leads to double the netlist accuracy compared to *KiCad* file representation, demonstrating its effectiveness in capturing essential design semantics. In comparison, *KiCad* file representation yields the lowest performance across all metrics, indicating the challenges LLMs face in generating complex and long schematic file formats. Code-L1 (used in*IoTGen*) achieves a slightly lower pass ratio than Code-L3, which is mainly due to the wire connection API error reporting: *connect_pins()* in Code-L1 checks pin name validity rigorously, while function *add_new_wire()* in Code-L3 does not check pin position are valid or not, which makes it easier to pass the execution.



(a) Valid Circuits (%) **(b) Spatial Violation** **(c) Netlist Accuracy (%)**

Figure 9: Schematic generation performance comparison of different models

Comparison of different models. We evaluate the performance of our schematic generation model in *IoTGen* versus other language models, including gpt-oss-20b base model, *GPT-o4mini*, *GPT-5*, and *Grok-3*. For models not fine-tuned on our dataset, we prompt them with descriptions of the schematic representation and provide one example usage of the representation. As shown in Figure 9, our model outperforms all other models across all metrics, despite being fine-tuned from a base model with only 20 billion parameters. Moreover, comparing the same model prompted by different representations, the performance of models prompted by the Code-L1 representation is superior in most metrics, which proves its efficiency for prompt engineering. The performance gap between the Code-L1 and Code-L2 is smaller than the result in Table 2. After checking the generated code, we find that large models often use relative coordinates for symbol placement and pin locations for wire connections, even if the example code doesn't adopt it.

Generalization. We remark that the *IoTGen* has the ability to generate novel schematics using unseen circuit components or chips that are not included in the training set. It proves that the model fine-tuned by the proposed code representation can learn the design logic of a specific type of function block, and it can be used to design various kinds of schematics based on the user's needs. We provide the visualization of two novel schematic examples in Figure 16 of Appendix 7.3, with two samples of successful examples in the test set in Fig. 15.

4.4 PCB Layout

After assigning footprints to all components in the schematics in our collected dataset, we can get the corresponding PCB board, where components are placed randomly without wires between them. Here, we focus on the complete PCB board of an IoT device, which includes all essential components from different functional blocks. We sample 20 boards from the dataset as the test benchmark.

Evaluation Metrics: We employ the *Layout Score* defined by *Freerouting* [44], a scale-invariant metric that evaluates routing quality across heterogeneous PCB designs. It normalizes the routing performance as

$$S_{\text{norm}} = 1000 \cdot \max\left(0,\ \frac{S}{S_{\max}}\right), \tag{7}$$

where $S_{\max} = N_{\max} \cdot \alpha$ denotes the ideal score achieved when all nets are routed without violations, and α is the penalty weight per unrouted net. The actual routing score subtracts violation- and resource-related penalties from this maximum:

$$S = S_{\max} - N_{\text{inc}} \cdot \alpha - N_{\text{clr}} \cdot \beta - N_{\text{bend}} \cdot \gamma - (L_{\text{trace}} \cdot \delta + N_{\text{via}} \cdot \eta), \tag{8}$$

where N_{inc} is the number of unrouted connections, N_{clr} the clearance violations, N_{bend} the routing bends, L_{trace} the total trace length, and N_{via} the via count. The coefficients β, γ, δ, and η control the respective penalty weights.

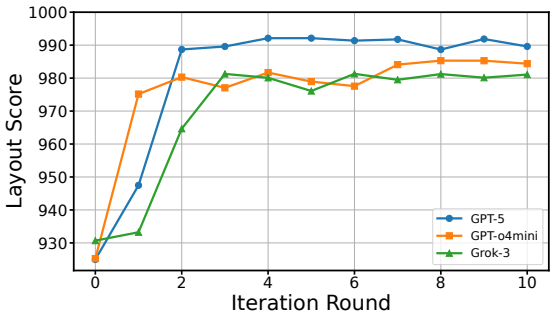


Figure 10: Layout scores with agentic iteration

Baselines: To illustrate the effectiveness of the designed iterative layout agent, we compare it with the two baselines below: 1) *Random Layout*: We directly do auto-wiring on the randomly placed PCB components converted from the schematic. 2) *Layout Agent w/o iteration*: We conduct only one-step layout with the agent using the netlist and schematic image without feedback from the auto-wiring and design rule check. We evaluate the layout agent using different models of *GPT-o4mini*, *GPT-5*, and *Grok-3* with respect to the metrics defined above.

Results: Figure 5b in Section 3.4 provides an illustrative outcome from the layout agent. Compared to the *Random Layout*, which simply piles up components according to their types, the layout agent in *IoTGen* can place components and draw wires following the correct circuit logic.

Figure 10 shows the variation of layout score against iteration rounds of the circuit in Figure 5b. This shows the agentic iteration of *IoTGen* brings clear score improvement, especially compared to 0 iterations, which is the random layout. The layout score of all models can converge after a few rounds with improvements up to 60. We can see that *GPT-5* has superior performance on this task, attributing to its advanced ability to analyze multi-modal input.

