SchGen: PCB Schematic Generation with Semantic-Grounded Code Representations

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

Printed circuit board (PCB) schematic design defines nearly all electronic hardware, but it remains manual and expertise-intensive. While generative AI has advanced digital and analog IC design, PCB schematic generation from natural-language intent is largely unexplored. This paper presents SchGen, the first large language model that generates editable PCB schematics from natural-language requests. The key challenge lies in the lack of an LLM-suited representation and a large-scale dataset. Current schematic formats are dominated by verbose, tool-specific syntax and geometry-heavy descriptions, making them difficult to generate reliably. We introduce a semantically grounded code representation that encodes schematic editing primitives with relative placement and pin-name-based wiring, transforming a geometry-driven generation problem into a semantics-driven matching task amenable to LLMs. We further construct a large-scale dataset of PCB schematics paired with user prompts via a human–agent collaborative pipeline that converts open-source hardware designs into our representation. Experiments show that SchGen significantly outperforms alternative representations and even larger general-purpose LLMs on wire connectivity accuracy and functional correctness. Our results highlight the critical role of representation design in enabling generative models for complex hardware design tasks. The source code is available at https://github.com/microsoft/SchGen.

1 Introduction

Printed circuit boards (PCBs) are foundational to nearly all electronic hardware, from consumer electronics to Internet-of-Things devices and embodied AI hardware. As new applications proliferate, the demand for customized PCB designs is accelerating, while the design workflow remains largely manual, requiring substantial domain expertise to operate electronic design automation (EDA) tools. Given a user's design request, engineers create the PCB schematic by selecting circuit components (symbols) and specifying their interconnections (wires), then export a netlist that guides subsequent PCB layout and fabrication. Among these stages, schematic design is the first and most critical. It defines the system architecture, but remains the least automated due to its vast design space and heavy reliance on domain knowledge. In this work, we aim to automate the PCB schematic design process by generating schematics from user prompts using large language models (Figure 1).

*Ruichun Ma is the corresponding author.

Preprint.

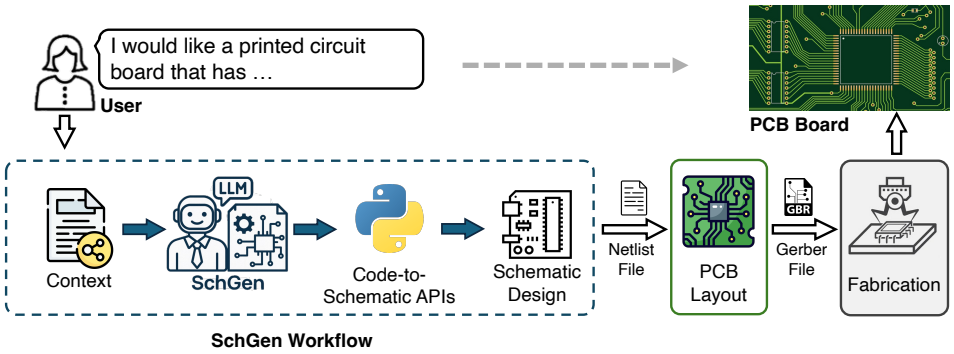


Figure 1: Overview of automated PCB design workflow. Based on the user request, SchGen generates a PCB schematic design using custom code representations, which is then converted to a netlist for PCB layout and fabrication.

Recently, generative models have shown promising capabilities for hardware design. However, generating PCB schematic designs based on user requests is largely unexplored. For digital integrated circuits, large language models (LLMs) have been studied to generate Boolean logic specified in high-level hardware description languages such as Verilog, VHDL Wu et al. [2024], Thakur et al. [2024], Fu et al. [2023]. On the other hand, analog ICs depend on circuit topology to achieve desired performance, leading to graph-structured representations and generation Dong et al. [2023], Lai et al. [2025], Chang et al. [2024]. Despite their success, prior works do not readily extend to our task. A PCB schematic connects diverse components, including various kinds of digital and analog IC components, passive elements (e.g., resistors and capacitors), and connectors. The vast range of components and complex wiring connections among heterogeneous components are not captured by prior work. Moreover, unlike specific digital logic or analog circuit performance targets, PCB schematics are driven by high-level functional requirements in natural language.

We present a new learning task of PCB schematic generation from natural language. Given a user's textual functional request, the model produces an editable schematic with both correct connectivity and a readable spatial layout. To tackle this task, we present SchGen, a large language model for generating PCB schematics based on natural-language prompts as shown in Figure 1. We address two critical challenges: (i) **efficient representation**. Unlike digital and analog ICs that have mature design representation, PCB schematic design is traditionally a GUI-centric visual workflow involving selecting, placing, and connecting components. As shown in Section 2, existing representations are poorly suited. Raw schematic files, while textual, are dominated by verbose, tool-specific, and version-specific metadata, making LLM generation prone to formatting errors and unusable, while image-based representations are not directly editable or machine-readable. Moreover, schematic design requires coherent component placement and readable wiring, while existing language models struggle to satisfy these requirements. (ii) **data scarcity**. Although many open-source hardware designs are publicly available, they appear in heterogeneous formats and are often released as images for human viewing, which makes it difficult to reuse or parse for training.

To bridge the gaps, we introduce a semantic-grounded code representation that transforms the schematic generation task from absolute-geometry prediction to semantics-driven matching. We define a compact set of editing primitives mirroring how experts draw schematics, including adding symbols, connecting pins, and querying pin locations, which strips redundant metadata and enables efficient learning. For spatial reasoning, our APIs use local coordinate systems with relative offsets from anchor components and connect wires by semantic pin names (e.g., VCC, TXD) rather than absolute coordinates, reducing the spatial reasoning burden on LLMs.

To tackle data scarcity, we develop an agentic pipeline that replicates open-source PCB designs as editable *KiCad* schematics, synthesizes user requests, and converts each design into executable Python code. Built on our representation and dataset, we finetune GPT-oss-20b to obtain SchGen.

We evaluate SchGen performance on a comprehensive set of metrics spanning design validity, spatial layout quality, netlist accuracy, and expert verification. Results confirm that our proposed Code-L1 representation achieves 82% valid circuit rate and 60.5% expert-verified functional correctness, compared to only 32% valid circuits from the raw *KiCad* file baseline. In comparison with frontier LLMs

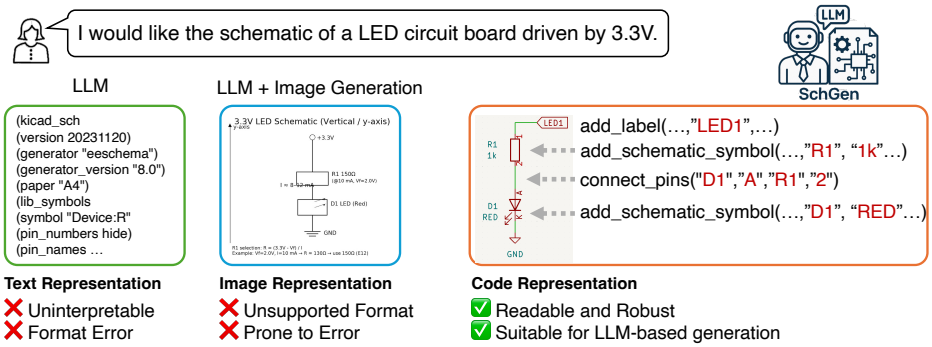


Figure 2: Comparison of schematic representations for LLM-based generation. Raw schematic files contain verbose metadata prone to formatting errors, while generated images are not machine-readable. Our code representation encodes editing primitives and pin-level semantics, transforming geometry prediction to a semantics-driven task amenable to LLM training.

prompted with the same APIs, including GPT-5.2, SchGen achieves the highest performance despite having only 20B parameters. We further validate generalization on an unseen out-of-distribution test set from GitHub projects, where SchGen matches the netlist accuracy of GPT-5.2.

