DSL-Coder: LLM-based Code Generation for Domain Specific Language

Ruobing Shang*, Shanping Su*, Shaoyi Wang†, Wenliang Chen*, Zhijun Li†‡
* School of Computer Science and Technology, Soochow University, Suzhou, China
† School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China
‡ Corresponding author: Zhijun Li (lizhijun_os@hit.edu.cn)

***Abstract*—Intelligent cockpit systems in modern vehicles require the integration of various functions, resulting in increasingly complex control logic. Domain Specific Language (DSL) is widely adopted to simplify the development process in such scenarios. Recently, Large Language Models (LLMs) have been explored as a promising approach for generating code automatically. However, they still struggle with DSL code generation due to limited prior knowledge and domain-specific syntax variations. In this paper, we introduce CockpitDSL, a dataset for DSL code generation in intelligent cockpit systems, containing 888 annotated user instructions, reference code, and corresponding system states. Based on this dataset, we propose a novel approach that employs Backus-Naur Form (BNF) to define grammar rules, uses a vector database to store the APIs and code snippets, leverages LLMs for code generation, and introduces an evaluation chain for automatically evaluating syntax and grammar. Experimental results demonstrate that the *Compile Success Rate* exceeds 90% and the *Pass@1* score exceeds 75%.**

***Index Terms*—Code Generation, Domain Specific Language, Large Language Model**

I. Introduction

Code generation involves generating code snippets from inputs, such as incomplete code and natural language descriptions [1]. Domain Specific Language (DSL) is tailored for specific application domains, enhancing efficiency and readability within the target domain by constraining its expressive power [2]. Currently, DSL is applied in areas such as configuration files, template engines, query languages, and build scripts, enabling developers to express and solve complex problems in a manner more closely aligned with the problem domain. DSL code generation facilitates the automatic creation of complex workflows, thereby reducing software development time and enhancing flexibility [3]. These advantages have made it a key focus in code generation research.

As shown in Table I, most code generation benchmarks focus on general programming languages such as Python and Java [4]–[6]. Only a limited number of datasets target DSL, and these are often restricted to domains such as query processing [7], hardware design [8], or abstract modeling [9]. Notably, few benchmarks capture the complexity of real-world control scenarios that involve multimodal interactions, hierarchical logic, and hardware-software integration.

The code implementation and dataset supporting this work are publicly available at https://github.com/shangruobing/DSL-Coder.

TABLE I: Code Generation Benchmarks Overview

Benchmark	Scenario	Quantity	Language	Type
BioCoder [4]	Bioinformatics	400	Python & Java	General
TextEditing [7]	Text Editing Command	100	Query Language	DSL
ATIS [7]	Flight Information Inquiry	100	Query Language	DSL
DS-1000 [5]	Data Science	1,000	Python	General
Deep-Bench [6]	Deep Learning	520	Python	General
VerilogEval [8]	Digital Circuit Design	156	Verilog	DSL
CRUST-Bench [10]	Code Transpilation	100	C & Rust	General
DISL [11]	EthereumSmart Contracts	514,506	Solidity	General
SLGPT [9]	Cyber-Physical System	400	Modeling Language	DSL
FloCo [12]	Flowchart-to-Code	11,884	Image & Python	General
Ours	**Intelligent Cockpit**	**888**	**CockpitDSL**	DSL

Intelligent cockpit systems are gradually becoming a central component of modern vehicles, requiring the integration of diverse functionalities such as seat adjustment, device control, and infotainment interaction into a unified and seamless user experience. However, since these systems often involve complex coordination between hardware and software, traditional manual programming approaches struggle to efficiently manage the increasingly complex control logic. Large Language Models (LLMs) offer a promising solution for DSL code generation, but the customized nature of DSLs makes the effective incorporation of their knowledge into the models a key factor in enhancing industrial applications.