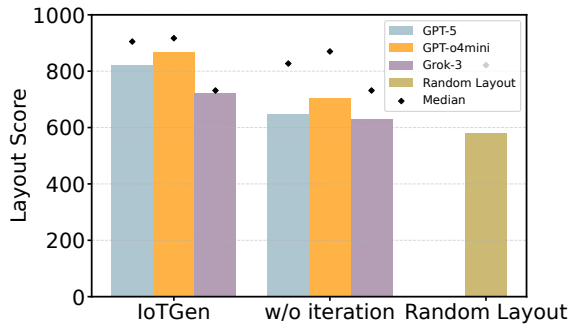


Figure 11: Comparison of mean layout score

Figure 11 shows that IoTGen's Layout Agent consistently outperforms both non-iterative routing and random layouts across all evaluated models. All LLM-driven agents (*GPT-5*, *GPT-o4mini*, and *Grok-3*) achieve substantially higher routing scores, indicating more complete routing, fewer violations, and cleaner topology. Among them, *GPT-o4mini* achieves the highest overall performance, demonstrating the strongest ability to improve PCB layouts through iterative refinement. *GPT-5* falls slightly behind, as it may automatically invoke smaller models with weaker reasoning ability on the layout, which constrains its performance on the complex task.

Random Layout suffers from a low layout score when handling a complex circuit with many components due to the small space and intersected connections. Comparing *IoTGen* and *Layout Agent w/o iteration*, we can conclude that relying on schematic input to do the layout is unreliable. By introducing feedback from DRC and auto-wiring, the layout agent better understands the constraints of PCB layout, consequently improving its performance on the task.

4.5 Case Study

In this section, we demonstrate how *IoTGen* can generate IoT devices for environment sensing and data collection, using a *multi-sensor module* PCB and textit CO2 monitoring device PCB as case studies. Given the user request and sub-tasks split by LLM, *IoTGen* first retrieves the components and then does schematic generation on each sub-task. Eventually, it merges all schematics and does PCB layout to get the final board.

4.5.1 Sensor Module. We first focus on designing a synthesized sensor module with available pins that can be directly plugged into a development board like *Arduino*. It should include multiple sensor functions according to the user's request.

User Request: *"I want to design a complete multi-sensor module that integrates pressure sensing, light sensing, a digital microphone, an IMU with gesture detection, a magnetometer, a full environmental sensor, an I2C connector interface, and a power-indicator LED."*

Conducting all sub-tasks, we can get a component list that satisfies the user request. As illustrated in Figure 12 (a), BMP280, BME680, and APDS-9301 are pressure, environmental, and light sensors, respectively. SPH0641LU4H-1 is a microphone, while ISM330DHCX is an IMU chip. There are also some decoupling capacitors on the board to filter out high-frequency noise. An LED is placed with a 10K resistor on the left to indicate the power. The PCB designs after layout and produced board are shown in Figure 12 (b)(c).

Verification. We program and test the key sensing functionalities of the generated sensor breakout board. The IMU gyroscope data (Figure 13a) shows clear, axis-specific responses to controlled manual perturbations, where distinct peaks are observed along the X, Y, and Z axes at different time instances, which validate the correct functionality and sensitivity of the inertial sensing pipeline. The temperature and humidity measurements (Figure 13b) exhibit smooth and stable temporal variations over 9 hours, demonstrating the reliable environmental sensing ability of the board. The absence of abrupt noise or discontinuities further confirms proper sensor integration and data acquisition. The results show that the system can simultaneously capture both dynamic motion signals and ambient environmental conditions with high fidelity.

4.5.2 CO2 Monitor Device. Next, we design a CO2 monitor device, which integrates multiple functional blocks including memory, microcontroller, USB-C port, power regulator, and CO2 sensor. It can read and store CO2 density in the environment when powered.

User Request: *"I want to design a complete CO2-monitoring device that can measure CO2 levels, process the data with a microcontroller with USB communication, and store the readings on a microSD card."*

Based on the component list retrieved, the final generated schematic is shown as Figure 12 (d), where an ESP32 chip is put in the center as the microcontroller. The whole circuit is powered on by the USB-C port through the interface of the CH340G and the voltage regulator AP2112K. It also has a micro-SD card to store the CO2 data read from SCD40-D-R2. Figure 12 (e)(f) depict the PCB design and the fabricated board.

Verification. We program the board and evaluate the generated CO2 monitoring device under two scenarios: (1) Validation Test (Figure 13c), where human breath was directed toward the sensor, and (2) Indoor Monitoring (Figure 13d), where the device continuously recorded CO2 levels in an indoor environment over 12 hours. The results in Figure 13 demonstrate that the generated IoT device can accurately capture and track ambient CO2 variations.

4.5.3 Time and Cost Efficiency. As shown in Figure 8, the token consumption of component retrieval for the two PCB boards above is around 10k, while it consumes 22k tokens per iteration on average for the PCB agentic layout process. Combining the three modules together, *IoTGen* can generate the PCBs shown in Figure 12 within 30 minutes with a token cost of approximately $1, while human designers may require hours, and even longer for non-experts to finish the same task.