To summarize, we make the following contributions: (1) We present the new task of PCB schematic generation from natural language and develop SchGen, the first LLM that generates editable PCB schematics from user prompts. (2) We propose a semantic-grounded code representation that abstracts schematic design into structured editing operations with relative placement and pin-level connectivity to facilitate semantics-driven matching. (3) We construct a large-scale dataset of PCB schematics paired with user prompts via a human–agent collaborative pipeline that converts open-source designs into executable code representations. (4) Experiments show that SchGen achieves high functional correctness rates and outperforms frontier LLMs with much larger parameter sizes.

2 Preliminaries and Related Works

2.1 PCB Schematic Design Process

PCB schematic design process conventionally depends on electronic design automation (EDA) software, including proprietary tools like Altium Designer [2025], Cadence Design Systems [2025], EAGLE [2023], and open-source alternatives like KiCad [2025]. While these tools support similar schematic content and editing operations, they rely on different file formats. Throughout this paper, we use the *KiCad* schematic format because it is open source.

A schematic contains three main item types: (1) *component symbols* representing physical circuit components such as chips, resistors, capacitors, (2) *power symbols* representing rails such as `VCC` for power source, GND for ground, (3) *net labels* that represent named net connections. Labels with the same name are treated as electrically connected, and are commonly used to define module interfaces so that modules connect automatically via shared label names. Each component symbol has one or more pins corresponding to the part's physical terminals; power symbols and net labels typically expose a single pin for wiring. We reference pins by an ID or a concise name (e.g., `VCC`, `GND`, `GPIO1`), and draw wires between pins to specify electrical connections.

The schematic design workflow typically starts by selecting the required symbols, then placing them to form a readable layout, and finally wiring the pins to realize the intended circuit. This process is critical as it decides hardware composition and electrical connections between every component, but it is also tedious as it involves placing dozens of symbols and connecting up to hundreds of pins. However, current design automation mostly focuses on the next stage, PCB layout [Tian et al., 2022, 2021, Liu et al., 2024] and routing [Zhang et al., 2020, Li et al., 2023, Siemens Digital Industries Software, 2025], while the schematic design process still heavily relies on the unscalable manual process. Recent research explored code-based PCB design, such as SKiDL [2025], but it still relies on

manual coding and skips the schematic design step to generate netlists. Thus, it is unable to provide a user-friendly, interpretable schematic visual image, making it not a mainstream practice.

2.2 Generative AI for Hardware Designs

Built on hierarchically abstracted Boolean logic representations, e.g., Verilog and VHDL, recent research Wu et al. [2024], Thakur et al. [2024], Fu et al. [2023] has explored language model-based digital IC design. For analog IC design, CktGNN [Dong et al., 2023] first represents the circuit topology as graph structures, formulating a graph generation task to enable the design of various topologies. AnalogCoder [Lai et al., 2025] presents a training-free LLM agent through Python code generation, while LaMAGIC [Chang et al., 2024] fine-tunes a masked language model. AnalogGenie [Gao et al., 2025] presents a comprehensive dataset to enable LLM pre-training for circuit topology generation. Another recent exploration is to generate 3D CAD models of mechanical parts. Wu et al. [2021] describes a shape as a sequence of computer-aided design (CAD) operations. Wang et al. [2025] introduces a VLM visual feedback stage for CAD generation model training, while Alam and Ahmed [2025] converts raw image input to editable parametric CAD sequences.

These methods commonly rely on task-specific representations and curated data. In contrast, generating PCB schematics from user requests remains largely unexplored, lacking both an effective representation and a dedicated dataset. Closest to our setting are works on converting circuit-diagram images to netlists [Huang et al., 2025, Xu et al., 2025] and translating netlists into schematics [Matsuo et al., 2024], which are distinct from our end-to-end schematic generation from user prompts. Zou et al. [2026] provides a feedback-based iterative design framework for the PCB schematic design task; however, it is evaluated on only 23 tasks and relies on SKiDL schematic graph rendering to avoid spatial reasoning issues, which does not scale well to larger schematics.

2.3 Hardware Design Datasets

The development of AI-based hardware generation depends on training on large-volume and high-quality datasets. Kunal et al. [2019], Dong et al. [2023], Gao et al. [2025] provide datasets for analog circuits, but do not apply to PCB schematic design. To tackle data scarcity, we have constructed a dataset of 1390 different schematic types using the open-source online design resources on *Sparkfun* [SparkFun Electronics, 2025] under the CC BY-SA 4.0 license, as a reference. We also present a scalable and efficient dataset collection pipeline for design collection in Section 3.2.

3 Method

To address the representation bottleneck, we first introduce our code representation for schematic designs, which captures the semantics of schematic editing operations and the rationales behind spatial placement and connections of circuit symbols. Built on this, we construct a dataset of schematic designs by converting web PCB designs to code. We propose an agent-human collaboration pipeline to enable scalable data acquisition and synthesize corresponding user request prompts. Finally, we perform the training of SchGen with the constructed dataset of user prompts and code representations.

3.1 Code Representation for PCB Schematics

Existing representations are hard for LLMs to learn from and generate correctly. Although existing LLMs appear to have the ability to generate PCB schematic designs when asked in the prompt, they fail to produce valid schematic files that can be parsed by EDA tools, e.g., *KiCad* . Section 2 illustrates a comparison between existing representations and our proposed code representation. When using text-based representation, i.e., raw *KiCad* schematic file's text content, LLMs struggle to capture the format of the schematic due to the presence of excessive schematic formatting details and redundant information that are irrelevant to the schematic's functionality, which also bring extra high token consumption. When using LLM and image generation, the generated schematics contain distorted symbols and show a random format, making it infeasible to convert them into valid schematic files.

To tackle the challenges above, our goal is to find a new learning-efficient representation. Our approach is inspired by the observation that human engineers typically follow a systematic process when drawing schematics that can be abstracted into a series of editing operations backed by

Table 1: Code representation illustration and ablation.