In summary, DSL code generation continues to face several persistent challenges. **(1) High-quality DSL code generation datasets are extremely scarce. (2) Most existing approaches are tailored to specific programming languages and lack the flexibility to accommodate the structural and semantic constraints inherent in DSLs.**

To bridge this gap, we introduce a code generation dataset tailored to intelligent cockpit systems. Our dataset, consisting of 888 instances, is designed to reflect realistic in-vehicle control tasks. This contribution provides a valuable resource for evaluating and improving the performance of LLMs on practical DSL generation tasks, and facilitates further research in low-resource code generation. Building on this, we propose a DSL code generation approach that leverages existing code examples and APIs, and employs Backus-Naur Form (BNF) to formally define the syntax of the DSL, thereby guiding LLMs to generate well-structured and domain-compliant code. As illustrated in Figure 1, the approach abstracts low-level

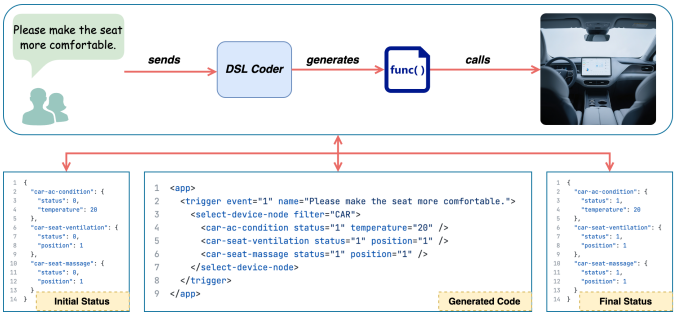

Fig. 1: DSL-Coder Overview: DSL-Coder supports the generation of various types of DSL code. In this study, it is applied to the intelligent cockpit scenario to generate control code for hardware devices such as seats and air conditioning.

hardware control logic into high-level declarative instructions, and utilizes LLMs to generate the corresponding code automatically. This not only enables high flexibility in system configuration, but also ensures clarity and interpretability in system design, making it a key solution to managing the increasing complexity of modern intelligent cockpit systems.

The following are the contributions of this research: **(1) Building a DSL dataset for the Intelligent Cockpit system, including APIs, events, snippets, syntax, and grammar definitions. (2) Design and implement a framework called DSL-Coder to automatically generate DSL code. (3) Design and implement an Evaluation Chain to automatically evaluate the generated code.**

Finally, the advantages of the designed approach are highlighted: **Scalability.** The generated code snippets adapt to input syntax, grammar, and examples, not limited to a specific language. **Accessibility.** The approach learns and infers from a few examples, reducing the manual annotation workload. **Efficiency.** The approach can automatically generate and evaluate code snippets, saving developers time.

II. Related Work

A. Code Generation

Code generation refers to the process of generating software code that meets users' requirements according to users' intent. The more abstract the requirements description is, the more challenging code generation becomes [13]. Therefore, generating code from natural language faces additional challenges, such as the gap between natural languages and programming languages, and the gap between requirements and the actual code [14]. The automatic code generation process is generally divided into three steps, the description of users' requirements, the description of the program search space, and the design of appropriate search ranking techniques [1]. Code generation can not only support the daily work of software developers, but also help users without any programming knowledge to automate repetitive tasks and implement their own new functions [15]. Allamanis et al. [16] propose a system named Haggis, which is capable of automatically mining code idioms from existing codebases based on probabilistic grammar. Jha et al. [17] introduce a novel approach to example-guided program synthesis, which combines oracle-guided learning with constraint-based component synthesis, utilizing Satisfiability Modulo Theories solvers to automatically synthesize loop-free programs.

B. Domain Specific Language

DSL is tailored for specific applications, offering better expressiveness and usability than general-purpose languages. However, creating a DSL is challenging and requires both domain knowledge and expertise in language design [2]. Current code generation methods for DSL, including LLM-based methods, often underperform in practical applications. One of the main challenges is how to make LLMs understand the complex syntax and grammar rules of DSL. In particular, the definition rules of DSL vary across domains and requirements [18]. Meanwhile, there are few examples related to DSL in training corpus, which further increases the difficulty for LLMs to understand DSL syntax and grammar rules. Existing studies rely too heavily on abundant annotated data to train or fine-tune models, which brings problems in terms of cost and timeliness [19]. Data annotation often takes a lot of time and resources, and the annotated data is time-sensitive, which makes the trained or fine-tuned models difficult to adapt to the latest domain requirements. To tackle these challenges and enhance the LLM performance in understanding DSL, researchers have been exploring innovative approaches. Wang et al. [18] leverage BNF to incorporate external knowledge and constraints into LLM during in-context learning.

In summary, this paper specifically focuses on the following two research questions:

RQ1: How can a DSL dataset contribute to addressing the complexity and variability of control logic in intelligent cockpit systems?

RQ2: How can a code generation method be designed to effectively leverage LLMs for generating DSL code?

