Smart-PCLib: A LLM-based Multi-Agent Framework for Automated PCB Component Library Generation

Zhaohai Di[1†], Jindong Tu[1†], Zhiyuan He[2], Yuan Pu[2], Jiawei Liu[2], Chong Tong[1], Tsung-Yi Ho[2], Bei Yu[2], Tinghuan Chen[1*]
[1]The Chinese University of Hong Kong, Shenzhen [2]The Chinese University of Hong Kong
chentinghuan@cuhk.edu.cn

***Abstract*—PCB design heavily relies on high-quality component libraries, while current library generation primarily depends on manual operation. To address the inefficiency and error-proneness inherent in this manual process, we propose Smart-PCLib, a novel multi-agent framework. It orchestrates a team of collaborative agents, each powered by a fine-tuned MLLM. These agents follow a structured workflow to extract data and generate code, governed by a robust verification-and-correction loop. A key innovation is PyPCLib, a Python-based domain-specific language (DSL) that reframes library creation as a structured code generation task, which not only improves reliability but also enables automated verification and modular design. Evaluated on a large-scale, diverse dataset, Smart-PCLib demonstrates high accuracy and efficiency, and its specialized agents outperform state-of-the-art general-purpose MLLMs on domain-specific tasks.**

I. Introduction

With the rapid advancement of the electronic industry, millions of new electronic components emerge each year. A comprehensive PCB component library, comprising symbols and footprints as shown in Fig. 1, serves as the foundation for PCB design [1]. Traditional component library generation relies heavily on manual extraction from datasheets, a process that is time-consuming and prone to errors. The sheer volume of new components makes it increasingly challenging to maintain up-to-date libraries through manual methods. Although there are some semi-automated library generation tools (such as SnapEDA [2] and FootPrintKu [3]), they lack appropriate verification mechanisms which may lead to inaccurate results. Additionally, the diverse formats of datasheets from various vendors and the differing requirements of electronic design automation (EDA) platforms add further complexity to the standardization process.

The emergence of large language models (LLMs) has revolutionized how we process and understand information. Advanced multimodal LLMs (MLLMs) [4] [5] [6] further enhance these abilities by processing and understanding multiple modalities, including text and visual information. This multimodal processing capability shows significant potential for PCB library generation, as key component information in datasheets, such as pin configurations and pad arrangements, is often presented in unstructured images, and charts.

However, applying off-the-shelf MLLMs directly to highly specialized documents like PCB component datasheets presents several challenges. Such documents are typically lengthy, highly unstructured, and packed with dense domain-specific knowledge [7]. Simply feeding an entire document to a general MLLM can lead to high computational costs and a loss of focus on critical details [8]. While MLLMs excel at high-level understanding, they often struggle with the precise geometric reasoning and intricate details required for PCB library design. For instance, extracting exact dimensions and spatial relationships from a technical diagram can be a significant limitation for a general-purpose model [9]. Furthermore, since these models are not inherently trained on the specific formats of specialized domains, they may produce inaccuracies or "hallucinations" when tasked with understanding and generating highly specific information [10]. Generating library files is a complex task, as it demands analysis of diverse information from different sections of the datasheet, followed by generating results that meet specific format requirements.

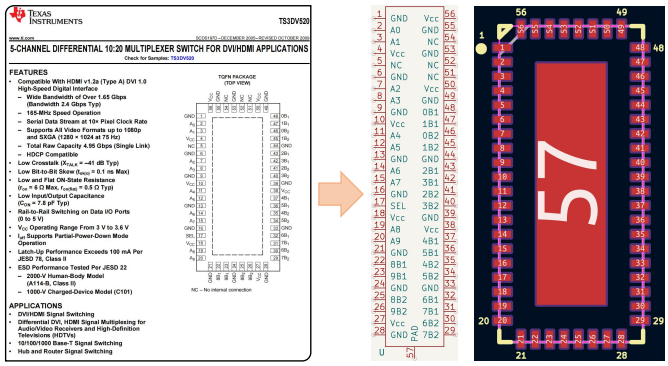


Fig. 1 Demonstration of PCB symbols and footprints generation.

To address these limitations, we propose Smart-PCLib, a multi-agent framework designed to automate the generation of PCB component libraries by dividing the complex process into specialized tasks. Each agent is powered by a fine-tuned MLLM to handle its specific role effectively. Datasheets are first preprocessed into a structured and concise format. The system then systematically extracts relevant information for library generation. We also propose PyPCLib, a Python-based DSL (Domain-Specific Language) that enables code-driven creation of symbol and footprint files, supporting exports to multiple EDA tool formats. Thus, LLMs can transform the task of library generation into a code generation problem based on extracted information. Furthermore, we propose comprehensive error detection and iterative repair mechanisms to further enhance performance.