Representation	Example	MDL	LZ Norm	Val Loss ($\times 10^{-2}$)
		(mean / median)		

Code-L1 (SchGen)	`add_schematic_symbol(...pos_x=center_x_1, pos_y=center_y_1,...)`
	`add_schematic_symbol(...pos_x=center_x_1-58, pos_y=center_y_1+18,...)`
	`connect_pins("PWR1", "+1V8", "U2", "VDDIO")`
		2.19/2.31	**1.36/1.42**	**1.29/0.25**

Code-L2 (w/o relative coord)	`add_schematic_symbol(...pos_x=157.48, pos_y=99.510,...)`
	`add_schematic_symbol(...pos_x=99.06, pos_y=117.29,...)`
	`connect_pins("PWR1", "+1V8", "U2", "VDDIO")`
		2.42/2.57	1.56/1.64	1.64/0.77

Code-L3 (w/o pin name)	`add_schematic_symbol(...pos_x=157.48, pos_y=99.510,...)`
	`add_schematic_symbol(...pos_x=99.06, pos_y=117.29,...)`
	`add_new_wire([99.06, 117.29], [114.3, 117.29])`
		2.43/2.63	1.60/1.64	3.96/3.20

clear rationales. Specifically, engineers first place central symbols that represent the core circuit components, then arrange other symbols around them based on their functional connections. Then, the pins are connected according to the connections between circuit component pins, e.g., VCC pins and GND pins are connected to power source and ground symbols, respectively. We summarize two key insights from the above process: (1) The schematic design can be abstracted as a series of editing operations, including adding symbols, placing labels, and connecting pins; (2) The placement of symbols and labels is typically relative to a local reference based on functional correlations; the wire connections follow clear rationales based on the pin names that encode the pin functionalities.

Based on the observations, we introduce the following code APIs to form our code representation:

```
def add_schematic_symbol(symbol_lib, symbol_name, x, y, ref, value, rotation, mirror)
def add_label(label_pos, label_text, label_ref, label_type, text_orient):
def get_pin_location(symbol_ref, pin_name):
def connect_pins(symbol_a, pin_a, symbol_b, pin_b):
def write_out_all_wires():
```

add_schematic_symbol() places a symbol with the given name from a symbol library 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 symbol reference name and the pin name. For power symbols and net labels that have one pin only, the pin name is set as default to '1'. *connect_pins()* connects two pins according to the symbol reference names and pin names of the two pins. Finally, *write_out_all_wires()* writes out all wires to a *KiCad* schematic file with specified connections and performs a basic automatic routing.

LLMs' ability to understand spatial relationships is relatively limited [Yamada et al., 2024]. For our task, the schematic may involve dozens of symbols/labels and numerical coordinates, which are hard for LLMs to generate correctly. To handle it, we choose to use local coordinate instead of absolute coordinates in *add_schematic_symbol* and *add_label*. More specifically, we first get the coordinates of anchor points (for symbols, it is the center symbol in the circuit; for labels, it is the pin that it attaches to), then calculate the offsets and represent the coordinates with respect to the anchor points. After executing the code, a valid *KiCad* schematic file is generated that can be opened and edited in *KiCad* tool, as illustrated in Section 2.

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 draw wire segments with absolute coordinates.

We use three metrics to compare these representations over our dataset: (1) **Minimum Description Length (MDL)**: MDL $= 8 \cdot$ compressed_bytes/raw_bytes, which uses lossless compression to approximate data complexity. Lower bits-per-byte indicates a more structured and learnable representation [Zhao et al., 2023, Shalev-Shwartz and Ben-David, 2014]. (2) **Lempel–Ziv Complexity (LZ Norm)**: LZ_norm $= c(n)\ \log n/n$, where $c(n)$ is the number of phrases from incremental parsing. A lower value implies lower intrinsic sequence complexity [Maveli et al., 2026, Amigó et al., 2004].

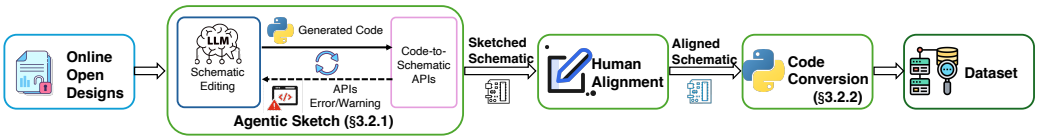


Figure 3: Pipeline of constructing the schematic dataset from online open-source PCB designs using LLM sketching and human correction.

(3) **Validation Loss (Val Loss)**: We calculate the validation loss of each representation on the test dataset. Lower val loss indicates better performance and generalization ability.

Across the three code variants, Code-L1 achieves the lowest MDL, LZ Norm, and Val Loss, indicating that relative coordinates and pin-name connectivity jointly produce a more structured and compressible representation. These metrics serve as proxies for better learnability, a prediction we validate empirically in Section 4.2. To summarize, our APIs abstract away the complexity of raw *KiCad* files by capturing key editing operations, including relative-coordinate placement and pin-name-based connectivity, which enables more structured and semantically meaningful schematic generation.

3.2 Constructing the Schematic Design Dataset

Most online PCB designs provide an image of the schematic design instead of editable source files, which avoids issues with EDA tool compatibility and version discrepancies. However, it brings challenges to constructing a dataset based on schematic files. Thus, we propose a pipeline shown in Figure 3 to convert the design image to editable schematic and corresponding code representations. We first introduce an *agentic-sketch* module to acquire a draft (Section 3.2.1), which is later annotated by human engineers to fix possible errors. Then, we develop a *schematic-to-code* converter to translate the aligned schematic into code representations to form the dataset (Section 3.2.2). Our approach enables scalable data acquisition from online open-source PCB designs.

3.2.1 Agentic Sketch

As shown in Figure 3, we take online images of reference designs as the input and leverage multi-modal LLMs, e.g., GPT-5, to generate Python code that uses APIs in Section 3.1. Through our underlying implementation, the compiled program outputs error and warning feedback in execution, e.g., wrong syntax and illegal symbols. Then, the multi-modal LLM iterates code generation based on the feedback, until no error or reaching max iterations, and then outputs a sketch version of the schematic file by executing the code.

When sketching the schematics, multi-modal LLMs can make mistakes regarding complex wire connections even after iterations. For example, it is hard for LLMs to determine whether two wires are connected or intersect. Thus, we introduce manual editing to adjust the incorrect schematic designs to ensure their alignment with the online design reference. The agentic sketch can accurately reproduce symbols in KiCAD from the image input within 5 rounds of iterations. On average, it takes less than 20 seconds for verification and potential alignment fix for each schematic design from our data pipeline, while it may take more than 5 minutes to plot the schematic manually from scratch. Combining the two steps, we produce a collection of PCB schematic designs in *KiCad* format by processing resources from online PCB designs like hardware datasheets and designs on *Sparkfun*.

