SCHGEN: PCB SCHEMATIC GENERATION WITH SEMANTIC-GROUNDED CODE REPRESENTATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Printed circuit board (PCB) schematic design defines nearly all electronic hardware, but it remains mostly 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, a 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.

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 board following the steps of: (1) *Schematic Design*: Select circuit components (symbols) and specifying their interconnections (wires) which outputs a netlist file; (2) *Layout*: Based on the netlist file, determine the placement of components on the metal layers and add metal wires/via holes; (3) *Fabrication:* Produce the final Gerber file and send to factory to produce. Among them, Schematic design is the critical and challenging step that defines the hardware architecture. And we ask the question: *how should schematic construction be represented to support reliable generation by language models?*

Recent works start to explore circuit schematic generation from natural-language requests using agent framework. PCBSchemaGen (Zou et al., 2026) emphasizes constraint-guided iterative verification. CircuitLM (Hasan et al., 2026) combines component retrieval and staged reasoning to generate custom CircuitJSON. These training-free systems improve generation through domain knowledge prompting, structured intermediate formats, and inference-time orchestration. However, they provide limited analysis of how representation affects the joint generation of component placement, spatial schematic layout, and net-level electrical connectivity. In particular, the effects of such representation choices for model training remain unstudied.

We study the generation of editable PCB schematics from natural-language functional requirements. Given a user's request, the goal is to produce a schematic with correct electrical connectivity and a readable spatial layout that engineers can inspect and modify in an EDA tool. This problem presents two key challenges. (i) **Representation**. Unlike digital and analog IC design, where mature textual representations are widely used, PCB schematic design remains largely centered around visual

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

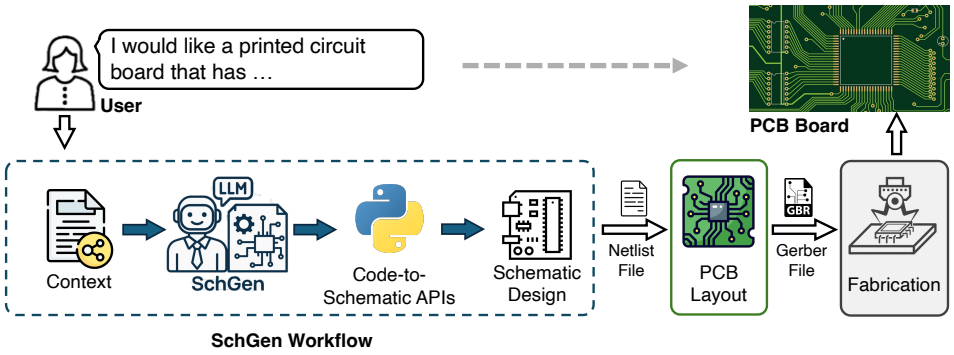


Figure 1: Overview of automated PCB design workflow. Based on the user request, SchGen generates a PCB schematic design using custom code representations, which is then converted to a netlist for PCB layout and fabrication.

operations such as selecting, placing, and connecting components. As shown in Figure 2, existing representations expose different limitations for language-model generation. Raw schematic files are dominated by verbose, tool-specific, and version-dependent syntax and metadata, requiring the model to generate low-level structural and geometric details in addition to circuit semantics. Image-based representations preserve visual appearance but are neither directly editable nor suitable for downstream electrical analysis. Netlists encode electrical connectivity but do not themselves specify component placement or wire geometry; these must be supplied separately to obtain a readable schematic. Consequently, generating an editable schematic requires a representation that captures both electrical intent and spatial organization without forcing the language model to explicitly predict unnecessary low-level geometry. (ii) **Data scarcity**. Although many open-source hardware designs are publicly available, they are distributed across heterogeneous formats and repositories and are often not paired with natural-language design requirements or executable construction procedures. This makes them difficult to directly reuse for supervised training of schematic-generation models.

To address the representation challenge, we introduce a semantically grounded executable representation of schematic construction which expresses component placement through offsets from local anchors and electrical connections through component references and pin names. To address data scarcity, we develop an agentic pipeline that reconstructs open-source PCB designs as editable *KiCad* schematics, synthesizes corresponding natural-language requests, and converts each design into executable construction programs to form a dataset.

On top of the code representation, we build SchGen, a model for generating editable *KiCad* schematics from natural-language requests. Using the dataset, we fine-tune two open-weight backbones, GPT-oss-20b (Agarwal et al., 2025) and Qwen-3.8 (Qwen Team, 2026). We also provide the representation's API specification and examples to general-purpose LLMs for prompted generation, which allows us to examine representation choices under both supervised fine-tuning and prompting.