To the best of our knowledge, this is the first work to address PCB component library generation using MLLMs. The key contributions are summarized as follows:

- We propose a novel multi-agent framework that decomposes the complex process of PCB library generation into specialized subtasks, ensuring efficient and accurate extraction and generation.
- We develop comprehensive solutions to tackle core challenges in PCB component library generation, including customized datasheet preprocessing, domain-specific LLM fine-tuning, code-based generation approaches, and refined verification workflows.
- We introduce PyPclib, a Python-based domain-specific language

† Equal contribution. * Corresponding author.

This work is supported in part by the National Key Research and Development Program of China (No. 2023YFB4402900) and the National Natural Science Foundation of China (No. 92573108, 62304197).

that bridges automated data extraction and standardized library generation for multiple EDA platforms.
- We construct a dataset of over 3,000 samples. Through testing on more than 2,000 samples, we demonstrate that Smart-PCLib achieves both high performance and efficiency.

II. Preliminaries

A. Problem Formulation

In PCB design, symbol and footprint are the foundational elements for constructing electronic circuits. As shown in Fig. 1, symbol represents the abstract functionality of an electronic component in the schematic design, illustrating its pin definitions and connectivity. It forms the basis for connecting components in circuit diagrams. Footprint refers to the physical layout of the component on the PCB, including the precise positions and dimensions of its pads. It directly impacts the manufacturability and assembly of the PCB. The accuracy of both the symbol and footprint is critical to ensuring that the design meets functional requirements and that the circuit can be successfully fabricated and assembled. Based on this context, we formulate the PCB component library generation problem as follows:

Problem 1 (PCB Component Library Generation)**.** *Given a component datasheet, for each unique combination of pin configuration and package type, generate corresponding symbol and footprint files.*

B. LLM related techniques

LLMs serve as the cognitive foundation for AI agents. To enhance agents' task-specific capabilities, several key techniques can be integrated. First, prompt engineering [11] enables agents to better understand task requirements and generate structured outputs by providing carefully crafted instructions. Chain of Thought (CoT) [12] further augments agents' reasoning abilities by encouraging step-by-step problem decomposition and structured thinking, making them more effective at handling complex tasks that require systematic analysis. RAG allows large models to query external knowledge bases as needed and incorporate the retrieved information into the context, enhancing reasoning and generation capabilities [13].

For domain adaptation, Supervised Fine-Tuning (SFT) enables agents to acquire task-specific expertise using annotated data. Low-Rank Adaptation (LoRA) provides an efficient fine-tuning method by introducing trainable rank decomposition matrices, reducing computational costs while preserving the model's general knowledge [14].

C. Multi-Agent Collaboration

LLM-based Agents [15] leverage these enhanced capabilities to perform complex tasks autonomously. They can generate and execute plans independently while utilizing various tools and resources to improve task completion accuracy and efficiency. Through Multi-Agent Systems [16], complex workflows can be decomposed into specialized subtasks, with each agent focusing on its area of expertise. This division of labor, supported by shared workspaces for maintaining and updating information, enables efficient collaboration. By sharing knowledge and maintaining up-to-date context, agents simplify complex tasks and ensure high-quality, coordinated outputs.

III. Approach

A. Overall flow

Fig. 2 illustrates our Smart-PCLib workflow, a multi-agent framework designed to automate the generation of PCB component libraries by dividing the complex process into specialized tasks. The multi-agent system comprises *Brain Agent*, *Vision Agent*, *Coding Agent*, *Verification Agent*, and *Fixer Agent*. For effective collaboration, all agents share a centralized Memory Module, which serves as a repository for storing intermediate results, extracted information, and generated outputs.

- **Brain Agent** acts as the global coordinator of the system. It analyzes the datasheet to extract key information such as component type, package type, pin count, and pin names. Using a CoT reasoning approach, it creates and schedules task plans for other agents, dynamically adjusts plans based on feedback, and iterates to optimize results.
- **Vision Agent** analyzes datasheet images to determine whether they are related to symbols or footprints. For symbol-related images, it extracts pin count and pin name information, while for footprint-related images, it extracts dimensions and spatial relationships. For images with complex layouts or unclear text, vision agent can invoke carefully designed Optical Character Recognition (OCR) tools to improve accuracy.
- **Coding Agent** generates DSL code for creating symbols and footprints based on the information available in Memory. It references a DSL knowledge base of syntax rules and validated examples to improve accuracy. The generated code is then rendered into visual representations (symbol and footprint figures), which are stored for downstream tasks.
- **Verification Agent** validates the generated symbols and footprints by comparing rendered visuals against the extracted datasheet information. It verifies attributes such as pin count, pin names, geometric dimensions. If discrepancies are found, they are passed to Fixer Agent for debugging. If verification is successful, the process concludes without further iterations.
- **Fixer Agent** resolves errors identified by the Verification Agent. It localizes the issues, provides debugging feedback and possible solutions to the Brain Agent. Brain Agent then updates the task plan and initiates iterative fixes. To balance efficiency, the iterative process is limited to a maximum of three cycles.