III. Cockpit-DSL: A DSL Dataset

We introduce a novel dataset for the task of code generation in intelligent cockpit systems, referred to as CockpitDSL. As shown in Table II, this dataset comprises 888 user instructions paired with their corresponding DSL reference codes, along with the initial and final system statuses. In addition, we provide a comprehensive set of supporting resources, including event definitions, service descriptions, and formal grammar specifications, to facilitate accurate understanding and generation of semantically valid code.

The DSL is designed to enhance the expressiveness of task orchestration in intelligent cockpit systems, serving as an intermediate representation within our framework. Unlike general-purpose programming languages, this DSL is not directly executable in standard runtime environments, making conventional test cases inapplicable for validating code correctness. To address this limitation, we represent the execution semantics of each DSL program using a pair of structured system statuses: an initial status and a final status. The

transition between two statuses captures the intended effect of the DSL program on the system environment, enabling indirect evaluation of the functional validity of the generated code, whether it successfully transforms the system from the initial status to the expected final status. The initial status is constructed based on default values specified in the service documentation, ensuring consistency and semantic plausibility. The final status is obtained through the execution of the code by a custom-built DSL interpreter, which translates the DSL program into executable Python code and runs it within an actual Python runtime environment to simulate and retrieve the resulting system status. To better understand the functional coverage of our dataset, we categorize all services into eight functional domains, as shown in Table III.

TABLE II: Summary of the Annotated Dataset.

Category	Item	Quantity	Format
Dataset Instances	Instruction	888	English
	Snippet		CockpitDSL
	Status		JSON
Domain Knowledge	Event	25	JSON
	Service	56	JSON
DSL Specification	Node Definitions	9	JSON
	Grammar	33	BNF

TABLE III: Distribution of Services Across Domains.

Domain	Frequency	Percentage (%)	Cumulative (%)
Comfort Control	545	34.5	34.5
Vehicle System Control	299	18.9	53.5
Phone Integration	185	11.7	65.2
Media & Entertainment	179	11.3	76.6
Navigation & Voice Interaction	112	7.1	83.7
Display & UI	100	6.3	90.0
Reminders & Settings	89	5.6	95.6
Image & Info Processing	69	4.4	100.0

IV. Problem Definition

Given a natural language description D, the goal is to generate a valid code snippet S that accurately fulfills the specified behavior. The process involves multiple stages, including vector search, context retrieval, code generation, grammar validation, and error handling.

- **Input**: A natural language description D of the desired functionality.
- **Output**: A DSL code snippet S that satisfies the behavior described in D.

The problem can be expressed as learning a mapping function f, implemented by the code generation model. The generated code S should compile successfully and produce the correct results.

$$f(D) \rightarrow S \tag{1}$$

A. Vector-Based Retrieval and Search

To enhance code generation, the approach employs vector-based retrieval to search for relevant code snippets, patterns, and templates from a knowledge base or code repository. The natural language description D is first converted into a high-dimensional vector representation V_D. This vector is then used to search for similar code examples or semantic patterns in the search space, represented by a set of vectors $V = \{V_1, V_2, \ldots, V_n\}$.

- **Vector Encoding**: Convert natural language description D into a vector representation V_D using techniques such as word embeddings.
- **Nearest Neighbor Search**: Perform a nearest neighbor search over the set of vectors V to retrieve relevant code snippets and patterns that are semantically similar to D.

The retrieved snippet vectors $S_{candidate} = \{S_1, S_2, \ldots, S_k\}$ serve as references or partial solutions that assist in the generation and refinement of the target code snippet S. The description D, retrieved results $S_{candidate}$, and role information R can be called context C. Such that:

$$C = D + S_{candidate} + R \tag{2}$$

where C is the prompt, which is fed into model f to generate code snippet S.

B. Syntax Validation and Error Handling

After generating the initial code snippet S, a syntax validation step is performed to ensure that the code conforms to the syntax rules of the DSL. If syntax errors are detected, the approach generates an error message E describing the errors in natural language. The approach then attempts to fix the issues in the code snippet S using the error message E, the previous description D and context C.

- **Syntax Validation**: Verify that code snippet S adheres to the syntax of the DSL.
- **Error Description**: If S contains syntax errors, generate an error message E, providing detailed information about the errors, such as the location and type of violation (e.g., missing tags, incorrect expressions).

$$v(S) \rightarrow E \tag{3}$$

C. Code Refinement and Resynthesis

Once an error E is produced, the approach iterates on the code generation process by refining the generated snippet S. This involves adjusting the generated code to address syntax and grammar issues.