We evaluate SchGen across two fine-tuned backbones and prompted general-purpose LLMs, measuring execution success, netlist accuracy, spatial layout quality, and expert-assessed functional correctness. Representation ablations show that proposed representation achieves the highest performance among the evaluated representations for both fine-tuned models with much smaller size compared to general-purpose frontier models. Further generalization study demonstrate that on out-of-distribution dataset, finetuned SchGen still maintains its advanced ability and achieves the highest performance.

Our contributions are threefold: (1) We introduce Code-L1, an executable schematic representation combining anchor-relative placement with named-pin connectivity, which we use for SchGen training. (2) We construct a dataset of 2,105 schematics spanning 1,390 unique designs, paired with synthesized natural-language requests and executable construction programs through an agent-assisted pipeline with human correction. (3) We evaluate representation choices across two fine-tuned backbones and prompted general-purpose LLMs, demonstrating improved netlist accuracy over alternative representations, meanwhile the finetuned small models achieve higher pass rate and netlist accuracy than general-purpose LLMs.

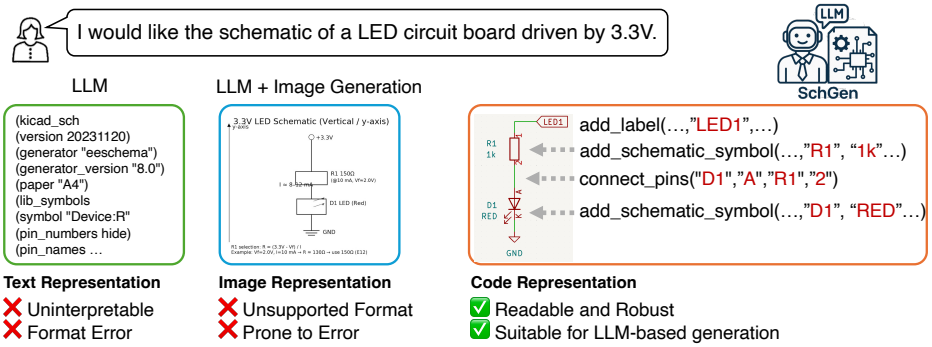


Figure 2: Comparison of schematic representations for LLM-based generation. Raw schematic files contain verbose metadata prone to formatting errors, while generated images are not machine-readable. Our code representation encodes editing primitives and pin-level semantics, transforming geometry prediction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

2 Preliminaries and Related Work

2.1 PCB Schematic Design Process

The 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 schematic design still heavily relies on labor-intensive, manual workflows. 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 Design

Built on hierarchically abstracted Boolean logic representations, e.g., Verilog and VHDL, recent studies(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.

Related work also studies translation between existing circuit representations. Netlistify (Huang et al., 2025) and Image2Net (Xu et al., 2025) recover netlists from schematic images, while Schemato (Mat-

suo et al., 2024) converts netlists into LTSpice schematic files. These tasks support circuit digitization and visualization, but begin with an existing design whose connectivity is specified or recoverable from the input. Generating a schematic from a natural-language request additionally requires determining components and connections that satisfy the stated requirements. Zou et al. (2026) proposes a feedback-based iterative framework for PCB schematic design. However, it evaluates on only 23 tasks, relies on SKiDL graph rendering that fails to scale to larger schematics, and lacks an open-source model implementation for direct baseline comparison. Similarly, while Hasan et al. (2026) introduces an agentic benchmarking framework for LLM-aided circuit design, its codebase is not publicly available, precluding a fair empirical comparison. Both approaches interface general-purpose LLMs with programmatic circuit libraries such as SKiDL (SKiDL, 2025) via multi-turn iterative prompting. However, they rely on automatic graph visualization engines that lack explicit spatial reasoning, producing tangled schematics on complex boards, and incur heavy multi-turn latency.

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. (Li et al., 2026) provides a comprehensive benchmark designed to systematically evaluate LLMs, but it includes mostly PCB-related visual question answering which cannot be used for the design task. Datasets for circuit topology generation and schematic conversion provide useful circuit structures or paired representations, but are not necessarily paired with natural-language requirements and executable schematic construction programs. To supply the paired forms for supervised request-to-schematic generation, we have constructed a dataset of 1,390 unique schematic designs curated from open-source hardware designs on SparkFun Electronics (SparkFun Electronics, 2025) under the CC BY-SA 4.0 license.

3 Approach

SchGen generates editable PCB schematics by translating natural-language requests into executable construction programs. We first propose the code representation named Code-L1 (Level 1), which specifies component placement and electrical connectivity through schematic editing operations. We then present a pipeline for constructing paired requests, schematics, and programs, followed by the use of this representation for supervised fine-tuning and prompted generation.