5 Discussion

Scope of IoT Device. The dataset we collected in *IoTGen* is constructed from *Sparkfun* open-source designs, which primarily represent digital IoT boards with well-structured schematic patterns. While this choice provides high-quality, clean schematics for modeling, it restricts the diversity of IoT devices the model encounters. Notably, the dataset does not include RF front-end circuits or other high-frequency circuits where other considerations, such

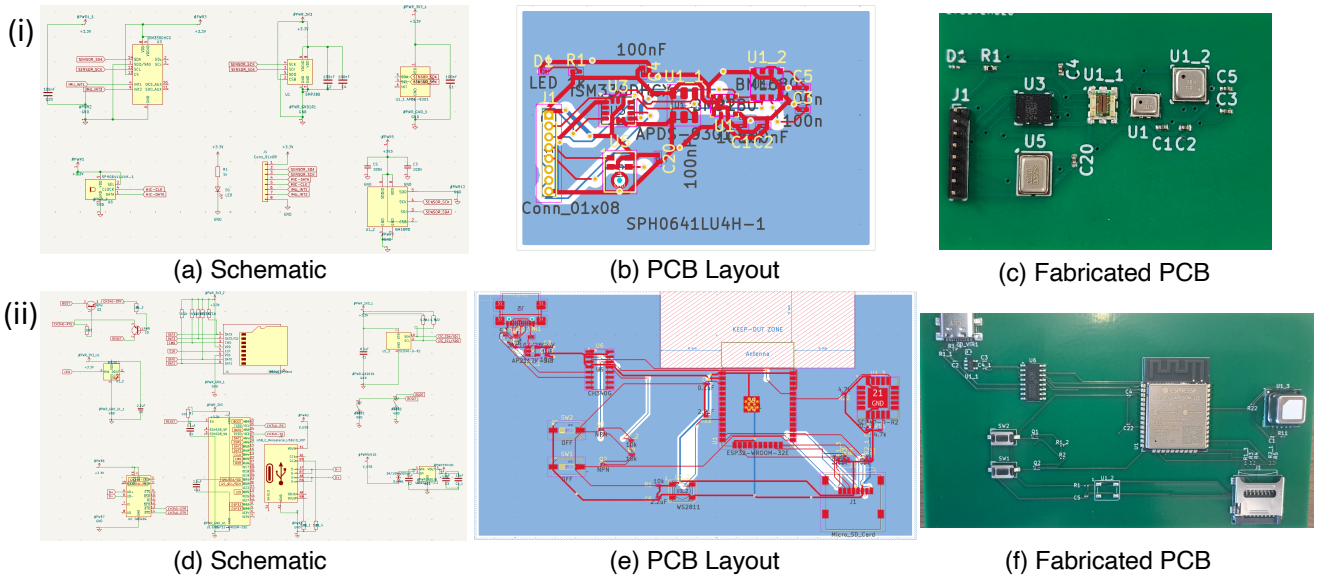
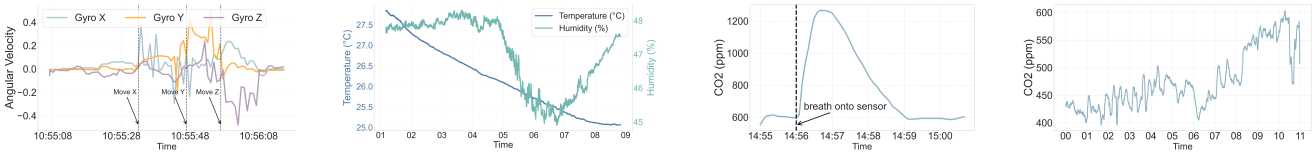


Figure 12: Case Study of (i) Synthesized Sensor Module (ii) CO2 Monitoring Device



(a) Case Study 1: ISM330DHCX Gyroscope **(b) Case Study 1: Temperature/Humidity Sensor** **(c) Case Study 2: Human breath onto the sensor.** **(d) Case Study 2: CO2 level monitoring for 12 hours**

Figure 13: Verification of Case Study

as impedance matching and board-stackup considerations, are essential. As a result, *IoTGen* currently treats PCB layout as a logical placement and connection task and does not reason about specific constraints of more advanced wireless devices integrating radios or high-speed interfaces, which may be addressed with models trained on specifically curated datasets and corresponding optimization goal of RF specifications. Future research can focus on a PCB generation system targeting advanced wireless devices.

Scale of PCB Design. *IoTGen* currently operates at the functional-block level, where each block contains a modest number of components. The agentic layout requires several iterations, with runtime growing alongside board size and component count. For large-scale PCBs, both LLM-based placement and the convergence of auto-routing algorithms such as Freerouting become increasingly time-consuming. Moreover, designing large-scale PCBs still requires external LLM APIs to decompose the task before invoking the schematic-generation model in *IoTGen*, incurring additional resource and time overhead. Following language model scaling laws [20], our schematic-generation model can potentially be improved by adopting base models with larger parameter counts and training on larger, more complex circuit datasets. Improving the scalability and efficiency of the agentic PCB design system is a worthwhile direction for future work.

6 Conclusion

We introduced *IoTGen*, an LLM-driven system that automates the IoT hardware design workflow from natural-language intent to PCB designs. *IoTGen* introduces semantic-rich programming abstractions, enabling a specialized language model to synthesize schematics with code. By combining semantic component retrieval, schematic generation, and an agentic layout procedure, *IoTGen* significantly reduces the expertise and manual effort required for IoT hardware development. Experiments show strong performance across component retrieval, schematic generation, and layout efficiency, with case studies validating its practicality for real IoT devices. *IoTGen* represents an emerging direction for research exploration and establishes a foundation for future research in developing an IoT hardware generation system on a larger scale and broader scope. We believe the concept of code-based agentic automation for IoT hardware leads to promising directions, such as self-evolving mobile systems, LLM-driven software-hardware co-design.

References

[1] Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, et al. 2025. gpt-oss-120b & gpt-oss-20b Model Card. arXiv:2508.10925 [cs.CL] https://arxiv.org/abs/2508.10925

[2] Altium Designer. 2025. PCB Design Software & Tools | Altium. https://www.altium.com/. Accessed: 2025-09-02.