B. Task Coordination and Decomposition

Datasheets for PCB components are often lengthy, complex, and may contain diverse types of component information, making direct analysis and library generation challenging. To address this, the Brain Agent takes charge of high-level coordination and strategic planning, ensuring that the entire workflow operates efficiently and effectively.

Before analysis, the datasheet is converted into a structured format of Markdown-based text and images, with positional relationships preserved. The Brain Agent then analyzes the content and plans the workflow based on task complexity. It determines task sequences, delegates responsibilities to the appropriate agents, and ensures seamless collaboration between them. For datasheets include multiple packaging types, the Brain Agent uses a divide-and-conquer strategy to decompose the problem into manageable tasks, effectively avoiding confusion and omissions. It also actively checks and maintains the memory module, keeping intermediate results and decisions clear and organized. By centralizing management, the Brain Agent ensures tasks are aligned, executed in order, and free of conflicts.

C. Multimodal information extraction

Although the Brain Agent performs an initial analysis of the datasheet, much critical information resides in images. The Vision Agent identifies and extracts this information by quickly locating images related to symbols and footprints, leveraging surrounding text and contextual clues. It further analyzes these images based on foundational information stored in the memory module.

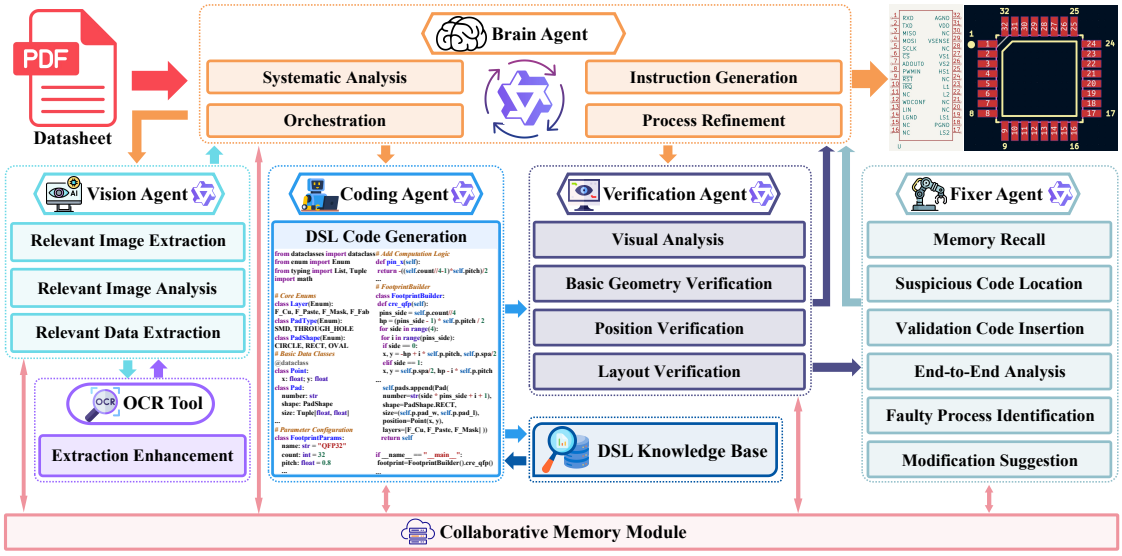


Fig. 2 Overall flow.

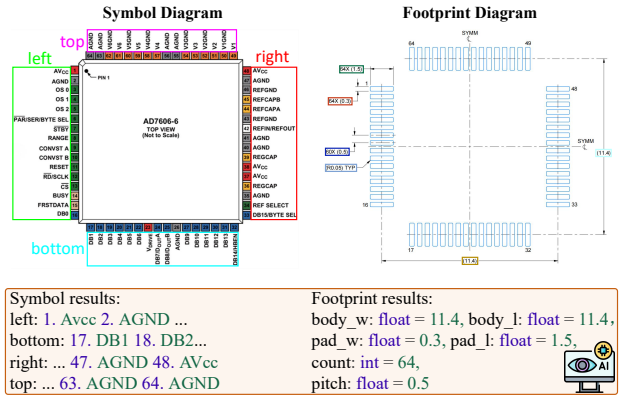


Fig. 3 Analysis results of symbol and footprint diagrams. Colored boxes represent OCR recognition results, for symbol diagram, elements are aggregated into sequences based on their coordinates.

