SCHGEN: PCB SCHEMATIC GENERATION WITH SEMANTIC-GROUNDED CODE REPRESENTATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Printed circuit board (PCB) design is fundamental to nearly all electronic devices and has a growing demand in embodied AI, mobile, and Internet-of-Things applications. However, the schematic design process remains expertise-intensive and time-consuming. Large language models (LLMs) have accelerated software development recently, but extending them to hardware design is challenging due to domain-specific gaps— notably the absence of large, high-quality datasets and effective representations that capture circuit semantics. This paper presents SchGen, a large language model for PCB schematic generation that lowers the barrier to creating custom, novel hardware. To address the gaps above, we curate a comprehensive dataset of PCB schematic designs and propose a novel semantic-grounded code representation, which effectively encodes spatial arrangement and wire connections of components, making schematic designs amenable to generative modeling. Experimental results demonstrate that SchGen generates high-quality schematics with superior performance on wire connectivity and spatial arrangement over baselines. Our work paves the way for transforming hardware design from a manual task into an automated process with generative AI.

1 INTRODUCTION

Printed circuit boards (PCB) are the foundational component of nearly all electronic hardware, driving innovation from daily personal electronics to emerging embodied AI hardware. The demand for customized PCB designs for novel applications and diverse AI embodiment is accelerating. However, conventional design workflow remains largely manual, requiring significant time and domain expertise to operate Electronic Design Automation (EDA) tools. Based on design requests, engineers draw the schematic by manually specifying circuit components (symbols) and their interconnections (wires), similar to how software developers weave together code modules and their interfaces. Then engineers export the schematic file as a netlist file, which informs the PCB layout. The layout result is then sent to the factory for the physical circuit board fabrication. Among these stages, schematic design is the first yet critical step that architects the hardware system by arranging and connecting circuit components, which is also the least automated step due to its large combinatorial design space and intellectual demand on domain knowledge (Figure 1).

Recently, generative models have shown transformative capabilities for conventional circuit design processes. However, generating PCB schematic design based on user requests remains unexplored. To automate the design process of digital integrated circuits (ICs), large language models (LLMs) have been explored to generate boolean logic representations described with high-level hardware description languages, like Verilog, VHDL (Wu et al. (2024); Thakur et al. (2024); Fu et al. (2023)). Analog ICs, on the other hand, depend on circuit topology to achieve desired performance, leading to graph structure representation proposed in CktGNN (Dong et al. (2023)). AnalogCoder (Lai et al. (2025)) and LaMAGIC (Chang et al. (2024)) leverages language models to generate analog circuits, while AnalogGenie Gao et al. (2025) presents a comprehensive dataset for LLM pre-training to enable novel circuit topology discovery. These representations in prior work are well-suited for digital and analog circuits respectively, but are unable to support our task. PCB schematic connects both digital and analog IC components and other components like resistors, capacitors, connectors. The complex wiring connections among heterogeneous components are unseen in prior work. Moreover, instead of specific digital logic or analog performance, a schematic design aims to achieve a high-level function, typically specified in the form of natural language.

There are several challenges for generating PCB schematic designs. First, we lack an efficient learning representation for this complex task. While digital and analog ICs have mature code-based hardware representation, schematic design is predominantly a visual and spatial task involving selecting, placing, and connecting components with GUI software. As shown in Figure 2, existing representations of schematics are poorly suited for generation by standard language models. Although schematic files are stored as structured text, they contain verbose, redundant, and tool-specific metadata. As a result, LLM-generated schematic files suffer from formatting errors and are unusable for follow-up design steps. Meanwhile, schematic images generated by LLMs are not machine-readable and cannot be parsed or

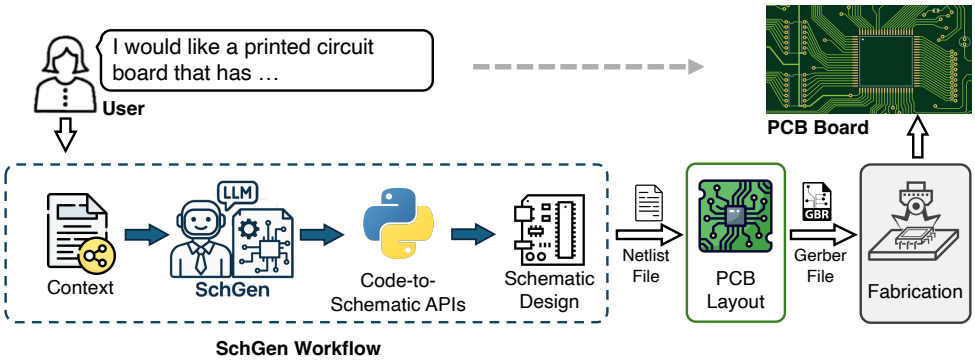


Figure 1: Overview. Based on the user request, SchGen generates PCB schematic design using custom code representation, which is then converted to netlist for PCB layout and fabrication.

converted into netlists. Second, schematic generation is inherently a spatial reasoning task, which requires placing components and ensuring interpretable wiring. Standard language models struggle to capture this due to the lack of spatial understanding. Lastly, there is a severe lack of dataset for PCB schematic design purpose. Although many open-source hardware designs are publicly available, they exist in various formats, and most are available as PDF images—easily interpretable by humans but unsuitable for training language models that require tokenized data.