[3] Tessy Badriyah, Fitri Setyorini, and Niyoko Yuliawan. 2016. The implementation of Genetic Algorithm and Routing Lee for PCB design optimization. In *2016 International Conference on Informatics and Computing (ICIC)*. 148–153. doi:10.1109/IAC.2016.7905706

[4] Cadence Design Systems. 2025. PCB Design Software | OrCAD X. https://www.cadence.com/en_US/home/tools/pcb-design-and-analysis/orcad.html. Accessed: 2025-09-02.

[5] Chen-Chia Chang, Yikang Shen, Shaoze Fan, Jing Li, Shun Zhang, Ningyuan Cao, Yiran Chen, and Xin Zhang. 2024. Lamagic: Language-model-based topology generation for analog integrated circuits. *arXiv preprint arXiv:2407.18269* (2024).

[6] Xi Chen, Julien Cumin, Fano Ramparany, and Dominique Vaufreydaz. 2024. Towards LLM-Powered Ambient Sensor Based Multi-Person Human Activity Recognition. In *2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)*. 609–616. doi:10.1109/ICPADS63350.2024.00085

[7] Xinghao Chen, Zhijing Sun, Wenjin Guo, Miaoran Zhang, Yanjun Chen, Yirong Sun, Hui Su, Yijie Pan, Dietrich Klakow, Wenjie Li, et al. 2025. Unveiling the key factors for distilling chain-of-thought reasoning. *arXiv preprint arXiv:2502.18001* (2025).

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805

[9] Zehao Dong, Weidong Cao, Muhan Zhang, Dacheng Tao, Yixin Chen, and Xuan Zhang. 2023. CktGNN: Circuit graph neural network for electronic design automation. *arXiv preprint arXiv:2308.16406* (2023).

[10] EAGLE. 2023. The Future of Autodesk EAGLE: Autodesk Fusion Electronics. https://www.autodesk.com/products/fusion-360/blog/future-of-autodesk-eagle-fusion-360-electronics/. Fusion Blog. Accessed: 2025-09-02.

[11] Xingyu Feng, Zehua Sun, Zhuangzhuang Chen, Chengwen Luo, Zhangbing Zhou, Victor C.M. Leung, and Weitao Xu. 2025. LLM-CoSen: Revisiting Collaborative Sensing With Large Language Models (LLMs). *IEEE Transactions on Mobile Computing* 24, 11 (2025), 11555–11567. doi:10.1109/TMC.2025.3583345

[12] Flux. 2026. Flux: Design PCBs with AI. https://www.flux.ai Accessed: 2026-04-06.

[13] Yonggan Fu, Yongan Zhang, Zhongzhi Yu, Sixu Li, Zhifan Ye, Chaojian Li, Cheng Wan, and Yingyan Celine Lin. 2023. Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[14] Jian Gao, Weidong Cao, Junyi Yang, and Xuan Zhang. 2025. AnalogGenie: A generative engine for automatic discovery of analog circuit topologies. *arXiv preprint arXiv:2503.00205* (2025).

[15] Yi Gao, Kaijie Xiao, Fu Li, Weifeng Xu, Jiaming Huang, and Wei Dong. 2024. ChatIoT: Zero-code Generation of Trigger-action Based IoT Programs. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 8, 3, Article 103 (Sept. 2024), 29 pages. doi:10.1145/3678585

[16] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[17] Namgyu Ho, Laura Schmid, and Se-Young Yun. 2022. Large language models are reasoning teachers. *arXiv preprint arXiv:2212.10071* (2022).

[18] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL] https://arxiv.org/abs/2106.09685

[19] Chun-Yen Huang, Hsuan-I Chen, Hao-Wen Ho, Pei-Hsin Kang, Mark Po-Hung Lin, Wen-Hao Liu, and Haoxing Ren. 2025. Netlistify: Transforming Circuit Schematics into Netlists with Deep Learning. In *2025 ACM/IEEE Symposium on Machine Learning for CAD (MLCAD)*.

[20] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. arXiv:2001.08361 [cs.LG] https://arxiv.org/abs/2001.08361

[21] KiCad. 2025. KiCad. https://www.kicad.org/. Accessed: 2025-09-02.

[22] Yao Lai, Sungyoung Lee, Guojin Chen, Souradip Poddar, Mengkang Hu, David Z Pan, and Ping Luo. 2025. Analogcoder: Analog circuit design via training-free code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 379–387.

[23] Haiyun Li, Jixin Zhang, Ning Xu, and Mingyu Liu. 2023. FanoutNet: A Neuralized PCB Fanout Automation Method Using Deep Reinforcement Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 37, 7 (Jun. 2023), 8554–8561. doi:10.1609/aaai.v37i7.26030

[24] Jindong Li, Lianrong Chen, Bin Yang, Jiadong Zhu, Ying Wang, Yuzhe Ma, and Menglin Yang. 2026. PCB-Bench: Benchmarking LLMs for Printed Circuit Board Placement and Routing. In *International Conference on Learning Representations (ICLR)*.