For symbols, the relevant images include schematic diagrams and pin diagrams. From these, the Vision Agent extracts details such as pin names and pad sizes. For footprints, the relevant images include pad arrangement or land pattern diagrams, which usually include size annotations. Accurately interpreting dimensional values along with contextual meanings is critical for understanding the component's layout and specifications. As shown in Fig. 3, the Vision Agent effectively extracts information from a 64-pin QFN package. For symbol diagrams, it provides a complete list of pin numbers and names in a standardized order (left-bottom-right-top). For footprint diagrams, it identifies the number of pads, their distribution, and dimensional values. The Brain Agent cross-verifies the extracted information, recommunicates with the Vision Agent if inconsistencies are found, maintains the accuracy and consistency of the memory module.

One significant challenge is that MLLMs often struggle with details, occasionally hallucinating pin names, pad counts, or dimensional values. To address this, we developed tools based on Optical Character Recognition (OCR) [17] to extract precise text and coordinates from images, aiding the accurate identification of pin names, dimensional values, and pad distributions. For symbol diagrams, pin names are aggregated into sequences and mapped to their relative positions around the IC. For footprint diagrams, the tools extract dimensional values and pad coordinates while avoiding interference from irrelevant elements. The Vision Agent dynamically decides when to invoke these tools for complex images, improving the reliability of results.

D. DSL-based symbol and footprint generation

After the Vision Agent analyzes the images, the Brain Agent integrates the extracted information with the memory module and instructs the Coding Agent to sequentially generate the symbol and footprint files based on this data. In traditional workflow, engineers need to manually draw symbol and footprint files in EDA platforms, which results in different formats of exported files from various EDA platforms. Among these, files exported from PADS and KiCad can be parsed into text formats. Although it is technically feasible to use LLMs to generate textual information from files exported by PADS and KiCad, the lengthy text and lack of inherent logic pose significant challenges to the generation task. To tackle this bottleneck, we have designed a Python-based DSL (Domain-Specific Language) named PyPCLib, which is tailored for generating symbol and footprint files, leveraging the superior code generation capabilities of LLMs.

The proposed DSL encompasses common operations required for symbol and footprint generation, such as drawing shapes, creating pins, setting attributes, and defining pads of specific shapes and sizes. Fig. 4 illustrates three representations of a 176-pin QFP package component: PADS, KiCad, and our PyPCLib. PADS-based representations often exceed 1000 lines, composed of cryptic commands with obscure syntax and extremely poor readability. KiCad-based representations typically consist of several hundred lines, made up of S-expressions. These S-expressions are structured but feature nested parentheses, leading to moderate readability. These traditional representations are both static text, lacking programming capability and extensibility. In contrast, our PyPCLib is typically under 100 lines, showing conciseness and excellent readability. It features complete programming capabilities, supporting logic control, loops, and functions. It also shows high extensibility, enabling easy addition of validation code for purposes such as parameter checks and logic verification. Moreover, this platform-independent DSL code can be

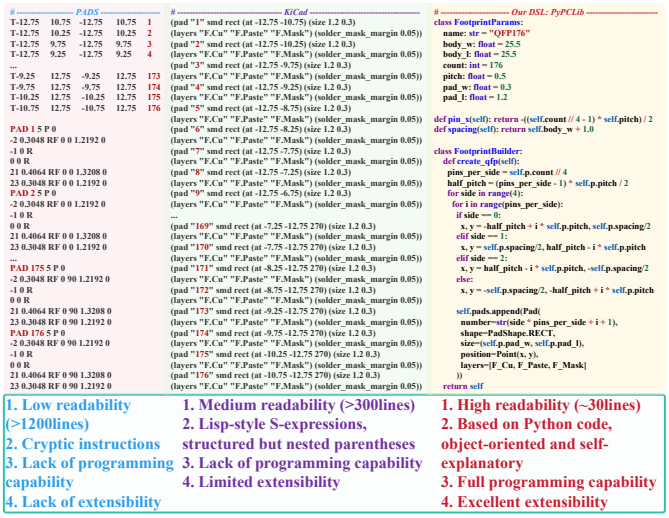


Fig. 4 Comparison of PADS, KiCad, and our PyPCLib.

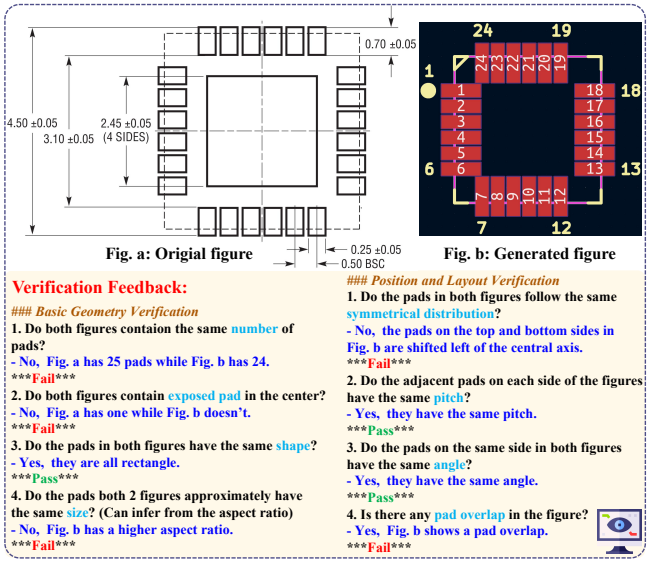


Fig. 5 Verification result of generated footprint.

easily converted into platform-specific formats (e.g., KiCad, PADS, Cadence, Altium) through dedicated interfaces. In summary, highly structured nature, informational conciseness, and the Design-as-Code philosophy make our PyPCLib particularly well-suited for LLMs.