3.2.2 Schematic-to-Code Conversion

Based on the collected schematic files, we develop a schematic-to-code converter that parses raw *KiCad* s-expression files into undirected graph representations, where pins and wires are treated as vertices and edges. Through graph traversal, we identify connected pin pairs and generate code following our API abstractions in Section 3.1, including relative-coordinate symbol placement, label attachment, and pin-name-based connectivity, ultimately reproducing the original schematic with *write_out_all_wires*. A sample schematic and corresponding representations are shown in 6.1.

3.3 Model Training

Recall that our goal is to build a model that can generate a schematic based on the user request (Figure 1). Thus, 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 further ensure consistency between the synthetic user requests and real ones, we prompt the model with examples authored by human users. We also introduce two styles of requests to augment the dataset: concise and detailed, to model users with different background knowledge levels. 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 brief summary of its function. For the detailed request, the user request includes the specific circuit components and connections that the user wants to include in the schematic. Two samples of different request styles of the schematic are shown in 6.2.

We use Apache-2.0 licensed GPT-oss-20B [Agarwal et al., 2025] as the base model of SchGen and perform supervised fine-tuning for the schematic code generation task. To utilize the reasoning capability of the model, we augment the dataset with distilled chain of thought reasoning, following the approach of prior work [Ho et al., 2022, Chen et al., 2025]. Specifically, we synthesize the thinking process by calling the larger reasoning model GPT-oss-120B and GPT-oss-20B itself, prompting them to generate the chain-of-thought (CoT) reasoning that leads to the output from the input request.

4 Experimental Validation

4.1 Experiment Setup

Dataset details. Our proposed dataset contains 2105 *KiCad* schematics with 1390 unique designs, each having code representations, user requests, and CoT. The total volume is quadrupled to 8420 samples using two styles of user request and CoT reasoning from two models as described in Section 3.3. The schematics span various types of functionality, covering microcontroller, analog modules, LED, power, storage, battery, USB, antenna, connectors, etc, with up to 39 symbols and 48 labels per design. We randomly select 500 samples as the held-out test set, and use the remaining as the training set. The distribution of the dataset can be found in Figure 5 of Section 6.3.

Training setup. We choose the open-source model of *GPT-oss* with 20 billion parameters as the base model and train it with supervised finetuning and LoRA [Hu et al., 2021]. The training setting and parameters are shown in Section 6.4. We run training and inference on an 80 GB Nvidia A100 GPU.

Evaluation metrics. PCB schematics are typically evaluated from multiple perspectives, including rule-based methods [Lee et al., 2003], netlist verification [Mitzner, 2009], and design review by human experts. Based on these, we consider the following metrics:

(a) Valid Circuits. Valid Circuits are determined by the ratio of generated circuits that can pass two sanity checks: (1) Python code 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) Zero critical errors reported by the Electrical Rules Check (ERC) of *KiCad* , including short circuits, net conflict, illegal connections, etc.

(b) Spatial Violation. Spatial Violation measures the number of *spatial overlaps* among symbols, labels, and wires. Each object is assigned a bounding box, and any intersection is counted as an overlap. To mitigate the statistical biases brought by different pass ratios, we normalize the average overlaps following: $\bar{n}_{weighted} = \frac{\bar{n}_{original}}{\text{pass ratio}}$. We use this metric as a readability proxy, as overlaps hinder engineer inspection and reflect the model's spatial reasoning ability.

(c) Netlist Accuracy. Netlist Accuracy compares the symbols and connections of the generated netlist against the ground truth. We define a node $\boldsymbol{v}_i = (\text{symbol}, \text{pin})$; nodes sharing a net form a set $\mathbb{N}$, and a netlist is $\mathbb{G} = \{\mathbb{N}_1, \cdots, \mathbb{N}_m\}$. We report *Jaccard*, *Precision*, and *Recall* between $\mathbb{G}_{gen}$ and $\mathbb{G}_{gt}$. Netlist accuracy is a strong proxy for schematic correctness, as it fully captures logical connectivity and serves as the sole input for downstream tasks of footprint layout and wiring.

(d) Expert Verification. We randomly sample 100 designs from the testing set and have two experts evaluate the generated schematics according to consistent rubrics. Symbol Error and Connection Error

Table 2: Performance comparison of SchGen finetuned with different representations

Method	Valid Circuits ↑	Spatial Violation ↓	Netlist Accuracy (%)↑			Expert Verification		
	(Pass ratio%)	(Number of overlaps)	Jaccard	Precision	Recall	Symbol Error ↓	Connection Error ↓	Functional Correctness(%) ↑
Code-L1 (SchGen)	**82.00**	**7.73**	**49.08**	**54.87**	**52.80**	**0.06**	**0.61**	**60.5**
Code-L1 w/o CoT	53.40	8.32	30.47	33.01	35.44	0.98	1.58	14.0
Code-L2	78.16	7.77	45.97	52.58	49.54	0.28	1.76	33.0
Code-L3	76.40	8.14	15.46	24.75	15.66	0.24	6.76	6.0
KiCad File	32.45	7.80	9.11	14.51	9.33	1.22	9.42	3.0

count incorrect/missing components and faulty pin connections (reported as averages). Functional Correctness reports the ratio of designs that can function as intended.[2]

These metrics cover structural correctness, connectivity, and end-to-end functional validity of the generated schematics to ensure a comprehensive assessment. We do not adopt SPICE simulation for evaluation because most PCB schematics are system-level mixed-domain designs containing components beyond the scope of available SPICE models. Consequently, SPICE is more suitable for small analog sub-circuits than holistic validation of full PCB schematics.

Baselines. To isolate the contribution of each design choice in our representation, we compare SchGen, trained with the proposed Code-L1 representation, against ablation variants: Code-L2 (w/o relative coordinates) and Code-L3 (w/o relative coordinates and pin-name connectivity), each finetuned from *GPT-oss-20b* on the same dataset. We also include the raw *KiCad* schematic file representation, which directly uses the text of the *KiCad* schematic file as training data. Moreover, using the testing set of the proposed dataset as the benchmark, we compare SchGen with the *GPT-oss-20b* vanilla model and three larger LLMs, including *GPT-o4mini* [OpenAI, 2025b], *GPT-5.2* [OpenAI, 2025a], and *Grok-4* [xAI, 2025]. All models are evaluated in a single-run setting on identical test inputs. To ensure fairness, we provide uniform prompts, including schematic representation descriptions and example code, and grant access to the same Python APIs and example code.