[25] Kaiwei Liu, Bufang Yang, Lilin Xu, Yunqi Guo, Neiwen Ling, Zhihe Zhao, Guoliang Xing, Xian Shuai, Xiaozhe Ren, Xin Jiang, and Zhenyu Yan. 2024. Poster Abstract: Tasking Heterogeneous Sensor Systems with LLMs. In *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems* (Hangzhou, China) *(SenSys '24)*. Association for Computing Machinery, New York, NY, USA, 901–902. doi:10.1145/3666025.3699428

[26] Wen-Hao Liu, Anthony Agnesina, and Haoxing Mark Ren. 2024. Challenges for Automating PCB Layout. In *Proceedings of the 2024 International Symposium on Physical Design* (Taipei, Taiwan) *(ISPD '24)*. Association for Computing Machinery, New York, NY, USA, 91–92. doi:10.1145/3626184.3635285

[27] Yang Liu, Haigang Yang, Wei Yu, Xiuhai Cui, and Juan Huang. 2014. An FPGA timing routing algorithm based on PathFinder and rip-up and retry approach. *Journal of Computer-Aided Design & Computer Graphics* 26, 1 (2014), 138–145.

[28] Ryoga Matsuo, Stefan Uhlich, Arun Venkitaraman, Andrea Bonetti, Chia-Yu Hsieh, Ali Momeni, Lukas Mauch, Augusto Capone, Eisaku Ohbuchi, and Lorenzo Servadei. 2024. Schemato–An LLM for Netlist-to-Schematic Conversion. *arXiv preprint arXiv:2411.13899* (2024).

[29] Larry McMurchie and Carl Ebeling. 1995. PathFinder: a negotiation-based performance-driven router for FPGAs. In *Proceedings of the 1995 ACM Third International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '95)*. Association for Computing Machinery, New York, NY, USA, 111–117. doi:10.1145/201310.201328

[30] M. Murakami and N. Honda. 2000. A maze-running algorithm using fuzzy set theory for routing methods of printed circuit boards. In *Ninth IEEE International Conference on Fuzzy Systems. FUZZ- IEEE 2000 (Cat. No.00CH37063)*, Vol. 2. 985–988 vol.2. doi:10.1109/FUZZY.2000.839177

[31] Fionn Murtagh and Pedro Contreras. 2012. Algorithms for hierarchical clustering: an overview. *Wiley interdisciplinary reviews: data mining and knowledge discovery* 2, 1 (2012), 86–97.

[32] OpenAI. 2025. GPT-5. https://openai.com/index/introducing-gpt-5/. Introducing GPT-5.

[33] OpenAI. 2025. GPT-o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/. Introducing OpenAI o3 and o4-mini.

[34] Xiaomin Ouyang and Mani Srivastava. 2024. LLMSense: Harnessing LLMs for High-level Reasoning Over Spatiotemporal Sensor Traces. arXiv:2403.19857 [cs.AI] https://arxiv.org/abs/2403.19857

[35] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. http://arxiv.org/abs/1908.10084

[36] Zhiwei Ren, Junbo Li, Minjia Zhang, Di Wang, Xiaoran Fan, and Longfei Shangguan. 2025. *Toward Sensor-In-the-Loop LLM Agent: Benchmarks and Implications.* Association for Computing Machinery, New York, NY, USA, 254–267. https://doi.org/10.1145/3715014.3722082

[37] Leming Shen, Qiang Yang, Xinyu Huang, Zijing Ma, and Yuanqing Zheng. 2025. GPIoT: Tailoring Small Language Models for IoT Program Synthesis and Development. arXiv:2503.00686 [cs.SE] https://arxiv.org/abs/2503.00686

[38] Leming Shen, Qiang Yang, Yuanqing Zheng, and Mo Li. 2025. Autoiot: Llm-driven automated natural language programming for aiot applications. In *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*. 468–482.

[39] Siemens Digital Industries Software. 2025. Design automation. https://eda.sw.siemens.com/en-US/pcb/engineering-productivity-and-efficiency/design-automation/. Accessed: 2025-09-03.

[40] SparkFun Electronics. 2025. SparkFun Electronics. https://www.sparkfun.com/ Accessed: 2025-09-13.

[41] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems* 29, 3 (2024), 1–31.

[42] Yidong Tian, Andrew J. Forsyth, Zhuoru Li, and Cheng Zhang. 2021. A Component Manipulation Algorithm to Enable Design Automation of Power Electronic PCBs. In *2021 IEEE Design Methodologies Conference (DMC)*. 1–6. doi:10.1109/DMC51747.2021.9529938

[43] Yidong Tian, Andrew J. Forsyth, Zhuoru Li, and Cheng Zhang. 2022. Automatic Layout Design for Power Electronics PCBs. In *2022 IEEE Energy Conversion Congress and Exposition (ECCE)*. 1–6. doi:10.1109/ECCE50734.2022.9947957

[44] Nikolaus Widiger. 2025. FreeRouting – Free and Open Source PCB Autorouter. https://freerouting.org/. Accessed: 2025-11-26.

[45] Haoyuan Wu, Zhuolun He, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. 2024. Chateda: A large language model powered autonomous agent for eda. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 10 (2024), 3184–3197.

[46] xAI. 2025. Grok 3. https://x.ai/news/grok-3. Grok 3 Beta — The Age of Reasoning Agents.