To further enhance the accuracy and adaptability of the generation process, we have constructed a knowledge base containing the DSL's syntax rules and previously verified example codes for specific types of components. This knowledge base is stored in a vector database. Leveraging the Retrieval-Augmented Generation (RAG) technique, the Coding Agent retrieves relevant content from the knowledge base by matching functional requirements and component information. By incorporating relevant content from knowledge base, this approach significantly improves generation accuracy. For instance, when generating a 176-pin QFP component, even if the knowledge base only contains examples for a 16-pin QFP, LLM can leverage its contextual learning capabilities to generalize effectively due to the similarity in principles. This knowledge base can be dynamically updated with special manufacturer requirements or new component types, allowing it to adapt to evolving needs.

Once the symbol and footprint files are generated, the Coding Agent renders them into visual figures for subsequent verification.

E. Verification

Following the generation of symbol and footprint files, verification is crucial. Manual verification is slow, labor-intensive, and prone to errors. To address this, we employ the Verification Agent to systematically validate the generated results. Specifically, the Verification Agent compares the generated visual figures with the original figures and textual information stored in the memory module.

We design multiple verification questions to evaluate the results from different perspectives, ensuring consistency and compliance with industry standards. For symbols, the Verification Agent checks component types, pin counts, and pin names. For footprints, it verifies physical details such as pad dimensions, distribution, and spacings.

As shown in Fig. 5, the Verification Agent performs a side-by-side comparison of the generated and original diagrams, answering relevant verification questions. When inconsistencies are detected, it highlights specific discrepancies. These detailed insights help identify errors in the process and guide corrections. When issues have already been identified in prior iterations, the Verification Agent observes

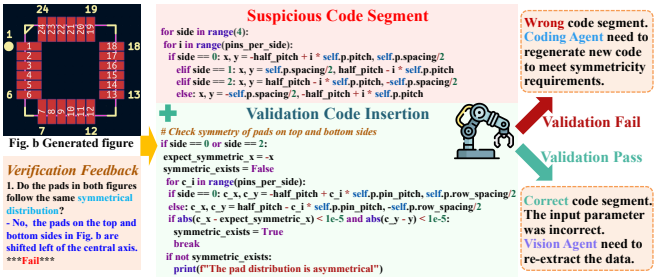


Fig. 6 A debugging example in Fixer Agent.

more rigorously to ensure the errors have been resolved and no new issues have been introduced, guaranteeing the quality and reliability.

F. Error Localization and Correction

When errors are detected by the Verification Agent, the Fixer Agent takes action to resolve them. It identifies the root cause of the issue based on the memory records of various agents, such as whether the Vision Agent failed to detect certain features or the Coding Agent generated wrong code. The Fixer Agent then provides debugging feedback and proposes solutions to the Brain Agent.

Fig. 6 has shown a debugging example in Fixer Agent. First, Fixer Agent needs to perform a semantic analysis of the errors from the verification feedback, and the analysis reveals that the errors originate from the pad distribution. The Fixer Agent then narrows down the inspection scope by slicing out a small suspicious segment from the generated PyPCLib code based on the semantic analysis results. After that, the relevant validation code will be inserted to check the functionality of the suspicious code segment. Next, Fixer Agent performs code validation with the default parameters. If it fails, it indicates that the functionality of the suspicious code segment is erroneous, and the Coding Agent needs to be instructed to regenerate the PyPCLib code. If it passes, it indicates that the code segment is correct, and the error lies in the information extraction process, requiring the Vision Agent to be instructed to re-extract the data. Specifically, for errors in the coding process, the feedback information will include specific code segment and modification suggestions to prevent code tampering caused by LLM hallucinations.

The Brain Agent incorporates the Fixer Agent's feedback into the task plan and initiates iterative corrections. This process is repeated for up to three iterations, or until the Verification Agent confirms that the issues have been resolved. By automating these refinements, the system minimizes manual intervention while ensuring consistent accuracy and high-quality outputs.

IV. Data Preparation and Agent Training

A. Data Collection

We constructed a dataset of 3480 triplets datasheet, symbol, footprint for electronic components, sourced from official manufacturer websites and professional EDA tool libraries to ensure authenticity and practical value. Each triplet includes a manufacturer's datasheet, the corresponding symbol, and footprint, providing complete coverage of the component's design stages.