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* . Figure 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 clear rationales. 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.* 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.

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

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

```
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.

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, an editable valid *KiCad* schematic file is generated in *KiCad* tool, as illustrated in Figure 2.

Table 1 shows the comparison between different schematic representations. *Code-L2* replaces anchor-relative placement with absolute coordinates. *Code-L3* additionally replaces named-pin connection operations with explicit wire segments. These variants examine the effects of relative placement and of delegating wire construction through electrical endpoint specification.

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 & 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). (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

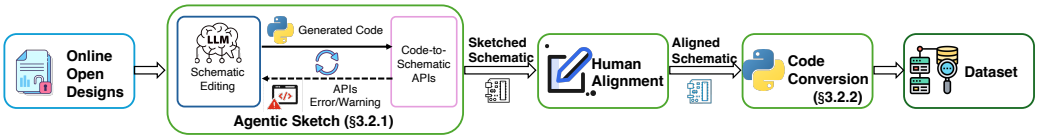


Figure 3: Pipeline of constructing the schematic dataset from online open-source PCB designs using LLM sketching and human correction.

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. 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.

Recall that our goal is to build a model that can one-shot 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.

3.3 Fine-Tuning and Prompted Generation

We separately fine-tune GPT-oss-20b and Qwen3.8-27B to generate schematic construction programs from natural-language requests. Both backbones use the same underlying request–program pairs, but receive backbone-specific synthetic chain-of-thought (CoT) supervision. Specifically, each backbone generates CoT traces for its own training examples, and these traces are incorporated into the corresponding training sequences. The CoT tokens contribute to the supervised training loss, providing explicit supervision for intermediate reasoning alongside schematic program generation.

At inference time, each model produces a construction program in a single generation attempt, after which the program is executed to obtain an editable schematic with no iterative generation or execution-feedback repair applied. Detailed training settings are provided in Section 6.5.

4 Experimental Validation

4.1 Experiment Setup

Dataset details. Our proposed dataset contains 2105 KiCad schematics from 1390 unique designs. The total sample number is doubled to 4210 using two styles of user request from two models as described in 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 partition the dataset at the schematic-design level to avoid possible leakage and randomly select 500 samples as the held-out test set, and use the remaining as the training set. For each model, we repeatedly run the experiments five times with different random seeds and calculate the corresponding standard deviation.

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-existent 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 conflicts, 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 have six experts evaluate the generated schematics using the rubric in Section 6.6. Symbol Error and Connection Error count incorrect/missing components and faulty pin connections (reported as averages). Functional Correctness reports the ratio of designs that can function as intended.[1]

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.