4.2 Main Results

Effectiveness of code representations. Table 2 reports the performance of finetuning *GPT-oss*-20b with datasets using different schematic representations. Our proposed Code-L1 achieves the best performance for all metrics, including valid circuit ratio, netlist accuracy, spatial violation, and expert verification results. Compared to Code-L2 and Code-L3 as ablation studies, Code-L1 outperforms both, confirming that relative coordinates and pin-name connectivity contribute to generation quality. Comparing Code-L2 to Code-L3, which differs only in wire specification, reveals a large drop in netlist accuracy, indicating that pin-name connectivity is critical for correct wiring. These trends align with the structural complexity analysis in Table 1. We analyze the failure cases and draw two observations: (1) Model trained with Code-L2 tends to put symbols at wrong locations, which induces critical connection errors; (2) Model trained with Code-L3 cannot correctly determine the locations of wires and leaves many pins unconnected. We also compare with Code-L1 without chain-of-thought (CoT) synthesis for the dataset, and observe a performance drop due to the lack of intermediate reasoning steps. Moreover, we note that netlist accuracy performance corresponds well with expert verification results, confirming that netlist accuracy is a good proxy for functional correctness of the generated schematics. Lastly, *KiCad* file representation yields the lowest performance for all metrics. The low valid circuit ratio shows the difficulty of learning from raw *KiCad* files, highlighting that the proposed code representation is necessary for the schematic generation task.

Comparison with frontier models. We use our testing set as the benchmark to evaluate the performance of SchGen versus frontier large language models, including GPT-oss-20b base model, *GPT-5.2*, *GPT-o4mini*, and *Grok-4*. For fair comparison, we select frontier models prompted with the same representation as SchGen to do expert verification. As shown in Table 3, SchGen outperforms all other models across all metrics, despite being finetuned from a base model with only 20 billion parameters. Moreover, comparing the same model prompted by different representations, the perfor-

[2]Cross-validation between the two experts shows high agreement, with average correctness ratio differences below 3%.

Table 3: Performance comparison of different models

Method	Valid Circuits ↑	Spatial Violation ↓	Netlist Accuracy (%)↑			Expert Verification		
	(Pass ratio%)	(Number of overlaps)	Jaccard	Precision	Recall	Symbol Error ↓	Connection Error ↓	Functional Correctness(%) ↑
SchGen	**82.00**	**7.73**	**49.08**	**54.87**	**52.80**	**0.06**	**0.61**	**60.5**
Vanilla GPT-oss 20b	10.99	18.37	9.48	9.60	9.60	0.75	2.21	17.0
GPT-5.2-L1	67.89	8.56	42.95	45.72	49.51	0.11	0.94	50.0
GPT-5.2-L2	59.26	11.12	35.53	37.53	40.14			
GPT-5.2-L3	68.03	9.55	41.06	43.38	46.71			
GPT-5.2-*KiCad*	10.53	–	0.53	0.88	0.88			
GPT-o4mini-L1	59.70	9.40	32.25	34.62	37.71	0.43	1.23	37.0
GPT-o4mini-L2	59.04	8.92	33.62	34.94	35.07			
GPT-o4mini-L3	60.91	8.61	34.97	36.74	37.13			
GPT-o4mini-*KiCad*	5.96	–	0.00	0.00	0.00			
Grok-4-L1	61.21	8.82	44.03	49.77	48.73	0.20	1.19	47.0
Grok-4-L2	52.07	10.69	32.74	34.56	34.14			
Grok-4-L3	53.13	9.98	35.88	37.80	37.26			
Grok-4-*KiCad*	3.95	–	1.32	1.32	1.32			

Table 4: Performance of SchGen on unseen Github design dataset

Method	Valid Circuits ↑	Spatial Violation ↓	Netlist Accuracy (%)↑		
	(Pass ratio%)	(Number of overlaps)	Jaccard	Precision	Recall
SchGen	65.59	**7.46**	**40.65**	**47.12**	**43.17**
GPT-5.2	77.02	12.12	40.64	43.53	43.03
GPT-o4mini	72.06	8.72	40.47	44.22	42.67

mance of models prompted by the Code-L1 representation is superior in most metrics, which proves its effectiveness 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 these large models often use relative coordinates for symbol placement and pin locations for wire connections, even if the prompt example code does not adopt them. We also prompt them with a *KiCad* raw text file, and the results show low pass ratio and netlist accuracy[3].

Table 3 also shows the differences among LLMs on this new benchmark task. We see that *GPT-5.2* achieves the best performance across the three code representations, while *Grok-4* performs very closely. This implies their advanced ability to handle complex spatial relations. Nevertheless, the superior performance of SchGen shows that performing training on our dataset with the right representation is crucial for the schematic generation task, outperforming even much larger models.

Generalization. We further test whether SchGen can generate novel schematics that are out of the domain of our training set. For this purpose, we build a challenging unseen dataset, which consists of 988 samples collected from 20 open-source *KiCad* projects on GitHub, which are from a completely different source compared to the dataset used for training. We select two representative frontier models and objective metrics to evaluate over it. The results in Table 4 show that SchGen can generalize to unseen designs with the learned design logic. Our performance is comparable to GPT-5.2 prompted with our code representations. Note that our model is much smaller than frontier models like GPT-5.2, with only 20 billion parameters, which limits the capacity of the model to memorize and generalize. We also provide the visualization of two novel schematic examples in 6.5, where we also put two samples of successful examples in the test set in Fig. 6.

5 Conclusion

In this work, we introduced SchGen, a large language model for PCB schematic design generation. By proposing a semantic-grounded code representation, we transform schematic design into a

[3]The number of overlaps is not available for *KiCad* file representation because valid circuits are too few.

structured sequence of interpretable editing Python code, which allows LLMs to effectively capture both spatial reasoning and functional intent. We further constructed a dataset of schematic designs through an agent–human collaborative pipeline, providing high-quality training data for the new task. Experimental results demonstrate that SchGen substantially outperforms ablation variants and general-purpose models in terms of valid circuit generation, spatial arrangement, and netlist accuracy. These findings demonstrate that representation design is the key enabler to apply LLMs to hardware design. SchGen currently focuses on structured schematics from SparkFun designs and does not yet model advanced PCB constraints. In addition, scaling to large and complex PCBs remains challenging due to the lack of datasets and capabilities of LLMs, motivating future work on larger datasets, stronger models, and more efficient design pipelines. Human involvement is also needed to ensure robustness and safety of the generated schematics. Looking forward, we envision SchGen as a foundation for broader automation in electronic design, paving the way toward fully generative, user-driven hardware creation.

References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, et al. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Md Ferdous Alam and Faez Ahmed. Gencad: Image-conditioned computer-aided design generation with transformer-based contrastive representation and diffusion priors, 2025. URL https://arxiv.org/abs/2409.16294.

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

José M. Amigó, Janusz Szczepański, Elek Wajnryb, and Maria V. Sanchez-Vives. Estimating the entropy rate of spike trains via lempel-ziv complexity. *Neural Computation*, 16(4):717–736, 04 2004. ISSN 0899-7667. doi: 10.1162/089976604322860677. URL https://doi.org/10.1162/089976604322860677.

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

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

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

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

EAGLE. The future of autodesk eagle: Autodesk fusion electronics. https://www.autodesk.com/products/fusion-360/blog/future-of-autodesk-eagle-fusion-360-electronics/, June 2023. Fusion Blog. Accessed: 2025-09-02.

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

Jian Gao, Weidong Cao, Junyi Yang, and Xuan Zhang. Analoggenie: A generative engine for automatic discovery of analog circuit topologies. *arXiv preprint arXiv:2503.00205*, 2025.