The dataset spans a wide range of component types (e.g., integrated circuits, resistors, capacitors) and over 30 package types, such as SOIC, QFP, BGA, and DIP. Component complexity ranges from simple passive devices to integrated circuits with over 200 pins. It also includes components from more than 20 major manufacturers, ensuring diversity in design philosophies and documentation styles. Additionally, multilingual datasheets (e.g., English, Chinese, Japanese) enhance its robustness for multilingual processing. With its extensive coverage and diversity, this dataset is a valuable resource for training and evaluation.

B. Data Preprocessing

Datasheets often contain unstructured elements such as text, tables, and images, making them difficult to process or annotate directly. To address this, we used Docling [18] to parse datasheets into Markdown and images, preserving structural relationships like text hierarchy, tables, and the relative positioning of images and text.

To efficiently extract relevant information from lengthy datasheets, we applied BM25 [19], a keyword-based retrieval method. Pages containing positive keywords (e.g., "pin" or "land pattern") were scored higher, while those with negative keywords (e.g., "electrical characteristics") were penalized. Only pages with scores exceeding a threshold were retained. With carefully designed keyword lists, this method effectively identified key pages, allowing the framework and annotation processes to focus solely on relevant content.

Beyond datasheets, we constructed a reverse engineering tool to process the corresponding symbol and footprint files. This tool extracts critical information, such as pin names, pad counts, and dimensional values, while also reconstructing the associated PyPCLib code. By automating the extraction and code reconstruction, this approach avoids starting annotation from scratch. The raw outputs are further refined and validated by experts to ensure consistency with the original files and correctness of the generated code. These refined outputs were later used as ground truth for training the Vision Agent and Coding Agent.

C. Pipeline for Multi-Agent Training

We adopted the Qwen2.5VL model [5] as the foundation for all agents, using the 7B configuration to balance performance and efficiency. While the base architecture was shared, each agent was fine-tuned for its specific task using distinct LoRA adapters and specific prompts. We constructed a training dataset of 1,160 samples, covering most common component types, packages, manufacturers, and language types to ensure broad generalization. The training process followed a progressive strategy, beginning with the development of individual agent capabilities and later focusing on improving their collaboration. Each agent was independently fine-tuned for its specific task using carefully constructed ground truth data derived from manual corrections, preprocessing, or synthetic datasets:

- Brain Agent: Extracts information from datasheets and generates task plans. Manually corrected outputs served as ground truth.
- Vision Agent: Interprets pin and footprint diagrams by combining visual data and extracted information. Expert-validated outputs from preprocessing were used as ground truth.
- Coding Agent: Generates PyPCLib code based on extracted information and visual data. Ground truth consisted of expert-validated PyPCLib code.
- Evaluator Agent: Identifies and explains inconsistencies in generated files by comparing them with datasheets. Training data included both positive samples (correctly generated outputs) and synthetic error cases (e.g., invalid dimensions), each error case was annotated with detailed explainationss.
- Fixer Agent: Corrects errors in the workflow and provides possible solutions, using synthetic datasets containing typical mistakes and their corresponding fixes.

After individual training, the agents' abilities in their specific tasks were effectively improved. However, some unexpected errors may emerge when the agents are integrated to handle complete datasheets. Testing revealed common issues, such as error localization failures by the Fixer Agent, were used to create new training data, and the respective agents were incrementally fine-tuned. This iterative refinement process improved collaboration and enabled the agents to work seamlessly across the entire workflow.

TABLE I Pin Count Distribution of Test Dataset

Group	Difficulty Level	Pin Count Range	Quantity
A	Easy	1–10	1236
B	Medium	11–30	637
C	Hard	31–80	355
D	Extremely Hard	>80	92

V. Experimental Results

A. Experimental Setup

The Smart-PCLib framework and model training were conducted on a Linux machine equipped with 4 NVIDIA A100 GPUs. Visualization was performed using KiCad 8.0.7. The dataset was divided into 1,160 samples for training and 2,320 samples for testing. According to the number of pins, we categorized the test set into different difficulty levels (Group A–D), Table I shows the distribution.

The framework was deployed using vLLM [20], a high-performance serving engine designed for efficient LLM inference. By dynamically loading the corresponding LoRA adapter weights, vLLM enables seamless switching between agent roles. On average, generating files for a single case from the test set took 26.1 seconds. Furthermore, the framework supports parallel processing of multiple datasheets, significantly outperforming human efficiency, which typically requires several minutes to handle a single sample.

To ensure accurate and detailed evaluation, we divide the pipeline into three distinct stages:

- **Stage 1: Information Extraction** The Brain Agent processes textual component information, while the Vision Agent extracts pin/pad counts, pin names, and dimensions from images.
- **Stage 2: Code Generation** The Coding Agent translates the correctly extracted information into PyPCLib code to produce symbol and footprint files.