[1] Cross-validation among six experts shows high agreement, with correctness ratio differences below 3%.

Table 2: Performance comparison of SchGen finetuned with different base models and representations.

Method	Pass Rate ↑	Spatial Violation ↓	Netlist Accuracy (%)↑			Expert Verification		
	(Pass ratio%)	(Number of overlaps)	Jaccard	Precision	Recall	Symbol Error ↓	Connection Error ↓	Functional Correctness(%) ↑
Qwen3.8-27B								
Code-L1	96.0 ± 0.62	$\mathbf{5.08 \pm 0.12}$	$\mathbf{70.7 \pm 0.58}$	$\mathbf{76.3 \pm 0.65}$	$\mathbf{74.5 \pm 0.61}$	$\mathbf{0.02 \pm 0.01}$	$\mathbf{0.24 \pm 0.03}$	$\mathbf{82.6 \pm 0.54}$
Code-L1 (CoT masked)	95.2 ± 0.74	5.43 ± 0.15	67.6 ± 0.72	73.2 ± 0.80	71.2 ± 0.78	0.05 ± 0.01	0.41 ± 0.04	76.0 ± 0.61
Code-L2	87.6 ± 0.95	6.10 ± 0.18	50.9 ± 0.88	59.0 ± 0.94	54.0 ± 0.85	0.16 ± 0.02	1.15 ± 0.08	48.6 ± 0.71
Code-L3	$\mathbf{97.6 \pm 0.55}$	5.99 ± 0.16	19.6 ± 0.42	34.3 ± 0.68	20.3 ± 0.45	0.18 ± 0.02	4.92 ± 0.22	12.0 ± 0.46
KiCad File	21.8 ± 0.98	–	11.5 ± 0.41	13.4 ± 0.45	12.1 ± 0.43	1.15 ± 0.07	8.95 ± 0.32	4.60 ± 0.30
GPT-OSS-20B								
Code-L1	82.0 ± 0.86	$\mathbf{7.73 \pm 0.19}$	$\mathbf{49.1 \pm 0.65}$	$\mathbf{54.9 \pm 0.72}$	$\mathbf{52.8 \pm 0.68}$	$\mathbf{0.06 \pm 0.01}$	$\mathbf{0.61 \pm 0.05}$	$\mathbf{60.6 \pm 0.69}$
Code-L1 (CoT masked)	53.4 ± 1.12	8.32 ± 0.22	30.5 ± 0.84	33.0 ± 0.91	35.4 ± 0.88	0.98 ± 0.06	1.58 ± 0.09	14.0 ± 0.49
Code-L2	78.2 ± 0.90	7.77 ± 0.17	46.0 ± 0.78	52.6 ± 0.85	49.5 ± 0.82	0.28 ± 0.03	1.76 ± 0.11	33.0 ± 0.67
Code-L3	$\mathbf{86.4 \pm 0.94}$	8.14 ± 0.24	15.5 ± 0.48	24.8 ± 0.62	15.7 ± 0.51	0.24 ± 0.03	6.76 ± 0.28	6.00 ± 0.34
KiCad File	32.5 ± 1.25	7.80 ± 0.21	9.11 ± 0.35	14.5 ± 0.46	9.33 ± 0.38	1.22 ± 0.08	9.42 ± 0.35	3.00 ± 0.24

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* and *Qwen-3.8-27B* on the same dataset. To assess explicit reasoning supervision, we additionally train a *CoT masked* variant for each backbone. This variant retains the CoT traces in the training sequences but excludes their tokens from the supervised loss. We compare the resulting models with prompted general-purpose LLMs. 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 three larger LLMs, including *GPT-5.5* (OpenAI, 2026), *GPT-5.2* (OpenAI, 2025), and *Grok-4* (xAI, 2025). All models are evaluated in multiple-run setting on identical test inputs. To ensure fairness, we provide uniform prompts in the same test set, 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 shows that Code-L1 achieves the highest netlist accuracy and expert-assessed functional correctness, with the lowest spatial violations, for both backbones. Functional correctness reaches 82.6% for Qwen3.8-27B and 60.6% for GPT-OSS-20B, compared with 48.6% and 33.0% for Code-L2. Replacing pin-based connections with explicit wire geometry further reduces functional correctness to 12.0% and 6.00% for Code-L3. Although Code-L3 achieves higher pass rates, this is mainly due to the wire connection API error reporting: directly adding wire segments with coordinates does not check pin position are valid or not, which makes it easier to pass execution. Its substantially lower connectivity and functional scores demonstrate that passing execution and electrical-rule checks does not establish functional correctness. These results support combining relative placement with pin-based connection operations.