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

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

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

KiCad. KiCad. https://www.kicad.org/, 2025. Accessed: 2025-09-02.

Kishor Kunal, Meghna Madhusudan, Arvind K. Sharma, Wenbin Xu, Steven M. Burns, Ramesh Harjani, Jiang Hu, Desmond A. Kirkpatrick, and Sachin S. Sapatnekar. Invited: Align – open-source analog layout automation from the ground up. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–4, 2019.

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

Geun-Cheol Lee, Yeong-Dae Kim, Jae-Gon Kim, and Seong-Hoon Choi. A dispatching rule-based approach to production scheduling in a printed circuit board manufacturing system. *Journal of the Operational Research Society*, 54(10):1038–1049, 2003.

Haiyun Li, Jixin Zhang, Ning Xu, and Mingyu Liu. Fanoutnet: A neuralized pcb fanout automation method using deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(7):8554–8561, Jun. 2023. doi: 10.1609/aaai.v37i7.26030. URL https://ojs.aaai.org/index.php/AAAI/article/view/26030.

Wen-Hao Liu, Anthony Agnesina, and Haoxing Mark Ren. Challenges for automating pcb layout. In *Proceedings of the 2024 International Symposium on Physical Design*, ISPD '24, page 91–92, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704178. doi: 10.1145/3626184.3635285. URL https://doi.org/10.1145/3626184.3635285.

Ryoga Matsuo, Stefan Uhlich, Arun Venkitaraman, Andrea Bonetti, Chia-Yu Hsieh, Ali Momeni, Lukas Mauch, Augusto Capone, Eisaku Ohbuchi, and Lorenzo Servadei. Schemato–an llm for netlist-to-schematic conversion. *arXiv preprint arXiv:2411.13899*, 2024.

Nickil Maveli, Antonio Vergari, and Shay B. Cohen. Can llms compress (and decompress)? evaluating code understanding and execution via invertibility, 2026. URL https://arxiv.org/abs/2601.13398.

Kraig Mitzner. *Complete PCB design using OrCAD Capture and PCB editor*. Newnes, 2009.

OpenAI. Gpt-5. https://openai.com/index/introducing-gpt-5-2/, 2025a. Introducing GPT-5.2.

OpenAI. Gpt-o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/, 2025b. Introducing OpenAI o3 and o4-mini.

Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

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

SKiDL. Skidl, 2025. URL https://github.com/devbisme/skidl. Accessed: 2025-09-18.

SparkFun Electronics. Sparkfun electronics, 2025. URL https://www.sparkfun.com/. Accessed: 2025-09-13.

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

Yidong Tian, Andrew J. Forsyth, Zhuoru Li, and Cheng Zhang. A component manipulation algorithm to enable design automation of power electronic pcbs. In *2021 IEEE Design Methodologies Conference (DMC)*, pages 1–6, 2021. doi: 10.1109/DMC51747.2021.9529938.

Yidong Tian, Andrew J. Forsyth, Zhuoru Li, and Cheng Zhang. Automatic layout design for power electronics pcbs. In *2022 IEEE Energy Conversion Congress and Exposition (ECCE)*, pages 1–6, 2022. doi: 10.1109/ECCE50734.2022.9947957.

Ruiyu Wang, Yu Yuan, Shizhao Sun, and Jiang Bian. Text-to-cad generation through infusing visual feedback in large language models, 2025. URL https://arxiv.org/abs/2501.19054.

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

Rundi Wu, Chang Xiao, and Changxi Zheng. Deepcad: A deep generative network for computer-aided design models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 6772–6782, October 2021.

xAI. Grok 4. https://x.ai/news/grok-4, 2025. Grok 4.

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. Image2net: Datasets, benchmark and hybrid framework to convert analog circuit diagrams into netlists, 2025. URL https://arxiv.org/abs/2508.13157.

Yutaro Yamada, Yihan Bao, Andrew K. Lampinen, Jungo Kasai, and Ilker Yildirim. Evaluating spatial understanding of large language models, 2024. URL https://arxiv.org/abs/2310.14540.

Cong Zhang, Huilin Jin, Jienan Chen, Jinkuan Zhu, and Jinting Luo. A hierarchy mcts algorithm for the automated pcb routing. In *2020 IEEE 16th International Conference on Control & Automation (ICCA)*, pages 1366–1371, 2020. doi: 10.1109/ICCA51439.2020.9264558.

Zirui Zhao, Wee Sun Lee, and David Hsu. Large language models as commonsense knowledge for large-scale task planning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 31967–31987. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/65a39213d7d0e1eb5d192aa77e77eeb7-Paper-Conference.pdf.

Huanghaohe Zou, Peng Han, Emad Nazerian, and Alex Q. Huang. Pcbschemagen: Constraint-guided schematic design via llm for printed circuit boards (pcb), 2026. URL https://arxiv.org/abs/2602.00510.

6 Appendix

6.1 Example of Schematic Design and Code Representations

Fig. 4 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.

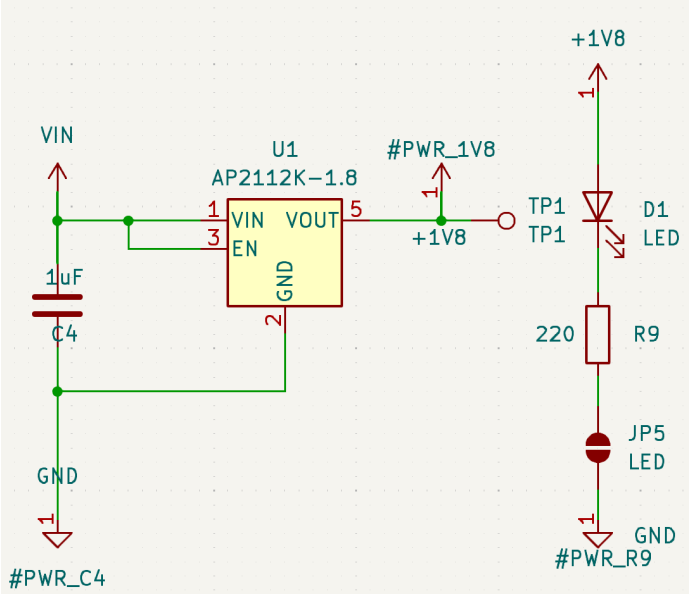


Figure 4: An example schematic designed in *KiCad*

Listing 1: Level 1 Representation

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

# Get project path and import 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, 105

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), pos_y=
    center_y_1 + (5), reference="#PWR1", value="VIN", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Device", symbol_name="C", pos_x=center_x_1 + (-20), pos_y=
    center_y_1 + (-5), reference="C1", value="1uF", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="power", symbol_name="GND", pos_x=center_x_1 + (-20), pos_y=
    center_y_1 + (-24), reference="#PWR_C1", value="GND", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="power", symbol_name="+1V8", pos_x=center_x_1 + (13), pos_y=
    center_y_1 + (5), reference="#PWR_1V1", value="+1V8", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Connector", symbol_name="TestPoint", pos_x=center_x_1 + (16),
     pos_y=center_y_1 + (2), reference="TP1", value="TP1", rotation=270, mirror="x")