- **Stage 3: Verification and Iteration** Verification Agent flags discrepancies; Fixer Agent pinpoints the cause and returns targeted fixes to Brain Agent, the entire workflow is then re-executed for refinement.

B. Evaluation Metrics

We propose four metrics for comprehensive evaluation:

1) **PCC** (Pin/Pad Count Correctness) checks whether the number of pins and pads matches the ground truth: 1 if correct, 0 if incorrect. If PCC is 0, all other metrics are directly set to 0.
2) **NMR** (Pin Name Match Rate) measures how many pin names are correctly matched:

$$\text{NMR} = \frac{\text{Number of Correct Pin Names}}{\text{Total Number of Pins}}. \tag{1}$$

3) **DMR** (Dimension Match Rate) evaluates the accuracy of extracted footprint dimensions. A dimension is considered correct only if both its value and meaning are accurate.

$$\text{DMR} = \frac{\text{Number of Correct Dimensions}}{\text{Total Number of Dimensions}}. \tag{2}$$

4) **POR** (Pad Overlap Rate) measures the geometric accuracy of generated footprint files by computing the overlap between generated and ground-truth pad regions. A POR of 1 indicates perfect overlap; 0 means no overlap:

$$\text{POR} = \frac{\text{Area}_{\text{gen}} \cap \text{Area}_{\text{truth}}}{\text{Area}_{\text{gen}} \cup \text{Area}_{\text{truth}}}. \tag{3}$$

Stage 1 employs PCC, NMR, and DMR to measure the correctness of the extracted information. Stage 2, Stage 3, and the end-to-end workflow utilize PCC, NMR, and POR to evaluate the quality of the generated symbol and footprint files. To reflect the framework's ability to handle varying levels of complexity, we report the average metrics across all groups in Table I.

C. Evaluation

Table II summarizes the end-to-end workflow performance across four difficulty groups. The column "wo.Ver&Fix" indicating results without Verification and Iteration. The framework performs exceptionally well in simpler groups, while performance declines as pin count and complexity increase. The drop in PCC is primarily due to missed special structures like exposed pads, while NMR declines stem from challenges in recognizing irregular pin names or pin arrangements in complex images. Similarly, POR decreases as extracting dimensions and generating accurate footprints for complex components becomes harder. With Verification and Iteration, all metrics improved, particularly in complex groups, showcasing the effectiveness of the check-and-retry mechanism.

We also evaluated Information Extraction and Code Generation independently to test the agents' task-specific performance. For comparison, we replaced our agents with the state-of-the-art multimodal language model, GPT-5 [4]. Table III presents the performance comparison on the Information Extraction task. The column "wo.OCR" represents our agents without utilizing OCR tools, while the other methods used them. GPT-5 performed comparably to our agents on simpler tasks but struggled significantly with complex ones, highlighting its limitations in understanding PCB-specific content. For example, GPT-5 often misclassified application diagrams as symbols, leading to incorrect pin counts, and struggled to interpret the meaning of extracted dimension values in complex footprint diagrams. In contrast, our agents, trained on domain-specific data, accurately extracted PCB information. Additionally, OCR tools effectively improved performance; for example, they correctly recognized pin characters and aggregated them into sequences, which reduced LLM hallucinations and improved NMR.

TABLE II Ablation study of our end-to-end generation framework.

G.	PCC		NMR		POR	
	Ours	wo.Ver&Fix	Ours	wo.Ver&Fix	Ours	wo.Ver&Fix
A	**99.8%**	99.4%	**99.0%**	98.4%	**94.7%**	89.2%
B	**99.2%**	97.8%	**97.7%**	95.6%	**89.1%**	82.3%
C	**97.4%**	95.4%	**94.5%**	91.6%	**80.6%**	71.8%
D	**95.5%**	92.1%	**90.8%**	86.1%	**73.8%**	60.9%

TABLE III Comparison of information extraction performance.

G.	PCC			NMR			DMR		
	Ours	GPT5	wo.OCR	Ours	GPT5	wo.OCR	Ours	GPT5	wo.OCR
A	**99.4%**	99.2%	98.9%	**98.4%**	97.8%	95.9%	**97.8%**	84.5%	95.6%
B	**97.8%**	95.1%	94.5%	**95.1%**	92.6%	87.0%	**92.8%**	71.2%	89.7%
C	**95.4%**	89.4%	90.2%	**91.2%**	85.1%	79.3%	**85.3%**	54.2%	82.7%
D	**90.1%**	82.3%	84.7%	**86.7%**	76.4%	72.7%	**76.2%**	32.7%	70.5%

TABLE IV Comparison of code generation performance.