- **Refinement**: Modify the initial code snippet S based on the error message E and the retrieved code snippets $\{S_1, S_2, \ldots, S_k\}$ to improve the code's correctness and quality.
- **Resynthesis**: Generate a new or revised code snippet S' that resolves errors and meets the requirements outlined in D and C.

The final output is a refined code snippet S^* that passes syntax and grammar validation, is functionally correct, and is

semantically aligned with the natural language description and context.

$$f(D, C, S') \rightarrow S^* \tag{4}$$

V. DSL-Coder: A DSL Code Generator

In many real-world applications, the user's intent may require generating function calls in multiple programming languages, each targeting different hardware devices. As illustrated in Figure 2, without a DSL, LLM must generate language-specific code (e.g., Java, C++, JavaScript) directly from natural language. This significantly increases both generation complexity and the risk of introducing syntactic and semantic errors. More critically, directly generated code may contain security vulnerabilities that are difficult to detect. As the number of interfaces and target languages grows, the challenges of code generation and debugging become increasingly pronounced. In contrast, adopting a DSL allows the LLM to produce a unified intermediate representation. A dedicated parser and dispatcher can then translate this representation into language-specific function calls. This approach substantially reduces generation complexity and improves system modularity, maintainability, and security.

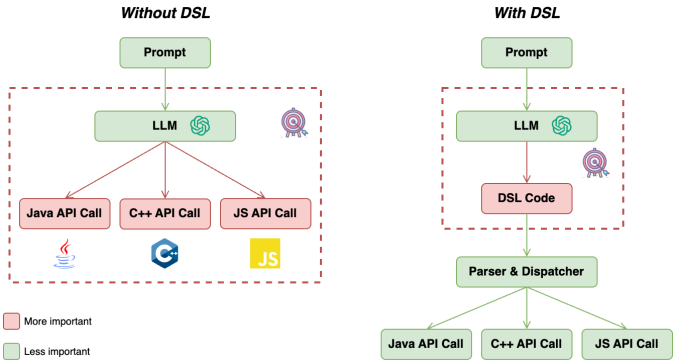

Fig. 2: Comparison of code generation with and without DSL: Without DSL, the LLM generates language-specific function calls, which increases complexity and reduces robustness. With DSL, the LLM produces a unified representation which is parsed and dispatched to appropriate language backends, simplifying generation and improving modularity.

As shown in Figure 3, our approach comprises the following five key modules:

Vector Store handles text vectorization, embedding, and persistent storage. Its tasks include data serialization and deserialization, converting data between text and vector forms.

Retriever retrieves relevant information from the Vector Store, ensuring the LLM receives necessary context. Context includes services, events, code snippets and grammar rules, which are essential for generating accurate code.

Context Manager integrates information from the Retriever to construct a complete prompt, which includes grammar rules, contextual details, chain-of-thought reasoning, and code snippets.

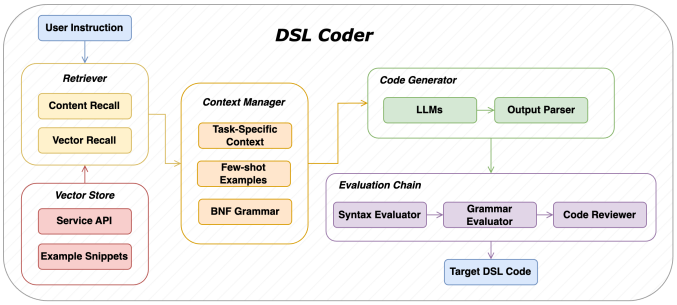

Fig. 3: DSL-Coder: The process starts with receiving the User Instruction. The Retriever accesses the Vector Store to retrieve context relevant to the User Instruction. The Context Manager integrates the retrieved information to build a prompt, which includes grammar, context, and code snippets. The Code Generator uses this prompt to generate the corresponding DSL code. The Evaluation Chain verifies the output by ensuring content, grammar, and overall correctness.

Code Generator utilizes LLMs to generate DSL code based on the prompt. The Output Parser then extracts the DSL code from the model's response.

Evaluation Chain performs content checks, grammar validation, and code reviews. It validates the generated code through syntax and semantic checking. If the code passes the validation of these checkers, the target DSL code is obtained. Otherwise, the Evaluation Chain provides error information about the generated code.