[47] Haohang Xu, Chengjie Liu, Qihang Wang, Wenhao Huang, Yongjian Xu, Weiyu Chen, Anlan Peng, Zhijun Li, Bo Li, Lei Qi, Jun Yang, Yuan Du, and Li Du. 2025. Image2Net: Datasets, Benchmark and Hybrid Framework to Convert Analog Circuit Diagrams into Netlists. arXiv:2508.13157 [cs.AR] https://arxiv.org/abs/2508.13157

[48] Yutaro Yamada, Yihan Bao, Andrew K. Lampinen, Jungo Kasai, and Ilker Yildirim. 2024. Evaluating Spatial Understanding of Large Language Models. arXiv:2310.14540 [cs.CL] https://arxiv.org/abs/2310.14540

[49] Huanqi Yang, Mingzhe Li, Mingda Han, Zhenjiang Li, and Weitao Xu. 2024. EmbedGenius: Towards Automated Software Development for Generic Embedded IoT Systems. arXiv:2412.09058 [cs.SE] https://arxiv.org/abs/2412.09058

[50] Cong Zhang, Huilin Jin, Jienan Chen, Jinkuan Zhu, and Jinting Luo. 2020. A Hierarchy MCTS Algorithm for The Automated PCB Routing. In *2020 IEEE 16th International Conference on Control & Automation (ICCA)*. 1366–1371. doi:10.1109/ICCA51439.2020.9264558

7 Appendix

7.1 Example of Schematic Design and Code Representations

Fig. 14 shows the PCB schematic design of the voltage regulation module and an LED indicator attached. along with three different levels of code representations of the given schematic.

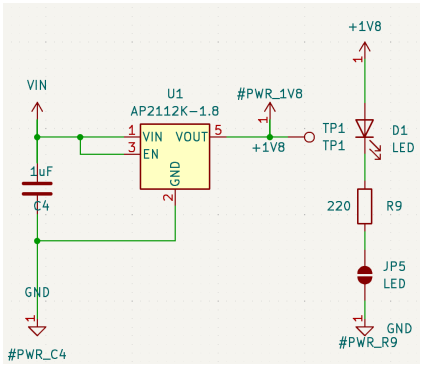


Figure 14: An example schematic designed in *KiCad*

Listing 2: Level 1 Representation

```
# Auto-generated schematic symbols
import sys
import os

# Get project path and import the Kicad schematic interface
PROJECT_PATH = os.environ['PROJECT_PATH']
sys. path.append(PROJECT_PATH)
from modules.kicad_sch_interface import *

### Placing center symbol 1 : Regulator_Linear:AP2112K-1.8###
center_x_1, center_y_1 = 120.650, 104.590
add_schematic_symbol(symbol_lib="Regulator_Linear",
      symbol_name="AP2112K-1.8", pos_x=center_x_1, pos_y=center_y_1,
      reference="U1", value="AP2112K-1.8", rotation=0, mirror="None")

### Placing other symbols in the Schematic with respect to the
      center symbol 1###
add_schematic_symbol(symbol_lib="power", symbol_name="VAA",
      pos_x=center_x_1 + (-20.32), pos_y=center_y_1 + (5.08),
      reference="#PWR1", value="VIN", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Device", symbol_name="C",
      pos_x=center_x_1 + (-20.32), pos_y=center_y_1 + (-5.08),
      reference="C4", value="1uF", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="power", symbol_name="GND",
      pos_x=center_x_1 + (-20.32), pos_y=center_y_1 + (-24.13),
      reference="#PWR_C4", value="GND", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="power", symbol_name="+1V8",
      pos_x=center_x_1 + (13.97), pos_y=center_y_1 + (5.08),
      reference="#PWR_1V8", value="+1V8", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Connector",
      symbol_name="TestPoint", pos_x=center_x_1 + (16.51),
      pos_y=center_y_1 + (2.54), reference="TP1", value="TP1",
      rotation=270, mirror="x")

### Placing all global labels in the Schematic and connecting them
      to the neighbor pin ###

### Connecting all wires in the Schematic ###
# Connecting #PWR_1V8 pin +1V8 (Pin ID 1 -- Name +1V8) to TP1 pin
      TP1 (Pin ID 1 -- Name TP1)
connect_pins("#PWR_1V8", "+1V8", "TP1", "TP1")
# Connecting #PWR1 pin VIN (Pin ID 1 -- Name VIN) to C4 pin 1 (Pin
      ID 1 -- Name None)
connect_pins("#PWR1", "VIN", "C4", "1")
# Connecting U1 pin VOUT (Pin ID 5 -- Name VOUT) to TP1 pin TP1
      (Pin ID 1 -- Name TP1)
connect_pins("U1", "VOUT", "TP1", "TP1")
# Connecting U1 pin VIN (Pin ID 1 -- Name VIN) to U1 pin EN (Pin ID
      3 -- Name EN)
connect_pins("U1", "VIN", "U1", "EN")
# Connecting C4 pin 2 (Pin ID 2 -- Name None) to #PWR_C4 pin 1 (Pin
      ID 1 -- Name None)
connect_pins("C4", "2", "#PWR_C4", "1")
# Connecting #PWR1 pin VIN (Pin ID 1 -- Name VIN) to U1 pin VIN
      (Pin ID 1 -- Name VIN)
connect_pins("#PWR1", "VIN", "U1", "VIN")
# Connecting C4 pin 2 (Pin ID 2 -- Name None) to U1 pin 2 (Pin ID 2
      -- Name None)
connect_pins("C4", "2", "U1", "2")

### Placing center symbol 2 : Jumper:SolderJumper_2_Open###
center_x_2, center_y_2 = 148.590, 86.810
add_schematic_symbol(symbol_lib="Jumper",
      symbol_name="SolderJumper_2_Open", pos_x=center_x_2,
      pos_y=center_y_2, reference="JP5", value="LED", rotation=270,
      mirror="None")

### Placing other symbols in the Schematic with respect to the
      center symbol 2###
add_schematic_symbol(symbol_lib="power", symbol_name="+1V8",
      pos_x=center_x_2 + (0.0), pos_y=center_y_2 + (31.75),
      reference="#PWR_1V1", value="+1V8", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Device", symbol_name="LED",
      pos_x=center_x_2 + (0.0), pos_y=center_y_2 + (21.59),
      reference="D1", value="LED", rotation=90, mirror="None")
add_schematic_symbol(symbol_lib="Device", symbol_name="R",
      pos_x=center_x_2 + (0.0), pos_y=center_y_2 + (10.16),
      reference="R9", value="220", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="power", symbol_name="GND",
      pos_x=center_x_2 + (0.0), pos_y=center_y_2 + (-6.35),
      reference="#PWR_R9", value="GND", rotation=0, mirror="None")

### Placing all global labels in the Schematic and connecting them
      to the neighbor pin ###

### Connecting all wires in the Schematic ###
# Connecting R9 pin 2 (Pin ID 2 -- Name None) to JP5 pin A (Pin ID
      1 -- Name A)
connect_pins("R9", "2", "JP5", "A")

# Connecting JP5 pin B (Pin ID 2 -- Name B) to #PWR_R9 pin 1 (Pin
      ID 1 -- Name None)
connect_pins("JP5", "B", "#PWR_R9", "1")

# Connecting D1 pin K (Pin ID 1 -- Name K) to R9 pin 1 (Pin ID 1 --
      Name None)
connect_pins("D1", "K", "R9", "1")

# Connecting #PWR_1V1 pin +1V8 (Pin ID 1 -- Name +1V8) to D1 pin A
      (Pin ID 2 -- Name A)
connect_pins("#PWR_1V1", "+1V8", "D1", "A")

write_out_all_wires()
```