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

```
### Connecting all wires in the Schematic ###


# Connecting #PWR_1V1 pin +1V8 (Pin ID 1 -- Name +1V8) to TP1 pin TP1 (Pin ID 1 -- Name TP1)
connect_pins("#PWR_1V1", "+1V8", "TP1", "TP1")

# Connecting #PWR1 pin VIN (Pin ID 1 -- Name VIN) to C1 pin 1 (Pin ID 1 -- Name None)
connect_pins("#PWR1", "VIN", "C1", "1")

# Connecting U1 pin VOUT (Pin ID 5 -- Name VOUT) to #PWR_1V1 pin +1V8 (Pin ID 1 -- Name +1V8)
connect_pins("U1", "VOUT", "#PWR_1V1", "+1V8")

# 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 C1 pin 2 (Pin ID 2 -- Name None) to #PWR_C1 pin 1 (Pin ID 1 -- Name None)
connect_pins("C1", "2", "#PWR_C1", "1")

# 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 C1 pin 2 (Pin ID 2 -- Name None) to U1 pin 2 (Pin ID 2 -- Name None)
connect_pins("C1", "2", "U1", "2")

### Placing center symbol 2 : Device:LED###

center_x_2, center_y_2 = 149, 108

add_schematic_symbol(symbol_lib="Device", symbol_name="LED", pos_x=center_x_2, pos_y=
    center_y_2, reference="D1", value="LED", rotation=90, 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), pos_y=
    center_y_2 + (10), reference="#PWR_1V2", value="+1V8", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Device", symbol_name="R", pos_x=center_x_2 + (0), pos_y=
    center_y_2 + (-11), reference="R1", value="220", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Jumper", symbol_name="SolderJumper_2_Open", pos_x=center_x_2
     + (0), pos_y=center_y_2 + (-21), reference="JP1", value="LED", rotation=270, mirror="
    None")
add_schematic_symbol(symbol_lib="power", symbol_name="GND", pos_x=center_x_2 + (0), pos_y=
    center_y_2 + (-27), reference="#PWR_R1", value="GND", rotation=0, mirror="None")

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


### Connecting all wires in the Schematic ###


# Connecting JP1 pin B (Pin ID 2 -- Name B) to #PWR_R1 pin 1 (Pin ID 1 -- Name None)
connect_pins("JP1", "B", "#PWR_R1", "1")

# Connecting R1 pin 2 (Pin ID 2 -- Name None) to JP1 pin A (Pin ID 1 -- Name A)
connect_pins("R1", "2", "JP1", "A")

# Connecting D1 pin K (Pin ID 1 -- Name K) to R1 pin 1 (Pin ID 1 -- Name None)
connect_pins("D1", "K", "R1", "1")

# Connecting #PWR_1V2 pin +1V8 (Pin ID 1 -- Name +1V8) to D1 pin A (Pin ID 2 -- Name A)
connect_pins("#PWR_1V2", "+1V8", "D1", "A")

write_out_all_wires()
```

Listing 2: Level 2 Representation

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

```
import os

# Get project path and import 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=100.33, pos_y=109.67,
    reference="#PWR1", value="VIN", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Device", symbol_name="C", pos_x=100.33, pos_y=99.51,
    reference="C4", value="1uF", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="power", symbol_name="GND", pos_x=100.33, pos_y=80.46,
    reference="#PWR_C4", value="GND", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="power", symbol_name="+1V8", pos_x=134.62, pos_y=109.67,
    reference="#PWR_1V8", value="+1V8", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Connector", symbol_name="TestPoint", pos_x=137.16, pos_y
    =107.13, reference="TP1", value="TP1", rotation=270, mirror="x")

### Placing all global labels in the Schematic and connect 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 : Device:LED###
center_x_2, center_y_2 = 148.590, 108.400
add_schematic_symbol(symbol_lib="Device", symbol_name="LED", pos_x=center_x_2, pos_y=
    center_y_2, reference="D1", value="LED", rotation=90, 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=148.59, pos_y=118.56,
    reference="#PWR_1V1", value="+1V8", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Device", symbol_name="R", pos_x=148.59, pos_y=96.97,
    reference="R9", value="220", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Jumper", symbol_name="SolderJumper_2_Open", pos_x=148.59,
    pos_y=86.81, reference="JP5", value="LED", rotation=270, mirror="None")
add_schematic_symbol(symbol_lib="power", symbol_name="GND", pos_x=148.59, pos_y=80.46,
    reference="#PWR_R9", value="GND", rotation=0, mirror="None")

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

### Connecting all wires in the Schematic ###

# 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 R9 pin 2 (Pin ID 2 -- Name None) to JP5 pin A (Pin ID 1 -- Name A)
connect_pins("R9", "2", "JP5", "A")
# 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()
```

Listing 3: Level 3 Representation

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

# Get project path and import 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=100.33, pos_y=109.67,
    reference="#PWR1", value="VIN", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Device", symbol_name="C", pos_x=100.33, pos_y=99.51,
    reference="C4", value="1uF", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="power", symbol_name="GND", pos_x=100.33, pos_y=80.46,
    reference="#PWR_C4", value="GND", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="power", symbol_name="+1V8", pos_x=134.62, pos_y=109.67,
    reference="#PWR_1V8", value="+1V8", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Connector", symbol_name="TestPoint", pos_x=137.16, pos_y
    =107.13, reference="TP1", value="TP1", rotation=270, mirror="x")

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

### Adding all wires in the Schematic ###
add_new_wire([106.68, 107.13], [113.03, 107.13])
add_new_wire([100.33, 91.89], [100.33, 95.7])
add_new_wire([120.65, 91.89], [120.65, 96.97])
add_new_wire([100.33, 80.46], [100.33, 91.89])
add_new_wire([134.62, 109.67], [134.62, 107.13])
add_new_wire([100.33, 103.32], [100.33, 107.13])
add_new_wire([106.68, 104.59], [106.68, 107.13])
add_new_wire([100.33, 107.13], [100.33, 109.67])
add_new_wire([106.68, 104.59], [113.03, 104.59])
add_new_wire([128.27, 107.13], [134.62, 107.13])
add_new_wire([100.33, 91.89], [120.65, 91.89])
add_new_wire([100.33, 107.13], [106.68, 107.13])
add_new_wire([134.62, 107.13], [137.16, 107.13])