Effect of CoT supervision. Masking the CoT loss reduces netlist Jaccard from 70.7% to 67.6% for Qwen3.8-27B and from 49.1% to 30.5% for GPT-OSS-20B. Functional correctness decreases from 82.6% to 76.0% and from 60.6% to 14.0%, respectively. Explicit CoT supervision therefore benefits both backbones, with a larger improvement for GPT-OSS-20B. Since reasoning traces remain in the training sequences, this ablation measures the contribution of directly supervising CoT in the reasoning process over schematic connectivity logic.

Comparison with general-purpose models. Table 3 shows that SchGen with Qwen3.8-27B achieves the best reported results across all metrics on the in-domain benchmark, including a 96.0% pass rate, 70.7% netlist Jaccard, and 82.6% functional correctness. Compared with GPT-5.5 using Code-L1, it improves Jaccard and functional correctness by 24.9 and 27.6 percentage points, respectively. The GPT-OSS-20B implementation also exceeds all evaluated Code-L1 prompted baselines in netlist accuracy and functional correctness, reaching 49.1% Jaccard and 60.6% functional correctness.

Effectiveness under prompting. Code-L1 achieves the highest netlist accuracy and functional correctness among the evaluated representations for each prompted model. Functional correctness

Table 3: Performance comparison of different models. Bold marks the best values within each panel.

Method	Pass Rate ↑	Spatial Violation ↓	Netlist Accuracy (%)↑			Expert Verification		
	(Pass ratio%)	(Number of overlaps)	Jaccard	Precision	Recall	Symbol Error ↓	Connection Error ↓	Functional Correctness(%) ↑
SchGen(Qwen3.8-27B-L1)	$\mathbf{96.0 \pm 0.62}$	$\mathbf{5.08 \pm 0.12}$	$\mathbf{70.7 \pm 0.58}$	$\mathbf{76.3 \pm 0.65}$	$\mathbf{74.5 \pm 0.61}$	$\mathbf{0.02 \pm 0.01}$	$\mathbf{0.24 \pm 0.03}$	$\mathbf{82.6 \pm 0.54}$
SchGen(GPT-oss-20B-L1)	$\mathbf{82.0 \pm 0.86}$	$\mathbf{7.73 \pm 0.19}$	$\mathbf{49.1 \pm 0.65}$	$\mathbf{54.9 \pm 0.72}$	$\mathbf{52.8 \pm 0.68}$	$\mathbf{0.06 \pm 0.01}$	$\mathbf{0.61 \pm 0.05}$	$\mathbf{60.6 \pm 0.69}$
GPT-5.5-L1	$\mathbf{64.8 \pm 0.92}$	8.13 ± 0.18	$\mathbf{45.8 \pm 0.74}$	$\mathbf{49.1 \pm 0.80}$	$\mathbf{48.6 \pm 0.77}$	$\mathbf{0.08 \pm 0.01}$	$\mathbf{0.82 \pm 0.06}$	$\mathbf{55.0 \pm 0.70}$
GPT-5.5-L2	62.0 ± 0.98	16.4 ± 0.34	40.9 ± 0.82	44.2 ± 0.86	43.8 ± 0.85	0.30 ± 0.03	2.10 ± 0.12	31.0 ± 0.65
GPT-5.5-L3	58.6 ± 1.05	$\mathbf{7.30 \pm 0.17}$	37.2 ± 0.76	40.4 ± 0.81	40.6 ± 0.79	0.26 ± 0.03	6.20 ± 0.26	10.6 ± 0.43
GPT-5.5-*KiCad*	12.8 ± 0.85	–	5.87 ± 0.28	6.57 ± 0.31	6.24 ± 0.29	1.35 ± 0.08	9.80 ± 0.36	2.00 ± 0.20
GPT-5.2-L1	67.9 ± 0.88	$\mathbf{8.56 \pm 0.19}$	$\mathbf{43.0 \pm 0.71}$	$\mathbf{45.7 \pm 0.76}$	$\mathbf{49.5 \pm 0.78}$	$\mathbf{0.11 \pm 0.02}$	$\mathbf{0.94 \pm 0.07}$	$\mathbf{50.0 \pm 0.71}$
GPT-5.2-L2	59.3 ± 1.02	11.1 ± 0.26	35.5 ± 0.79	37.5 ± 0.83	40.1 ± 0.85	0.34 ± 0.03	2.45 ± 0.14	27.6 ± 0.63
GPT-5.2-L3	$\mathbf{68.0 \pm 0.86}$	9.55 ± 0.22	41.1 ± 0.75	43.4 ± 0.80	46.7 ± 0.82	0.28 ± 0.03	6.85 ± 0.28	9.00 ± 0.41
GPT-5.2-*KiCad*	10.5 ± 0.76	–	0.53 ± 0.12	0.88 ± 0.15	0.88 ± 0.15	1.48 ± 0.09	10.5 ± 0.38	1.00 ± 0.14
Grok-4-L1	$\mathbf{61.2 \pm 0.95}$	$\mathbf{8.82 \pm 0.21}$	$\mathbf{44.0 \pm 0.73}$	$\mathbf{49.8 \pm 0.82}$	$\mathbf{48.7 \pm 0.79}$	$\mathbf{0.20 \pm 0.02}$	$\mathbf{1.19 \pm 0.08}$	$\mathbf{47.0 \pm 0.71}$
Grok-4-L2	52.1 ± 1.08	10.7 ± 0.25	32.7 ± 0.76	34.6 ± 0.80	34.1 ± 0.78	0.42 ± 0.04	2.85 ± 0.16	23.6 ± 0.60
Grok-4-L3	53.1 ± 1.04	9.98 ± 0.24	35.9 ± 0.78	37.8 ± 0.83	37.3 ± 0.81	0.36 ± 0.03	7.42 ± 0.31	7.60 ± 0.37
Grok-4-*KiCad*	3.95 ± 0.45	–	1.32 ± 0.18	1.32 ± 0.18	1.32 ± 0.18	1.65 ± 0.10	11.2 ± 0.41	0.60 ± 0.11

Table 4: Performance on the OOD GitHub dataset. Bold marks the best values within each panel.

Method	Pass Rate ↑	Spatial Violation ↓	Netlist Accuracy (%)↑			Expert Verification		
	(Pass ratio%)	(Number of overlaps)	Jaccard	Precision	Recall	Symbol Error ↓	Connection Error ↓	Functional Correctness(%) ↑