We present SchGen, a large language model for generating PCB schematic based on input prompts, as shown in Figure 1. We introduce a novel semantic-grounded code representation that captures both the functional intent of schematic editing operations and the rationales behind spatial placement and interconnections of circuit components (symbols). Specifically, we build a concise set of schematic editing functions that captures the semantics of fundamental editing operations, including adding symbols, connecting pins, getting pin locations. This allows us to generate code that matches the editing steps of human engineers and remove unnecessary information, which enables LLM to learn efficiently by utilizing their general domain knowledge during pre-training. For spatial reasoning, we note that engineers do not interpret a schematic by their absolute coordinates or the graph topology of circuits, instead they rely on the relative offsets and semantic connections between symbol and pin names. For example, the VCC pin of ESP32 chip is for power, so we should place and connect a decoupling capacitor 10mm left of this pin. Instead of relying on absolute coordinates as in the raw schematic files, we design our code APIs to allow setting up local coordinate systems to place circuit symbols by relative offsets and connecting wires by pin names. This converts a difficult spatial reasoning task to a semantic matching task, which can be more efficiently handled by LLM training.

To tackle the lack of dataset, we curate a comprehensive dataset containing pairs of user requests and schematic code representations. Most online open-source designs do not provide source files and only sharing images that do not support editing or parsing. We present an agent-human collaboration approach of replicating open source PCB schematic designs as *KiCAD* schematic files. We further synthesize user request corresponding to the schematic designs. Then, we develop a schematic-to-code converter that generates python code that can recreate the schematic designs using our schematic editing APIs above, as our code representation is equivalent to raw schematic designs. Built on our code representation and dataset, we train SchGen by finetuning the gpt-oss-20b (Agarwal et al., 2025).

This paper makes the following key contributions: (1) We present SchGen, the first generative model that enables automated PCB schematic generation using domain-specific code according to user request prompt. (2) We propose a semantic-grounded code representation that efficiently captures schematic editing operations, and spatial arrangement of circuit symbols. (3) We collect the first comprehensive dataset of PCB schematic designs by converting online image designs to code representation. Our novel human-agent collaborative pipeline enables scalable data acquisition from online open-source designs. (4) Experiment results show that SchGen generates circuits with better spatial arrangement and higher accuracy of wire connections when compared to using other representations. SchGen also outperforms leading LLMs with much larger parameter sizes even when they are prompted to use the same representation.

2 Preliminaries and Related Works

2.1 PCB Schematic Design Process

PCB schematic design process depends on computer software tools, i.e., EDA software, including proprietary software like Altium Designer (2025), Cadence Design Systems (2025), EAGLE (2023), and open-source solution like KiCad

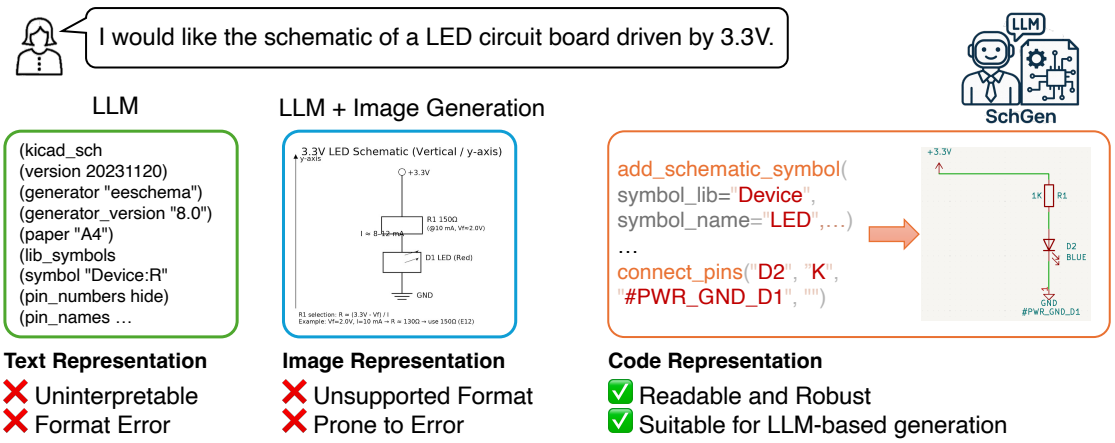


Figure 2: Comparison of generating schematic with different representations

(2025). They have similar schematic content and editing operations, but use very different schematic design file format. We use *KiCAD* schematic format throughout this paper as it is open source.