7.2 Example of the request

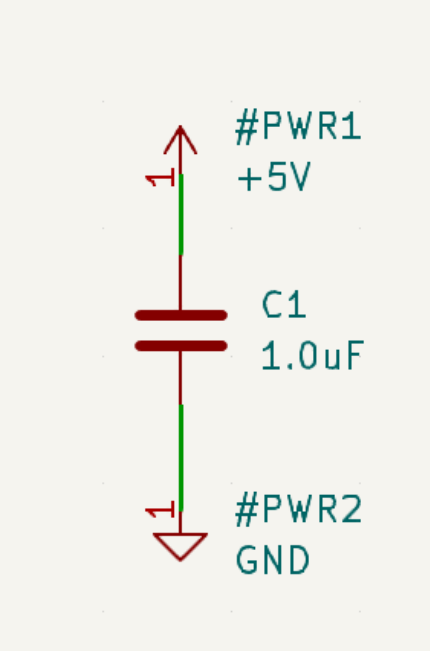
Listing 3: Concise style of request

```
{"messages": [{"role": "user", "content": "I want a 1.8V regulated
      supply from VIN using an AP2112K LDO, with a test point on the
      1.8V rail and a solder-jumper-selectable LED indicator.\n"}]}
```

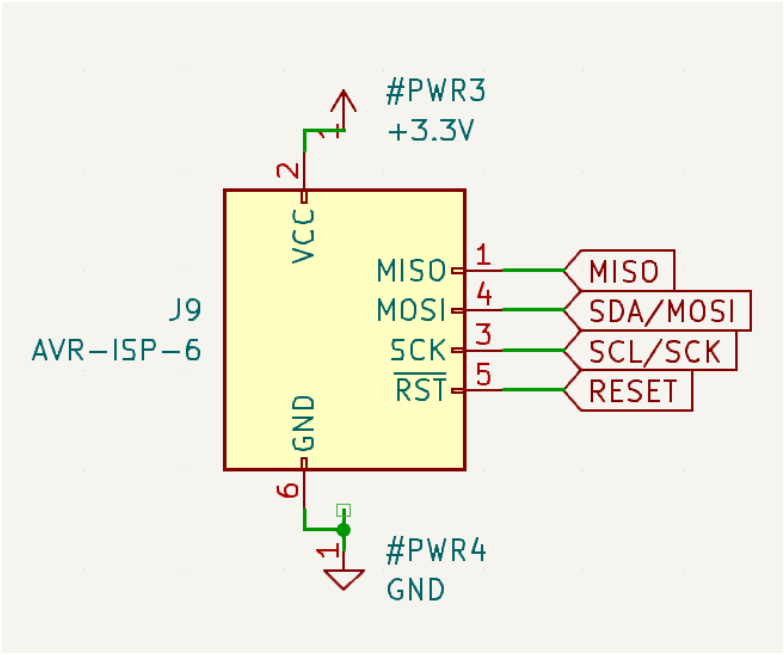
Listing 4: Detailed style of request

```
{"messages": [{"role": "user", "content": "I'd like you to design a
      small 1.8 V power sub-circuit with an onboard status LED and a
      test point, essentially mirroring the connectivity below. This
      will sit on a bigger board and just needs to take VIN and
      produce +1V8, with an optional solder-jumper to let me
      enable/disable the LED indicator.\n\nWhat I want, by parts and
      nets\n- Rails/labels: VIN, +1V8, GND. Please use these exact
      net names. There are also two internal nets created by the LED
      chain: Net-(D1-K) and Net-(JP5-A).\n- 1x LDO regulator: U1 =
      AP2112K-1.8 (SOT-23-5). This should generate the +1V8 rail
      from VIN. EN is tied to VIN so the regulator is on whenever
      VIN is present. NC pin is left unconnected.\n- 1x input
      capacitor: C4 = 1 uF, unpolarized, from VIN to GND (input
      decoupling for the LDO).\n- 1x LED power indicator: D1 = LED
      (generic symbol), lit when +1V8 is present and the solder
      jumper is closed.\n- 1x LED series resistor: R9 = 220 Ohm.\n-
      1x solder jumper: JP5 = SolderJumper_2_Open (default open).
      When shorted, it completes the LED return to GND.\n- 1x test
      point: TP1 = TestPoint on the +1V8 rail.\n\nExact connectivity
      (pin-by-pin)\n- U1 (AP2112K-1.8)\n - Pin 1 VIN -> net VIN.\n -
      Pin 2 GND -> net GND.\n - Pin 3 EN -> net VIN (LDO enabled
      whenever VIN is present).\n - Pin 4 NC -> leave unconnected.\n
      - Pin 5 VOUT -> net +1V8.\n- C4 (1 uF)\n - Pin 1 -> net VIN.\n
      - Pin 2 -> net GND.\n- D1 (LED)\n - Pin 2 A (anode) -> net
      +1V8.\n - Pin 1 K (cathode) -> net Net-(D1-K).\n- R9 (220
      Ohm)\n - Pin 1 -> net Net-(D1-K) (from LED cathode).\n - Pin 2
      -> net Net-(JP5-A).\n- JP5 (SolderJumper_2_Open)\n - Pin 1 A