VI. Experiment

A. Setup

Dataset Setting The CockpitDSL dataset contains a total of 888 samples. We randomly select 200 samples as the test set, while the remaining 688 samples are stored in the vector database as candidate entries.

Model Selection. We prioritize an API-based interaction paradigm with LLMs. We conduct the experiments with LLMs including GPT [20], Claude [21], DeepSeek [22], Qwen [23], and LLaMA [24].

Metrics. We evaluate code generation quality from multiple perspectives, focusing on syntactic correctness, functional validity, semantic similarity to reference implementations, and the model's ability to select appropriate APIs. The evaluation includes the following metrics:

- **Compile Success Rate (Compile)**: Evaluates whether the generated code snippet adheres to the syntax rules and can be successfully compiled.
- **Pass@1** [25]: Measures the proportion of generated code whose final system state matches the expected state after execution on the first attempt.
- **Ratcliff/Obershelp Ratio (R/O)** [26]: Computes the character-level similarity between the generated snippet and the reference implementation, based on the longest matching subsequences.

- **BLEU** [27]: Assesses the n-gram overlap between the generated snippet and the reference, reflecting surface-level textual similarity.
- **Accuracy, Precision, Recall, and F1**: Evaluate the LLM's ability to correctly identify and select appropriate APIs from the candidate set.

B. Main Result

We evaluate a series of LLMs with different levels of openness (open-source vs. closed-source). In Table IV, all models achieve a Pass@1 score exceeding 64.50%, with Claude 3.5 Sonnet achieving the best performance among closed-source LLMs (75.00%). Notably, among open-source LLMs, LLaMA3.1-405B achieves the best overall performance (Pass@1 is 75.00%), even surpassing closed-source LLMs such as GPT-4o (68.50%) and Claude 3.5 Haiku (73.00%). Furthermore, when comparing different model sizes (8B, 70B, 405B) within the LLaMA 3.1 family, we observe that even the lightweight 8B model delivers strong performance (65.00%).

TABLE IV: Performance Comparison of LLMs on the Dataset.

Model	Correctness		Similarity		API Selection			
	Compile	Pass@1	R/O	BLEU	Acc.	Prec.	Rec.	F1
GPT-3.5-Turbo	93.00	71.00	66.27	60.66	82.00	86.08	85.62	85.44
GPT-4o	98.00	68.50	66.62	59.34	78.50	81.73	81.90	81.56
GPT-4o-Mini	91.50	69.00	68.59	60.73	**82.50**	87.98	86.74	87.02
Claude-3-Opus	91.50	65.50	67.91	65.50	65.50	79.96	73.38	75.45
Claude-3-5-Haiku	**99.50**	73.00	64.49	58.06	79.00	88.32	85.50	86.35
Claude-3.5-Sonnet	**99.50**	**75.00**	67.34	59.92	82.00	**90.91**	**90.16**	**89.77**
Deepseek-V3	**99.50**	70.50	61.44	58.27	75.50	82.89	80.87	81.39
Qwen-Turbo	89.50	64.50	66.66	58.65	79.50	82.50	82.27	82.05
Qwen-Plus	97.50	69.00	64.86	57.26	78.00	84.10	82.57	82.98
Qwen-Max	98.00	70.00	67.26	59.56	78.00	84.05	84.59	83.81
LLaMA3.1-8B	89.50	65.00	65.24	62.47	66.50	81.07	74.62	76.57
LLaMA3.1-70B	93.00	72.50	69.18	65.12	69.00	86.54	79.77	81.72
LLaMA3.1-405B	94.00	**75.00**	**70.48**	**65.74**	76.00	89.78	83.77	85.45

C. Ablation Study

We perform ablation experiments to analyze the impact of providing API documentation (A), retrieved examples (S), and grammar rules (G) on model performance. Notably, in conditions without S, a fixed example is provided as a default to ensure the model still receives an illustrative reference. In Table V, the results clearly demonstrate that providing contextual information through retrieval significantly enhances the model's ability to understand and perform the task, as evidenced by improvements in multiple metrics, including correctness, similarity and API selection capability.

D. Scaling Law

We evaluate the performance of the Qwen2.5 series with varying parameter scales, aiming to inform model size selection for practical applications. In Table VI, model performance consistently improves with increasing parameter size across multiple metrics. Notably, the Qwen2.5-3B model achieves a Pass@1 score of 54.50%, indicating a reasonable level of usability that opens up possibilities for deployment on resource-constrained devices such as mobile platforms. The Qwen2.5-32B model further improves Pass@1 score to 73.00% and achieves the highest F1 score of 85.81%, demonstrating that medium-sized models can achieve a favorable balance between performance and deployment feasibility.