A schematic file includes three main types of items: (1) component symbols that represent physical circuit components such as chips, resistors, capacitors (2) power symbols that represent power rails such as VCC for power source, GND for ground (3) net labels that represent named net connections. Labels with the same name are considered as connected together in the same circuit net; labels are often used to mark the input and output interfaces of a hardware module, so that multiple modules are automatically connected through labels that share names. Each component symbol has one or multiple pins, each representing one electrical pin of the physical chip or other components; each power symbol and net label typically has one pin to allow wire connection. We identify a specific pin by either a pin ID number or a concise name for ease of human understanding, for example, a microcontroller can have a VCC pin for power supply, a GND pin for ground, and GPIO1, GPIO2, etc. for general purpose input/output pins, a LED component can have 'A' pin for anode and 'K' pin for cathode. We draw wire connections between pins of these symbols and labels in the schematic to mark electrical connections.

The design process typically begins with choosing the types and numbers of symbols. Then, the engineer places them on the schematic to form a proper spatial layout, which is followed by making wire connections among pins of the symbols. 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 symbols, and connect 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) as the task is relatively simpler, while the schematic design process is still relying on the unscalable manual process. There have been explorations with code-based PCB design, such as SKiDL, but it focuses on generating netlist directly based on python code, which skips the schematic design step. As a result, it does not provide a user-friendly, interpretable schematic visual and still relies on manual coding. Because visual feedback and reasoning is important for hardware design, making a schematic design before generating netlist is still the mainstream practice for hardware engineers.

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, Wu et al. (2024); Thakur et al. (2024); Fu et al. (2023) explore 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 for designing analog circuits 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 and develops a sequence-based graph representation to enable LLM pre-training for novel circuit topology generation. Another recent exploration is to generate 3D CAD models for mechanical parts and assemblies. Wu et al. (2021) describes a shape as a sequence of computer-aided design (CAD) operation and presents a dataset. Wang et al. (2025) introduces a visual feedback stage, which utilizes LVMs to provide feedback scores during CAD generation model training, while Alam & Ahmed (2025) converts raw image input to editable parametric CAD sequences through a contrastive learning framework.

Table 1: Comparison of different representations.

Representation	Example	MDL	LZ Norm (mean / median)	Token Length

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")`
		2.73/2.61	**1.64/1.62**	1143.1/830

Code-L2	`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")`
		3.00/2.91	1.80/1.82	959.9/693

Code-L3	`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])`
		2.86/2.75	1.76/1.76	821.2/608

KiCAD File	`(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)))) ...`	1.60/1.62	1.22/1.23	10609.6/5765.0

These prior works each presents a unique representation for the specific design tasks, while our task of schematic generation is not explored yet, lacking both representation and datasets. Recent works (Huang et al., 2025; Xu et al., 2025) has made some progress on conversion from circuit diagram images to netlists and Matsuo et al. (2024) focuses on netlist-to-schematic translation, which is different from our task of generating schematic designs from user requests.

2.3 Open-Source Datasets of Hardware Designs

The development of AI-based hardware generation depends on training on large-volume and high-quality datasets. Previously, circuit datasets provided by Kunal et al. (2019); Dong et al. (2023); Gao et al. (2025) have offered plentiful resources regarding analog circuits, while Wang et al. (2025); Wu et al. (2021); Xu et al. (2022) create datasets of human-annotated 3D objects for CAD. However, we still lack structured PCB schematic design datasets for our task. To tackle these fundamental problems, we have constructed a diverse dataset of more than 2400 schematics using the open-source online design resources such as datasheets and hardware designs on *Sparkfun*(SparkFun Electronics, 2025), as reference. To ensure high-quality dataset samples, we build a dataset collection framework including both LLM auxiliary and human annotations, which presents a scalable and efficient approach for design collection.

3 Approach

SchGen is a novel language model aiming to generate PCB schematic designs based on user requests. We first introduce our code representation for schematic designs, which captures the semantics of schematic editing operations and the rationales behind spatial placement and interconnections of circuit symbols. We quantitatively demonstrate that our code representation is more structured and easier to learn than other representations. Built on this, we construct a dataset of schematic designs by converting web PCB designs to code. We propose an agent-human collaboration approach pipeline to enable scalable data acquisition and synthesize corresponding user requests. Finally, we perform the training of SchGenwith the constructed dataset with paired user requests and code representations.