      -> net Net-(JP5-A) (from R9).\n - Pin 2 B -> net GND.\n- TP1
      (TestPoint)\n - Pin 1 -> net +1V8.\n\nFunctional intent\n- The
      AP2112K-1.8 takes VIN (2.5–6 V capable per datasheet) and
      produces a regulated +1V8 rail on U1 pin 5.\n- C4 provides
      input decoupling between VIN and GND.\n- The LED D1 is wired
      from +1V8 (anode) through the LED and R9 to the A pad of JP5.
      When JP5 is shorted (A to B), the chain returns to GND and the
      LED lights, indicating +1V8 is present. With JP5 left open,
      the LED is disabled/off.\n- TP1 gives me an easy probe point
      for the +1V8 rail.\n- U1.EN is tied to VIN so no separate
      enable control is required; the regulator is active whenever
      VIN is applied.\n- U1.NC remains unconnected as
      specified.\n\nDeliverables/notes\n- Keep the reference
      designators and values exactly as above: U1 AP2112K-1.8, C4 1
      uF, D1 LED, R9 220 Ohm, JP5 SolderJumper_2_Open, TP1
      TestPoint.\n- Use the AP2112K-1.8 in SOT-23-5 as the
      footprint.\n- Net labels must be: VIN, +1V8, GND (plus the
      internal nets named by the tool for the LED chain).\n- Default
      BOM should reflect JP5 as a solder jumper (open by
      default).\n\nThat's the full requirement; please base the
      schematic on this connectivity so it matches exactly.\n"}]}
```

7.3 Examples of generation

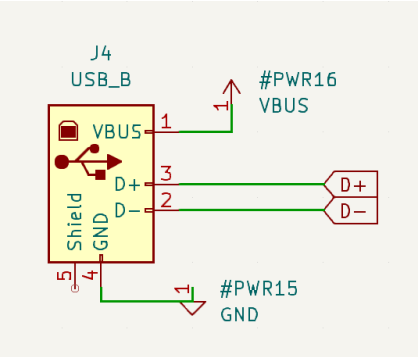


(a) User Request–"I want a 1uF capacitor connected between +5V and GND for power supply decoupling."

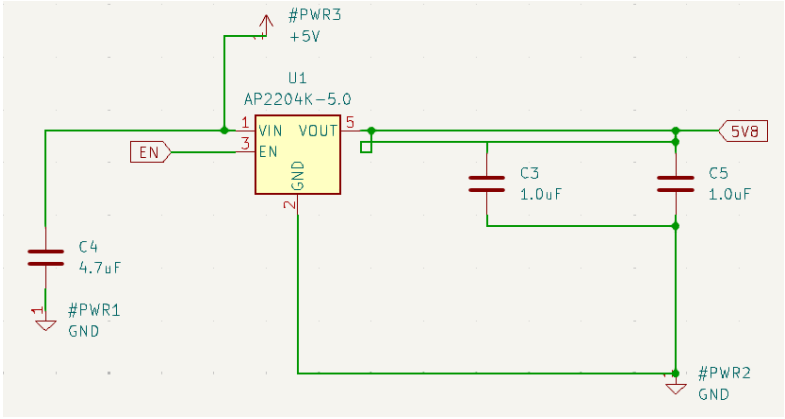


(b) User Request–"I want a 3.3V AVR ISP-6 programming header exposing MISO, MOSI/SDA, SCK/SCL, RESET, VCC, and GND to program a microcontroller."

Figure 15: Examples of using *IoTGen* to generate schematic based on users' requests



(a) Novel User Request–"I would like to add a USB-B connector interface in the schematic, exporting two labels, namely D+ and D-."



(b) Novel User Request–"I would like a voltage regulator module with an 5V output, using AP2204K."

Figure 16: Examples of using *IoTGen* to generate schematic based on users' requests over unseen chips.