SchGen(Qwen3.8-27B)	$\mathbf{90.7 \pm 0.72}$	4.55 ± 0.11	$\mathbf{56.7 \pm 0.65}$	$\mathbf{63.5 \pm 0.70}$	$\mathbf{60.3 \pm 0.68}$	$\mathbf{0.04 \pm 0.01}$	$\mathbf{0.38 \pm 0.04}$	$\mathbf{71.6 \pm 0.64}$
SchGen(GPT-OSS-20B)	83.2 ± 0.84	$\mathbf{3.63 \pm 0.09}$	44.2 ± 0.72	50.9 ± 0.78	47.7 ± 0.74	0.07 ± 0.01	0.68 ± 0.05	56.0 ± 0.70
GPT-5.2	62.1 ± 0.90	8.72 ± 0.20	40.5 ± 0.74	44.2 ± 0.79	42.7 ± 0.76	0.14 ± 0.02	1.06 ± 0.08	46.6 ± 0.71

reaches 55.0%, 50.0%, and 47.0% for GPT-5.5, GPT-5.2, and Grok-4, respectively, compared with 31.0%, 27.6%, and 23.6% using Code-L2. Code-L3 and raw *KiCad* generation perform substantially worse on functional correctness. These results extend the observed benefits of the representation to prompting without task-specific fine-tuning, although Code-L1 does not consistently achieve the highest pass rate or lowest spatial violations.

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 out-of-distribution (OOD) 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. As shown in Table 4, both SchGen implementations outperform GPT-5.2 across all reported metrics. The Qwen3.8-27B implementation achieves a 90.7% pass rate, 56.7% netlist Jaccard, and 71.6% expert-assessed functional correctness, compared with GPT-5.2's 62.1%, 40.5%, and 46.6%, respectively. It also produces fewer component and connection errors, indicating that its advantage extends beyond execution success to functional quality. The GPT-OSS-20B implementation achieves 56.0% functional correctness and the fewest spatial violations among the evaluated models. These results support the transfer of SchGen to schematics collected outside the training-data source, with improvements in connectivity, readability, and expert-assessed functionality on the evaluated GitHub designs.

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 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 representation design as the key enabler for hardware design generation.

AI USE STATEMENT

We used LLMs to assist with language and LaTeX polishing, synthetic dataset construction. The authors take full responsibility for the final manuscript, methods, results, and claims.

ETHICS STATEMENT

SchGen aims to make PCB schematic design more accessible, while recognizing that errors in generated circuits can cause hardware failures or physical harm. The reported execution checks, connectivity metrics, and expert assessments do not establish safety or reliability in physical deployment. Generated designs therefore require qualified engineering review and application-appropriate electrical validation before fabrication or use, particularly in safety-critical settings. The dataset is derived from publicly available hardware designs, including SparkFun designs described in the paper as licensed under CC BY-SA 4.0; redistribution of source and derived assets must preserve applicable attribution and license obligations. The study does not involve external human participants or the collection of personal or sensitive data; expert evaluation was performed by co-authors and external experts. The concentration on SparkFun designs and the use of synthetic requests may limit coverage of circuit families and real user requirements, so the reported results should be interpreted within the evaluated scope.

REPRODUCIBILITY STATEMENT