TABLE V: Ablation Study on the Dataset.

A	S	G	Correctness		Similarity		API Selection			
			Compile	Pass@1	R/O	BLEU	Acc.	Prec.	Rec.	F1
			7.00	2.50	40.88	37.66	3.00	9.29	7.29	7.84
✓			81.50	20.00	46.17	38.83	4.00	31.06	52.95	36.33
✓	✓		84.50	47.50	63.83	54.34	48.50	61.21	70.73	63.66
✓		✓	**97.00**	70.50	57.54	47.83	72.50	84.88	80.59	81.99
✓	✓	✓	93.00	**71.50**	**68.32**	**61.02**	**83.00**	**88.21**	**87.50**	**87.48**

Note: Model is GPT-4o-Mini. A: API, S: Snippet, G: Grammar.

TABLE VI: Scaling Law on the Dataset.

Model	Correctness		Similarity		API Selection			
	Compile	Pass@1	R/O	BLEU	Acc.	Prec.	Rec.	F1
Qwen2.5-0.5B	31.00	21.50	31.46	25.39	27.00	34.85	44.78	37.84
Qwen2.5-1.5B	63.50	47.00	52.82	42.48	58.50	65.22	72.58	67.11
Qwen2.5-3B	77.00	54.50	60.89	54.34	72.50	77.50	76.24	76.48
Qwen2.5-7B	90.00	60.50	63.28	58.00	71.00	78.72	76.24	76.98
Qwen2.5-14B	89.50	65.00	66.04	57.95	78.50	81.48	82.29	81.60
Qwen2.5-32B	**96.00**	**73.00**	63.40	56.34	**81.50**	**86.68**	85.67	**85.81**
Qwen2.5-72B	92.50	69.50	**66.36**	**59.28**	**81.50**	86.25	**85.98**	85.51

E. Error Analysis

We analyze code generation errors across multiple LLMs, categorizing them into Syntax Errors and State Mismatches. Syntax Errors arise from invalid syntax that prevents parsing (e.g., formatting mistakes, fabricated APIs, undefined events, illegal parameters), while State Mismatches occur when syntactically valid code fails to realize the intended state. These error types respectively capture a model's capacity for producing valid DSL code and for meeting user requirements. As shown in Figure 4, Claude 3.5 Haiku and Claude 3.5 Sonnet exhibit fewer Syntax Errors but more State Mismatches, indicating that while these models achieve higher syntactic fidelity, they remain limited in satisfying task-specific requirements.

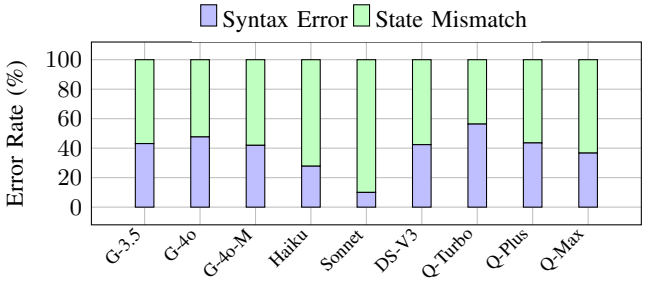

Fig. 4: Error type distribution across models. Model abbreviations: G: GPT, DS: DeepSeek, Q: Qwen.

VII. Discussion

Substitutability of LLMs in Retrieval. The understanding of user's intent and related information retrieval relies on

the embedding ability. The intent needs to be encoded into a vector, and the relevant data is searched by calculating the vector similarity. Fine-tuning the Embedding Model for specific scenarios can improve the accuracy of understanding user's intent. It may be possible to understand the user's intent more accurately, so as to provide contextual information with higher quality for subsequent code generation tasks.

Grammar-based Controllable Generation. Our approach adopts the *"one question, one answer, and one check"* mode when generating the answer. In this mode, grammar checks must wait until the answer generation is finished. If grammar rules can be used, the LLMs are required to generate a part of the text, then the generated content is checked, and if it satisfies the rules, generation continues. Otherwise, the LLMs must adjust the generated content and attempt again until the grammar rules are satisfied before proceeding with the next generation. This controlled generation based on grammar helps to ensure the quality and correctness of the generated text.