### Placing center symbol 2 : Device:R###
center_x_2, center_y_2 = 148.590, 96.970
add_schematic_symbol(symbol_lib="Device", symbol_name="R", pos_x=center_x_2, pos_y=center_y_2,
     reference="R9", value="220", rotation=0, 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=148.59, pos_y=118.56,
    reference="#PWR_1V1", value="+1V8", rotation=0, mirror="None")
add_schematic_symbol(symbol_lib="Device", symbol_name="LED", pos_x=148.59, pos_y=108.4,
    reference="D1", value="LED", rotation=90, mirror="None")
add_schematic_symbol(symbol_lib="Jumper", symbol_name="SolderJumper_2_Open", pos_x=148.59,
    pos_y=86.81, reference="JP5", value="LED", rotation=270, mirror="None")
add_schematic_symbol(symbol_lib="power", symbol_name="GND", pos_x=148.59, pos_y=80.46,
    reference="#PWR_R9", value="GND", rotation=0, mirror="None")

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

```
### Adding all wires in the Schematic ###
add_new_wire([148.59, 104.59], [148.59, 100.78])
add_new_wire([148.59, 83], [148.59, 80.46])
add_new_wire([148.59, 93.16], [148.59, 90.62])
add_new_wire([148.59, 118.56], [148.59, 112.21])

write_out_all_wires()
```

6.2 Example of the request

Listing 4: 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 5: Detailed style of request

```
{"messages": [{"role": "user", "content": "I am looking to add a small 1.8 V power block to a
     larger board, mainly to step down a VIN rail and provide a clean +1V8 output. I would
    also like a simple visual indicator (LED) to show when the rail is active, but with the
    option to disable it using a solder jumper. For the regulator, please use an AP2112K-1.8
     in SOT-23-5. VIN will be the input, and the output should be labeled +1V8. The enable
    pin should be tied to VIN so the regulator turns on automatically whenever power is
    present. The NC pin can remain unconnected. On the input side, include a 1 uF capacitor
    between VIN and GND for decoupling. For the status indicator, I want an LED connected to
     the +1V8 rail. The idea is that it only lights up when I close a solder jumper. The LED
     should go from +1V8 through a series resistor (220 Ohm), then into a solder jumper, and
     finally to GND. If the jumper is open, the LED stays off; if shorted, it lights up.
    Also, please add a test point on the +1V8 rail so I can easily probe it during debugging.
     In terms of naming, I would like to keep VIN, +1V8, and GND as the main net labels. The
     LED path can use intermediate net names generated by the tool.\n"}]}
```

6.3 Dataset Statistics

Here, we define the complexity of any schematic as the sum of the number of symbols and labels.

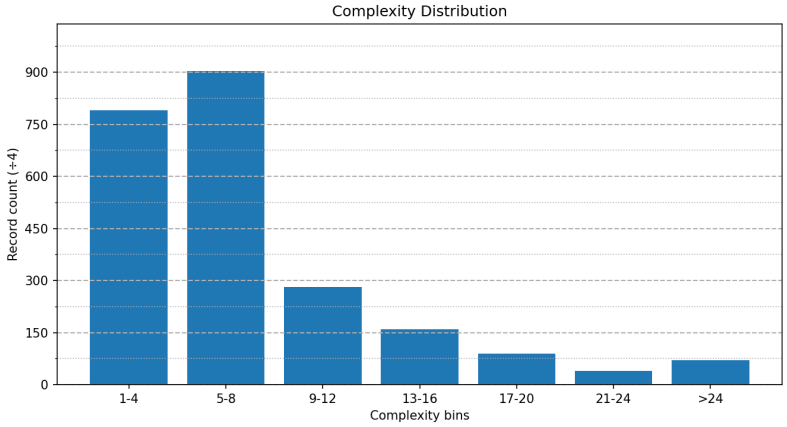


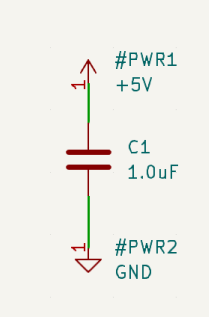
Figure 5: Histogram of the complexity distribution of the dataset

6.4 Training Setup

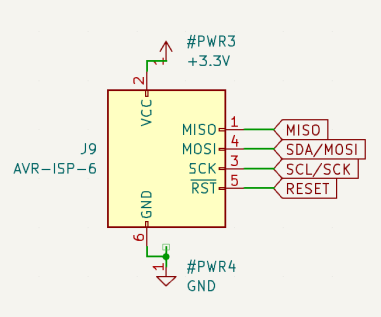
6.5 Examples of generation

Table 5: Training Configuration

hyperparameter	**value**
SFT configuration	
learning rate	`4e-4`
optimizer	`AdamW`
assistant loss only	✓
max token length	`13312`
number of epochs	`3`
per-device batch size	`1`
gradient accumulation steps	`16`
effective batch size	`16`
gradient checkpointing	✓
warmup ratio	`0.03`
learning rate scheduler	`cosine_with_min_lr`
minimum learning rate ratio	`0.1`
Quantization	`MXFP4, dequantized`
LoRA configuration	
rank	`8`
scaling factor	`16`
target modules	`all-linear`
target parameters	`MoE layers 7, 15, 23`
Compute details	
GPU	`NVIDIA A100 80GB`
number of GPUs	`1`
precision	`bfloat16`
training time	`~ 7 hours per epoch`
total GPU hours	`~ 21 GPU-hours`

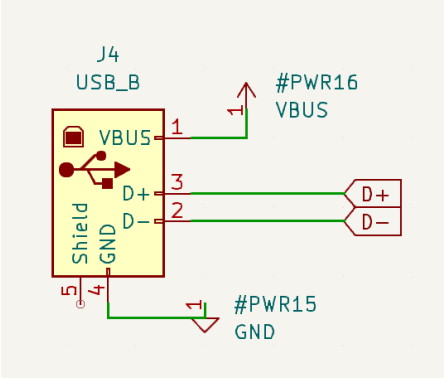


(a) User Request–"I want a 1uF capacitor connected between +5V and GND for power supply decoupling."

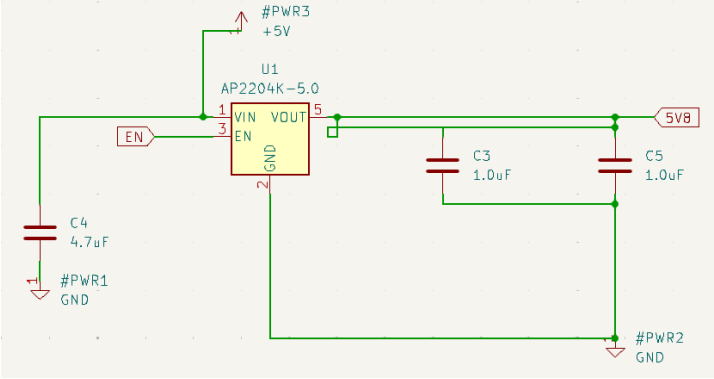


(b) User Request–"I want a 3.3V AVR ISP-6 programming header exposing MISO, MOSI/SDA, SCK/SCL, RESET, VCC, and GND to program a microcontroller."

Figure 6: Examples of using SchGen 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 7: Examples of using SchGen to generate a schematic based on users' requests over unseen chips.