G.	PCC			NMR			POR		
	Ours	GPT5	Ours-Ki	Ours	GPT5	Ours-Ki	Ours	GPT5	Ours-Ki
A	**99.8%**	95.3%	87.4%	**99.8%**	95.1%	85.9%	**97.1%**	83.2%	72.7%
B	**98.7%**	93.2%	81.3%	**98.5%**	89.7%	80.5%	**90.4%**	72.1%	60.2%
C	**98.1%**	89.3%	71.9%	**97.6%**	85.9%	69.6%	**83.2%**	51.6%	43.9%
D	**97.3%**	82.7%	46.2%	**96.5%**	77.4%	42.7%	**71.7%**	28.5%	21.3%

Table IV evaluates the performance comparison on the Code Generation task. All methods received the same, correct component information as input. GPT-5 also generated PyPCLib code to produce symbol and footprint files, while Ours-Ki refers to a model trained to generate KiCad-based representations instead of PyPCLib code. Ours-Ki used the same Qwen model, but trained with KiCad representation rules and (component information, KiCad-based representation) pairs. If generated result failed to execute or parse, all metrics were set to zero. Results show that GPT-5's code generation reliability and quality declined significantly for more complex groups, due to the lack of domain-specific training. Furthermore, even after training, the KiCad-based generation approach performed worse than the code-based approach, mainly because KiCad representations are verbose and lack logical structure. For example, in group D, over half of the KiCad-based outputs failed to parse correctly. Even for parsable results, hallucinations and inconsistencies were frequent in symbol and footprint generation. In contrast, our PyPCLib-based approach, with its explicit logical structure, consistently outperformed KiCad-based representations, particularly on complex components, demonstrating its clear superiority for accurate file generation.

VI. Conclusion

This paper introduced Smart-PCLib, an automated multi-agent framework that overcomes the challenges of traditional PCB component library generation. Our approach employs a series of specialized MLLM-powered agents that systematically deconstruct the generation process. These agents collaborate to perform information extraction from unstructured datasheets and then generate symbol and footprint files utilize a novel Python-based domain-specific language. This method transforms the ambiguous task of file generation into a more robust and manageable code-driven workflow. Experimental results confirm that this approach, enhanced by an automated verification and correction loop, significantly improves the accuracy and efficiency of the generation.

REFERENCES

[1] S. Sangwine, *Electronic components and technology*. CRC press, 2018.

[2] "Snapeda," https://www.snapeda.com/.

[3] "Footprintku," https://www.footprintku.com/.

[4] OpenAI, "GPT-5 System Card," OpenAI, Tech. Rep., Aug. 2025. [Online]. Available: https://openai.com/index/gpt-5-system-card/

[5] S. Bai, K. Chen, X. Liu, J. Wang, W. Ge, S. Song, K. Dang, P. Wang, S. Wang, J. Tang *et al.*, "Qwen2.5-VL technical report," *arXiv preprint arXiv:2502.13923*, 2025.

[6] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[7] G. Peikos, P. Kasela, and G. Pasi, "Leveraging large language models for medical information extraction and query generation," in *2024 IEEE/WIC International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*. IEEE, 2024, pp. 367–372.

[8] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *arXiv preprint arXiv:2307.03172*, 2023.

[9] J. Wei, C. Jia, Q. Chen, H. He, L. Sun, C. He, L. Wu, B. Yu, and C. Tan, "Geoint-r1: Formalizing multimodal geometric reasoning with dynamic auxiliary constructions," *arXiv preprint arXiv:2508.03173*, 2025.

[10] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, 2025.

[11] B. Chen, Z. Zhang, N. Langrené, and S. Zhu, "Unleashing the potential of prompt engineering in large language models: a comprehensive review," *arXiv preprint arXiv:2310.14735*, 2023.

[12] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Proc. NeurIPS*, vol. 35, pp. 24 824–24 837, 2022.

[13] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, H. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, vol. 2, 2023.

[14] E. J. Hu, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen *et al.*, "LoRA: Low-rank adaptation of large language models," in *Proc. ICLR*, vol. 1, no. 2, 2022, p. 3.

[15] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou *et al.*, "The rise and potential of large language model based agents: A survey," *Science China Information Sciences*, vol. 68, no. 2, p. 121101, 2025.

[16] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," in *Proc. IJCAI*, 2024.

[17] R. Team, "Rapid OCR: Ocr toolbox," https://github.com/RapidAI/RapidOCR, 2021.

[18] N. Livathinos, C. Auer, M. Lysak, A. Nassar, M. Dolfi, P. Vagenas, C. B. Ramis, M. Omenetti, K. Dinkla, Y. Kim *et al.*, "Docling: An efficient open-source toolkit for AI-driven document conversion," in *Proc. AAAI*, 2025.

[19] S. Robertson, H. Zaragoza, and M. Taylor, "Simple BM25 extension to multiple weighted fields," in *Proc. CIKM*, 2004, pp. 42–49.

[20] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proc. SOSP*, 2023, pp. 611–626.