3.1 Code Representation for Schematic Designs

Although existing LLMs claim to have the ability to generate PCB schematic designs when asked in 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 schematic is not learning efficient for LLMs. Figure 2 shows illustrative comparison of 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 symbols and show random format, making it infeasible to convert them into valid schematic files.

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 symbols that represent the core components of the circuit, then arrange other symbols 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 symbols, and GND pins are connected to ground symbols. 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 are 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.

Based on the observation, 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_reference, pin_name):
def connect_pins(symbol_ref_a, pin_name_a, symbol_ref_b, pin_name_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()* connect two pins according to the symbol reference names and pin names of the two pins. Finally, *write_out_all_wires()* writes out all wires on the schematic with specified connections, performing a basic automatic routing. Figure 3 illustrates code snippets and corresponding parts on the schematic.

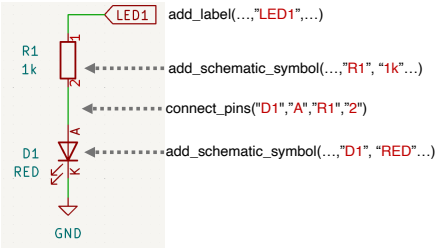


Figure 3: Illustration of the proposed code representation of schematics.

LLMs' ability to understand spatial relationships is relatively limited (Yamada et al., 2024). For our task, the schematic may involve dozens of symbols/label distinct 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 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 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.

To summarize, our code representation captures the key editing steps, meanwhile allowing establishing local coordinate systems based on pin locations and connecting wires between two pins based on pin names. As no existing python package provide such APIs, we implement these APIs by parsing and editing the symbolic expression format in the raw *KiCAD* schematic design files. Although the underlying file is still standard schematic file, we hide the complexity by our designed API abstractions.

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. Moreover, we compare with *KiCAD* raw schematic file in text format. We use the following metrics to evaluate these representations over our dataset:

(1) Minimum Description Length (**MDL**) (Grünwald, 2007). We use lossless compression to approximate data complexity; lower bits-per-byte (BPB) means the representation is more structured and thus easier to model.

$$\text{MDL} = \frac{8 \cdot \text{compressed\_bytes}}{\text{raw\_bytes}} \tag{1}$$

(2) Lempel–Ziv Complexity (**LZ Norm**) (Lempel & Ziv, 2003). We measure the rate at which new phrases appear in a sequence under LZ parsing, where $c(n)$ is the number of phrases produced by incremental LZ parsing; a lower normalized value implies more patterns and lower intrinsic complexity.

$$\text{LZ\_norm} = \frac{c(n)\ \log n}{n} \tag{2}$$

Figure 4: Pipeline of constructing the dataset used to train SchGen

(3) Token Length. We use the tokenizer in *BERT* (Devlin et al., 2019) to calculate the average token length.

Comparing three code-based representations, **MDL** and **LZ Norm** of the proposed representation are significantly lower than the other two, which means it has a more structured text pattern. *KiCAD* files have even lower values of the above metrics because their fixed format is verbose and include redundant metadata. However, *KiCAD* files have longer token length than code-based representations by an order of magnitude, which makes it very inefficient and can degrade LLM performance (Levy et al., 2024) in terms of output correctness. We train the LLM with the above four representations and evaluate the output correctness in Table 2.

3.2 Converting Open-Source PCB to Code Representation

Most online PCB designs provide image format schematic designs instead of editable source files, which avoids issues with EDA tool compatibility and version discrepancies. However, this brings challenges to constructing a dataset of code representations. Thus, we propose the pipeline in Figure 4 to convert the open-source PCB designs to code representations. We first introduce the agentic sketch module to acquire a draft (Section 3.2.1), which is later annotated by human engineers to fix the possible errors. Finally, we develop a schematic-to-code converter, which parses the aligned schematic as our python code representations for 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 5, we take online images of reference designs as the input and leverage multi-modal LLMs, e.g., GPT-5, to generate python code that calls APIs in Section 3.1. Through our underlying implementation, the compiled program outputs error and warning feedback in execution, e.g., wrong syntax, symbol not found. Then, multi-modal LLM iterates code generation based on the feedback, until no error or reaching max iterations, and then output a sketch version of the schematic file by executing the code.

The process above poses a challenging geometry-related visual task. Multi-modal LLMs often make mistakes when sketching the schematic even after iterations, especially when handling complex wire connections. For example, it is hard for LLMs to extract the locations of all symbols, or whether two wires are connected or intersect. Thus, we introduce manual editing to adjust the incorrect schematic designs to make it aligned with the online design reference. 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 hardware designs on *Sparkfun*(SparkFun Electronics, 2025).