We document the schematic representation and editing APIs in Section 3.1, the reconstruction and conversion pipeline in Section 3.2, and request synthesis, reasoning-trace generation, and model training in Section 3.3. Section 4.1 describes the dataset split, evaluation metrics, expert assessment procedure, and baseline setup. The appendix provides schematic and code examples (Section 6.1), examples of both request styles (Section 6.2), dataset statistics (Section 6.3), training hyperparameters, and compute details (Section 6.5).

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.

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.

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.

Khandakar Shakib Al Hasan, Syed Rifat Raiyan, Hasin Mahtab Alvee, and Wahid Sadik. Circuitlm: A multi-agent llm-aided design framework for generating circuit schematics from natural language prompts, 2026. URL https://arxiv.org/abs/2601.04505.

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.

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.

Jindong Li, Lianrong Chen, Bin Yang, Jiadong Zhu, Ying Wang, Yuzhe Ma, and Menglin Yang. Pcb-bench: Benchmarking llms for printed circuit board placement and routing. In *International Conference on Learning Representations (ICLR)*, 2026.

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.

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-5.2. https://openai.com/index/introducing-gpt-5-2/, 2025. https://openai.com/index/introducing-gpt-5-2/.

OpenAI. Gpt-5.5. https://openai.com/index/introducing-gpt-5-5/, 2026. https://openai.com/index/introducing-gpt-5-5/.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026. URL https://qwen.ai/blog?id=qwen3.8.

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)*, 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.

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.

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)*, pp. 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 (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 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.

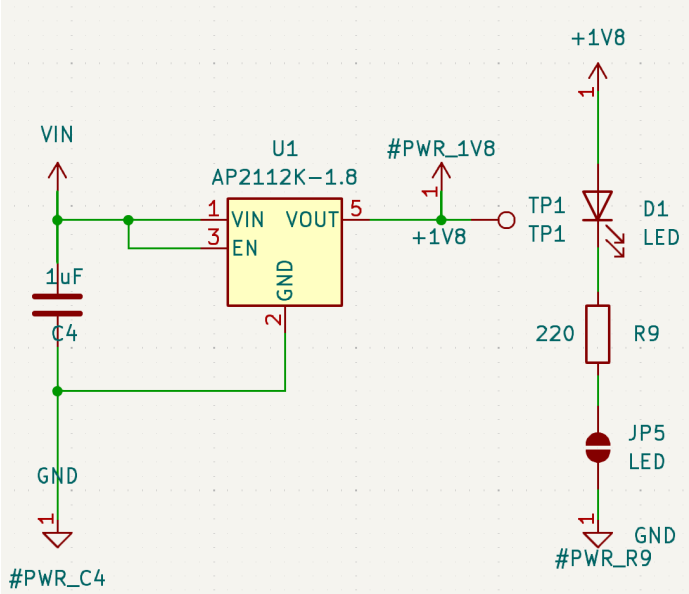


Figure 4: An example schematic designed in *KiCad*

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="1
    uF", 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.

6.4 Component Overlap with the GitHub Dataset

We audit component-symbol overlap between the main dataset, including its training and test splits, and the unseen GitHub dataset. After excluding universal resistor, capacitor, and inductor symbols, the main dataset contains 3,302 component instances across 542 unique component identities, while the GitHub dataset contains 985 instances across 394 unique identities.

As shown in Table 5, the datasets share 22 unique component identities, corresponding to 5.58% of the GitHub component vocabulary. Of the 985 GitHub component instances, 56 have identities that also occur in the main dataset, giving an instance-level overlap rate of 5.69%.

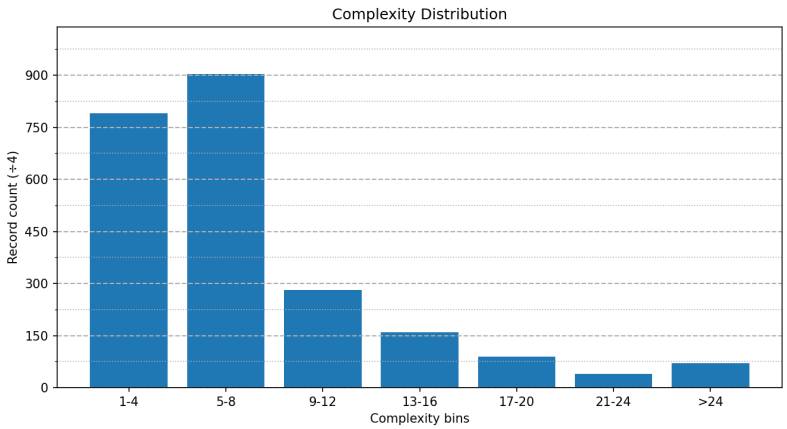


Figure 5: Histogram of the complexity distribution of the dataset

Table 5: Component-symbol overlap between the main and GitHub datasets after excluding universal resistor, capacitor, and inductor symbols. Overlap percentages use the GitHub counts as denominators.

	Main dataset	GitHub dataset	Overlap
Component types	542	394	22 (5.58%)
Component instances	3,302	985	56 (5.69%)

The remaining overlaps mostly consist of commonly used symbols in the *KiCad* library, such as switches and connectors. These results show that most non-universal component identities and instances in the GitHub dataset are absent from the main dataset, supporting evaluation on a largely different component vocabulary. This component-level audit does not, by itself, rule out overlap in circuit structures or reuse of learned circuit patterns.