VIII. Conclusion

This paper proposes a DSL code generation dataset named CockpitDSL for intelligent cockpit service orchestration. Based on this dataset, this paper designs a DSL code generation method called DSL-Coder. Experiments demonstrate that: (1) the CockpitDSL dataset effectively addresses the complexity and variability of control logic in intelligent cockpit systems; (2) DSL-Coder is capable of effectively generating syntactically correct and semantically meaningful DSL code from natural language instructions.

Acknowledgment

This work is supported by Suzhou Key Core Technologies Breakthrough Project under Grant SYG2024144.

References

[1] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh, *Program Synthesis*, Foundations and Trends® in Programming Languages, 2017.

[2] Marjan Mernik, Jan Heering, and Anthony M Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[3] Rohit Gupta, Sieglinde Kranz, Nikolaus Regnat, Bernhard Rumpe, and Andreas Wortmann, "Towards a systematic engineering of industrial domain-specific languages," in *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*. IEEE, 2021, pp. 49–56.

[4] Xiangru Tang, Bill Qian, Rick Gao, Jiakang Chen, Xinyun Chen, and Mark Gerstein, "Biocoder: A benchmark for bioinformatics code generation with contextual pragmatic knowledge," *openreview*, 2023.

[5] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu, "Ds-1000: A natural and reliable benchmark for data science code generation," in *International Conference on Machine Learning*. PMLR, 2023, pp. 18319–18345.

[6] Alireza Daghighfarsoodeh, Chung-Yu Wang, Hamed Taherkhani, Melika Sepidband, Mohammad Abdollahi, Hadi Hemmati, and Hung Viet Pham, "Deep-bench: Deep learning benchmark dataset for code generation," *arXiv preprint arXiv:2502.18726*, 2025.

[7] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy, "Program synthesis using natural language," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 345–356.

[8] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.

[9] Sohil Lal Shrestha and Christoph Csallner, "Slgpt: Using transfer learning to directly generate simulink model files and find bugs in the simulink toolchain," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, 2021, pp. 260–265.

[10] Anirudh Khatry, Robert Zhang, Jia Pan, Ziteng Wang, Qiaochu Chen, Greg Durrett, and Isil Dillig, "Crust-bench: A comprehensive benchmark for c-to-safe-rust transpilation," *arXiv preprint arXiv:2504.15254*, 2025.

[11] Gabriele Morello, Mojtaba Eshghie, Sofia Bobadilla, and Martin Monperrus, "Disl: Fueling research with a large dataset of solidity smart contracts," *arXiv preprint arXiv:2403.16861*, 2024.

[12] Shreya Shukla, Prajwal Gatti, Yogesh Kumar, Vikash Yadav, and Anand Mishra, "Towards making flowchart images machine interpretable," in *International Conference on Document Analysis and Recognition*. Springer, 2023, pp. 505–521.

[13] Zifan Nan, Hui Guan, and Xipeng Shen, "Hisyn: human learning-inspired natural language programming," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 75–86.

[14] Jiho Shin and Jaechang Nam, "A survey of automatic code generation from natural language," *Journal of Information Processing Systems*, vol. 17, no. 3, pp. 537–555, 2021.

[15] Dominik Sobania, Dirk Schweim, and Franz Rothlauf, "A comprehensive survey on program synthesis with evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 27, no. 1, pp. 82–97, 2022.

[16] Miltiadis Allamanis and Charles Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Nov. 2014, SIGSOFT/FSE'14, ACM.

[17] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 215–224.

[18] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A Saurous, and Yoon Kim, "Grammar prompting for domain-specific language generation with large language models," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[19] Ben Mann, N Ryder, M Subbiah, J Kaplan, P Dhariwal, A Neelakantan, P Shyam, G Sastry, A Askell, S Agarwal, et al., "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, vol. 1, 2020.

[20] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al., "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[21] Anthropic, "Claude 3.5 sonnet model card addendum," 2024.

[22] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al., "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[23] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al., "Qwen2.5 technical report," *arXiv preprint arXiv:2412.15115*, 2024.

[24] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al., "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[25] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang, "Spoc: Search-based pseudocode to code," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[26] John W Ratcliff and DM Metzener, "Gestalt: an introduction to the ratcliff/obershelp pattern matching algorithm," *Dr. Dobbs Journal*, vol. 7, pp. 46, 1988.

[27] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.