3.2.2 Schematic-to-Code Conversion

Based on the schematic files acquired in the previous step, we develop a schematic-to-code converter to parse the schematic and generate the equivalent code representation following Section 3.1. *KiCAD* schematic files are stored as trees of *symbolic expressions* (s-expressions), which can be parsed by python. We abstract the topology of a schematic as one or multiple undirected graphs, where pins and wires are considered as vertices and edges, respectively. By operating a graph traversal from each pin vertex, we can acquire all paired pins and connected components in the graph. For multiple pins of the same symbol, we consider they as connected.

Using our code APIs, we follow a common coding pattern to generate the code that re-produces the schematic. Given the symbols in an undirected graph, we first select the symbol with the most pins as the center symbol of that block, i.e., setting up reference point of the local coordinate system. Then, we calculate the offsets of other symbols, based on which we sort them in ascending order and call *add_schematic_symbol* to place symbols accordingly. Similarly, we place labels by calculating their offsets with respect to the pins they are attached to and calling *add_label* to place them. Finally, we traverse all paired pins in the graph and call *connect_pins* to connect them and attach *write_out_all_wires* at the end. A sample schematic design and the corresponding three levels of representations are shown in Appendix A.2.

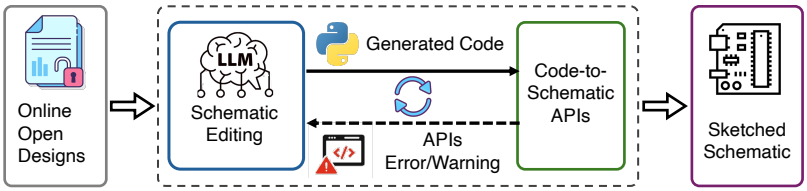


Figure 5: Agentic sketch module

3.3 Dataset Augmentation and SchGenTraining

As shown in Figure 1, our goal is to build a model that can generate a schematic based on the user's request. To this end, 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 care 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 showing two different request styles of the schematic in Figure 7 are shown in A.3.

We use GPT-oss-20B (Agarwal et al., 2025) as the base model of SchGen 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 reasoning distilled from reasoning models, following the approach of prior work (Ho et al., 2022; Guo et al., 2025; Chen et al., 2025). Specifically, we synthesize the thinking process by calling both the larger reasoning model GPT-oss-120B (Agarwal et al., 2025) and GPT-oss-20B itself, prompting them to generate the chain-of-thought reasoning that leads to the output from the input request.

4 Results

4.1 Experiment Setup

Dataset and training setup. Our proposed dataset 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. 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. We randomly select 500 samples as the test set, and use the remaining as the training set.

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 Hu et al. (2021). The training setting and parameters are shown in the Table 4 in Appendix A.5. We run training and inference on a single 80 GB Nvidia A100 GPU.

Evaluation metrics. We evaluate the performance of SchGenfrom 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, 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 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 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 $\boldsymbol{v}_i = (\text{symbol}, \text{pin})$. In the netlist file, each node is assigned to a net where nodes from the same net are bind together. We can represent each net as a set $\mathbb{N} = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \cdots, \boldsymbol{v}_n\}$, 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 $\mathbb{G}_{gt}$. We calculate the accuracy with

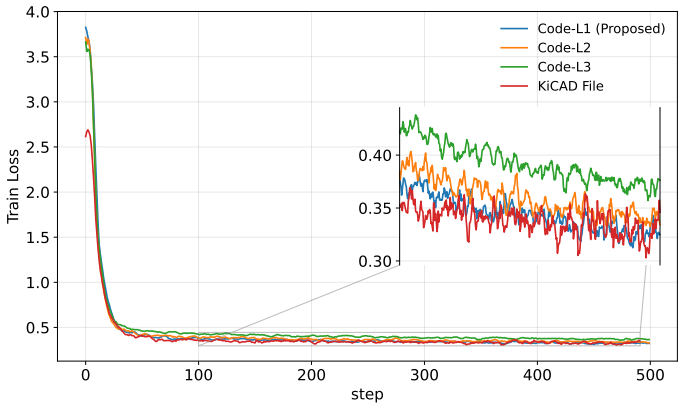


Figure 6: Train loss of finetuning *gpt-oss*-20b with four different levels of code representation.