6.5 Training Setup

6.6 Expert Evaluation Rubric

Evaluators review the user request, the generated schematic, the reference schematic, and relevant component datasheets when necessary. The reference schematic represents one valid implementation rather than the only acceptable solution; electrically equivalent alternatives that satisfy the request are accepted. Each schematic is assessed using the following three criteria.

Symbol Error. Evaluators count missing, incorrect, or duplicated components. When the user request does not specify an exact component, electrically suitable alternatives are accepted and are not counted as errors.

Connection Error. Evaluators count required functional pins that are unconnected, connected to an incorrect net, or involved in an unintended short. Electrically equivalent connectivity is accepted regardless of graphical wire routing or net-label naming. Differences in visual appearance alone therefore do not constitute connection errors.

Functional Correctness. Evaluators make a binary judgment of whether the complete schematic can reasonably perform the requested functions. Zero Symbol Error and Connection Error counts are necessary but not sufficient for a positive judgment. Evaluators additionally assess end-to-end functionality using their engineering experience and consult relevant datasheets when necessary. This criterion reflects expert assessment of schematic functionality, rather than measured operation of a fabricated board.

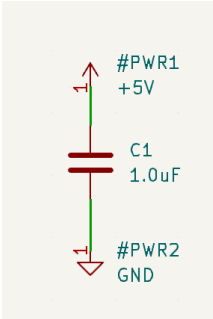
Symbol Error and Connection Error are reported as average per-schematic counts over the evaluated schematics. Functional Correctness is reported as the percentage of evaluated schematics receiving a positive judgment.

Table 6: Training configurations for Code-L1 full-parameter supervised fine-tuning. Qwen3.8-27B is fine-tuned on its language-model parameters, with the vision components frozen.

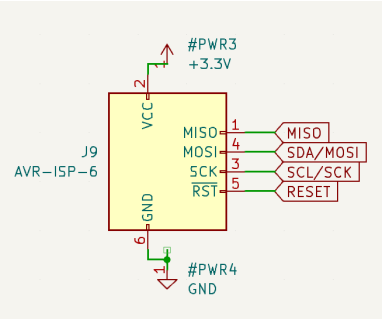
Setting	Qwen3.8-27B	GPT-OSS-20B
Training setup		
Training method	Full language-model SFT; vision frozen	Full-parameter SFT
Assistant-only loss	Yes	Yes
Maximum sequence length	16,384	13,312
Sequence packing	Yes	No
Epochs	2	2
Microbatch size per device	1	1
Gradient accumulation steps	8	2
Global batch size	32 packed sequences	32 examples
Gradient checkpointing	Full recomputation	FSDP activation checkpointing
Training quantization	None	MXFP4 source, dequantized to BF16
Optimization		
Optimizer	Adam (Megatron recipe); CPU offload	Fused AdamW
Adam (β_1, β_2)	$(0.9, 0.98)$	$(0.9, 0.95)$
Weight decay	0.1	0.1
Learning rate	2×10^{-5}	2×10^{-5}
Warmup ratio	0.05	0.03
Learning-rate schedule	Cosine with minimum LR	Cosine with minimum LR
Minimum learning rate	2×10^{-6}	2×10^{-6}
Minimum learning-rate ratio	0.1	0.1
Gradient clipping	1.0	1.0
Compute details		
GPU	NVIDIA A100 80 GB	NVIDIA H100 80 GB
Number of GPUs	16	16
Parallelism	TP 4 $\times$ DP 4	FSDP full-shard, 16 ranks

TP: tensor parallelism; DP: data parallelism; FSDP: fully sharded data parallelism. BF16 denotes bfloat16. The GPT-OSS-20B scheduler is `cosine_with_min_lr`.

6.7 EXAMPLES OF GENERATION

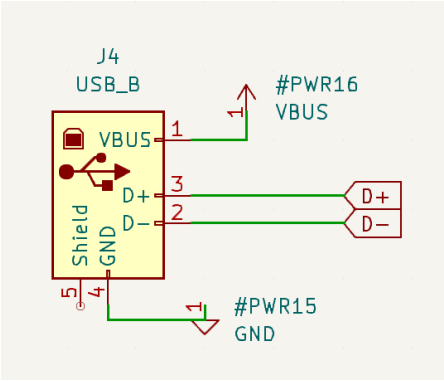


(a) User Request–"I want a 1uF capacitor connected between +5V and GND for power supply decoupling."

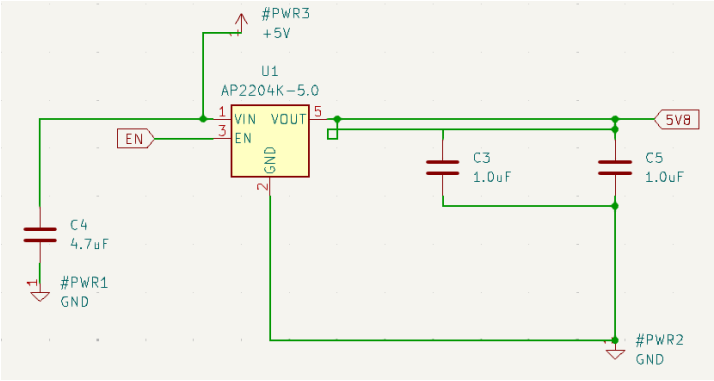


(b) User Request–"I want a 3.3V AVR ISP-6 programming header exposing MISO, MOSI/SDA, SCK/SCL, RESET, VCC, and GND to program a microcontroller."

Figure 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.