Jaccard, *Precision* and *Recall*, using the following equations

$$\text{Jaccard}(\mathbb{G}_{gen}, \mathbb{G}_{gt}) = \frac{|\mathbb{G}_{gen} \cap \mathbb{G}_{gt}|}{|\mathbb{G}_{gen} \cup \mathbb{G}_{gt}|} = \frac{|\mathbb{G}_{gen} \cap \mathbb{G}_{gt}|}{|\mathbb{G}_{gen}| + |\mathbb{G}_{gt}| - |\mathbb{G}_{gen} \cap \mathbb{G}_{gt}|} \tag{3}$$

$$\text{Precision} = \frac{|\mathbb{G}_{gen} \cap \mathbb{G}_{gt}|}{\mathbb{G}_{gen}}, \text{Recall} = \frac{|\mathbb{G}_{gen} \cap \mathbb{G}_{gt}|}{\mathbb{G}_{gt}} \tag{4}$$

Baselines. To show the effects of different schematic representations after supervised fine-tuning, we compare the performance of SchGen model with 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 finetuned with the dataset using Code-L3 representation; 3) *gpt-oss*-20b finetuned with the dataset using the raw *KiCAD* schematic file representation. Moreover, using the testing set of the proposed dataset as the benchmark, we compare the performance of SchGen with *gpt-oss-20b* base model and three other recently released models with much larger model sizes, including *gpt-o4mini* (OpenAI, 2025b), *gpt-5* (OpenAI, 2025a), and *grok-3* (xAI, 2025). We prompt these unmodified models with descriptions of different schematic representations and give one example usage of the representation.

4.2 Main Results

Training loss comparison. Figure 6 shows the training loss of different representations under the same setting. Our proposed representation shows lower loss compared to the alternative code formats. This suggests superior learnability, an observation consistent with the structural metrics presented in Table 1. Although the *KiCAD* file format provides slightly lower loss, we attribute this to the model fitting on superficial textual patterns within the rigid file structure, rather than capturing the core design semantics buried in the long text. This failure to grasp essential design logic is evidenced by the inferior performance of the model finetuned with *KiCAD* file format, as shown in Table 2.

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

Method	Valid Circuits ↑	Spatial Violation ↓	Netlist Accuracy (%)↑		
	(Pass ratio%)	(Number of overlaps)	Jaccard	Precision	Recall
Code-L1 (SchGen)	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

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 SchGen with 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

Table 3: Performance comparison of different models

Method	Valid Circuits ↑	Spatial Violation ↓	Netlist Accuracy (%)↑		
	(Pass ratio%)	(Number of overlaps)	Jaccard	Precision	Recall
SchGen	**82.40**	**5.83**	**59.48**	**60.74**	**59.31**
Base Model (gpt-oss 20b with L1)	65.20	6.84	46.61	47.30	46.26
GPT-o4mini-L1	69.80	7.30	53.05	52.97	51.75
GPT-o4mini-L2	68.20	7.74	52.98	52.83	51.80
GPT-o4mini-L3	64.00	7.59	49.56	48.33	50.77
GPT-o4mini-*KiCAD*	6.62	-[1]	0.00	0.00	0.00
GPT-5-L1	72.60	7.46	53.30	54.77	53.94
GPT-5-L2	72.80	7.17	51.82	52.68	51.32
GPT-5-L3	73.00	7.91	50.26	49.93	53.42
Grok-3-L1	75.80	6.98	54.81	55.64	53.93
Grok-3-L2	77.00	7.28	51.14	53.96	50.60
Grok-3-L3	57.40	10.82	46.25	45.29	47.26

file formats. Comparing the three code representations, Code-L1 outperforms Code-L2 and Code-L3 in netlist accuracy, agreeing with the training loss results in Figure 6 and analysis in Table 1. Code-L1 (SchGen) 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.

Comparison of different models. We use our testing set as the benchmark to evaluate the performance of SchGen versus other language models, including gpt-oss-20b base model, *gpt-o4mini*, *gpt-5*, and *grok-3*. For models not finetuned on our dataset, we prompt them with descriptions of the schematic representation and provide one example usage of the representation. 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 performance of models prompted by 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 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 it. We also prompt o4-mini for *KiCAD* file and the results show low pass ratio and netlist accuracy.

Table 3 also shows the differences among LLMs on this new benchmark task. We see that *grok-3* achieves the best performance regarding the Code-L1 representation, while *gpt-5* has slight advantages with the other two representations. This implies their advanced ability to handle complex spatial relations. Nevertheless, the superior performance of SchGenshows that performing training on our dataset with the right representation is crucial for the schematic generation task, outperforming even much larger models.

Generalization. We remark that the SchGen 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 finetuned 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 users' needs. We provide the visualization of two novel schematic examples in Figure 10 of Appendix A.6, where we also put two samples of successful examples in the test set in Fig. 9.

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 structured sequence of interpretable editing function calls, 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 this new task. Experimental results demonstrate that SchGen substantially outperforms baseline representations and strong general-purpose models in terms of valid circuit generation, spatial arrangement, and netlist accuracy. These findings highlight the promise of domain-specific representations and datasets in bridging LLMs with hardware design. Looking forward, we envision SchGen as a foundation for broader automation in electronic design, paving the way toward fully generative, user-driven hardware creation.

[1]The number of overlaps is not available because valid circuits are too few.

6 Reproducibility Statement

The construction of the dataset we use has been elaborated in Sec. 3.2, while the setup of training are illustrated in Sec. 3.3 and Sec. 4.1. We also provide the parameter setup in Table 4 in Appendix A.5. The dataset is included in the supplementary material with its statistics in Fig. 8 in Appendix A.4. The source code is under preparation and will be made public on Github in the future.

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.

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.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. URL https://arxiv.org/abs/1810.04805.

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)*, pp. 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.

Peter D Grünwald. *The minimum description length principle*. MIT press, 2007.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 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)*, pp. 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, pp. 379–387, 2025.

Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on information theory*, 22 (1):75–81, 2003.

Mosh Levy, Alon Jacoby, and Yoav Goldberg. Same task, more tokens: the impact of input length on the reasoning performance of large language models, 2024. URL https://arxiv.org/abs/2402.14848.

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, pp. 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.

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

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

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. 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)*, pp. 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)*, pp. 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)*, pp. 6772–6782, October 2021.

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

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.

Xiang Xu, Karl D. D. Willis, Joseph G. Lambourne, Chin-Yi Cheng, Pradeep Kumar Jayaraman, and Yasutaka Furukawa. Skexgen: Autoregressive generation of cad construction sequences with disentangled codebooks, 2022. URL https://arxiv.org/abs/2207.04632.

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)*, pp. 1366–1371, 2020. doi: 10.1109/ICCA51439.2020.9264558.

A Appendix

A.1 LLM Usage Statement

We acknowledge the use of large language models for light polishing of some manuscript text. Specifically, LLMs were used to assist refining the language, improving the readability and clarity of the paper. We declare that the LLM does not participate in making idea, experiment code and data processing. All research ideas, experiment and analyses were conducted by the authors.

A.2 Example of Schematic Design and Code Representations

Fig. 7 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.

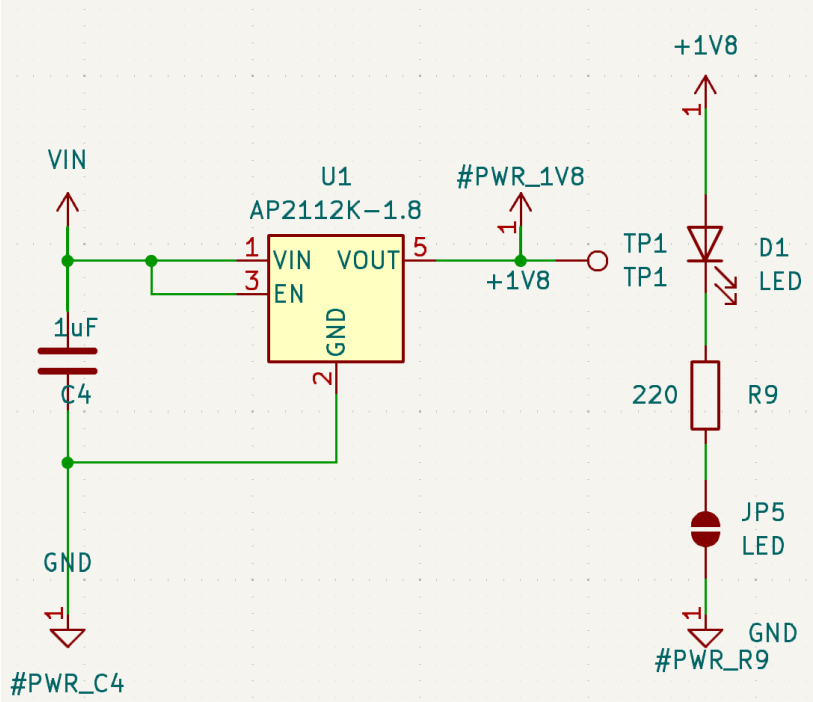


Figure 7: An example schematic designed in *KiCAD*

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 = 120.650, 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 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 : 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 connect 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()
```

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()
```

A.3 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'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 +1
    V8 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"}]}
```

A.4 Dataset Statistics

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

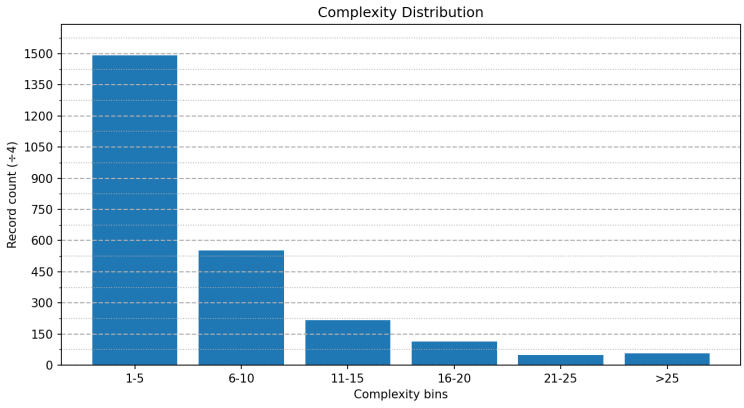


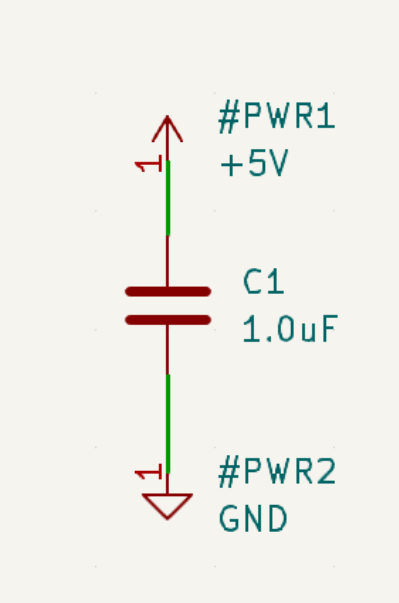
Figure 8: Histogram of the complexity distribution of the dataset

Table 4: Supervised finetuning and LoRA settings

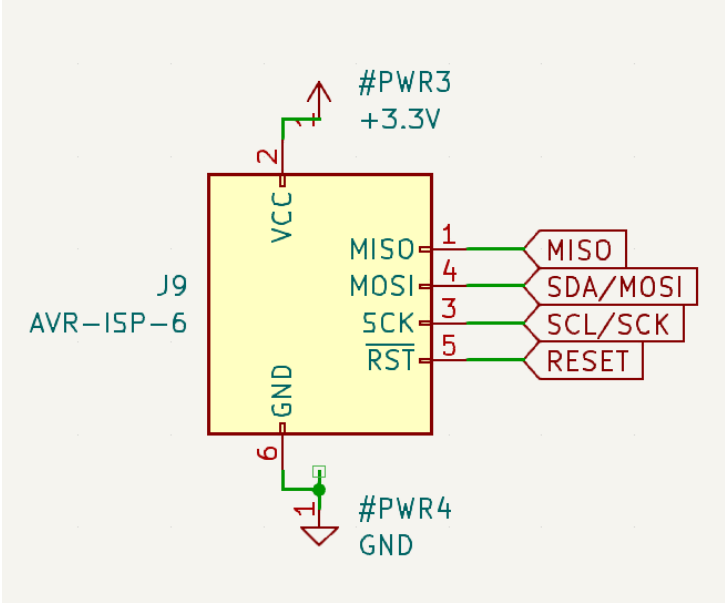
hyperparameter	value
SFT configuration	
learning rate	8e-4
assistant loss only	✓
max token length	13312
warmup ratio	0.01
learning rate scheduler	Cosine Annealing
Minimum learning rate	0.1
Quantization	4 bit
LoRA configuration	
rank	8
scaling factor	16
target modules	all-linear

A.5 Training Setup

A.6 Examples of generation

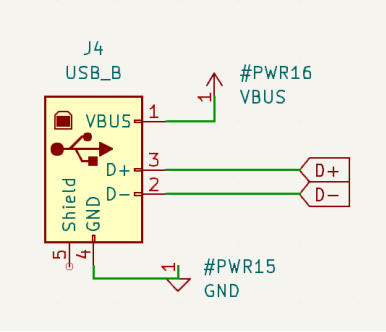


(a) User Request–"I want a 1uF capacitor connected between +5V and GND for power supply decoupling."

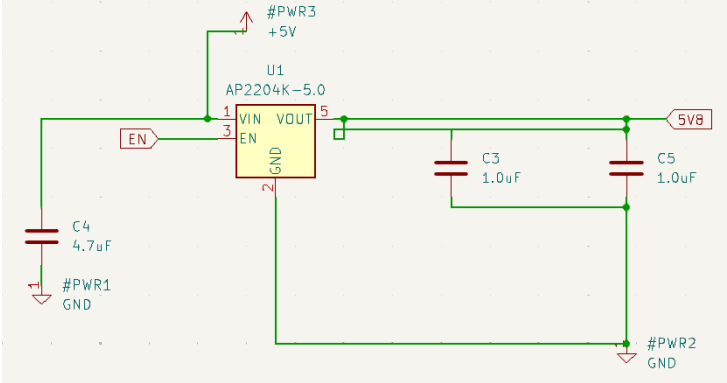


(b) User Request–"I want a 3.3V AVR ISP-6 programming header exposing MISO, MOSI/SDA, SCK/SCL, RESET, VCC, and GND to program a microcontroller."

Figure 9: 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 10: Examples of using SchGen to generate schematic based on users' requests over unseen chips.