2025 International Conference on Intelligent Computing
July 26-29, Ningbo, China
https://www.ic-icc.cn/2025/index.php

Pipeline Method for Domain-specific Language Generation in Low-code Platforms Using Large Language Models

Xin Cui[1][0009-0001-9545-8844], Weixing Zhang[1][0009-0002-1413-6024], Linnan Jiang[1][0009-0007-0443-9917], Aimin Pan[1][0009-0004-9823-1436] and Fei Yang[1]✉[0000-0003-4802-3191]

[1] Zhejiang Lab, Kechuang Ave., 311121 Hangzhou, China
yangf@zhejianglab.org

Abstract. The advancements in language models, particularly Large Language Models (LLMs) have propelled the evolution of front-end low-code platforms, transitioning from the traditional drag-and-drop approach to an automated Domain-Specific Language (DSL) code-based generation process. Within this context, the objective becomes to generate the appropriate DSL from textual descriptions using large language models. Nonetheless, due to the limitation of DSL data, challenges persist in training or fine-tuning LLMs for some DSL generation tasks such as the front-end low code platform. This study proposes a novel pipeline approach for DSL generation, taking advantage of the potential of prompt engineering. The methodology utilizes Named Entity Recognition (NER), a DSL knowledge vector database, and LLMs. The experiments demonstrated significant improvements in the quality of DSL generation while reducing token and time costs.

Keywords: DSL generation, LLMs, Vector database, In-context learning, Prompt engineering

1 Introduction

Artificial intelligence is revolutionizing production processes across various domains by offering automated assistance. A typical example is that automatic code generation gradually transforms how programmers approach their work, enabling them to focus on the core logic of their programs while freeing them from repetitive and boilerplate code details. Front-end low-code platform [1] stands out as a crucial application of automatic code generation. It facilitates the generation of DSL from textual descriptions, consequently allowing users to obtain front-end pages.

In general, automatic code generation is implemented by natural language models. The models take user input, typically brief kernel descriptions encompassing language, logic, and function aspects of the target code, and generate the corresponding code output by comprehending the key information. With the rapid development of LLMs [2], in-context learning-based code generation has become increasingly prevalent, showcasing remarkable reliability for programming languages.

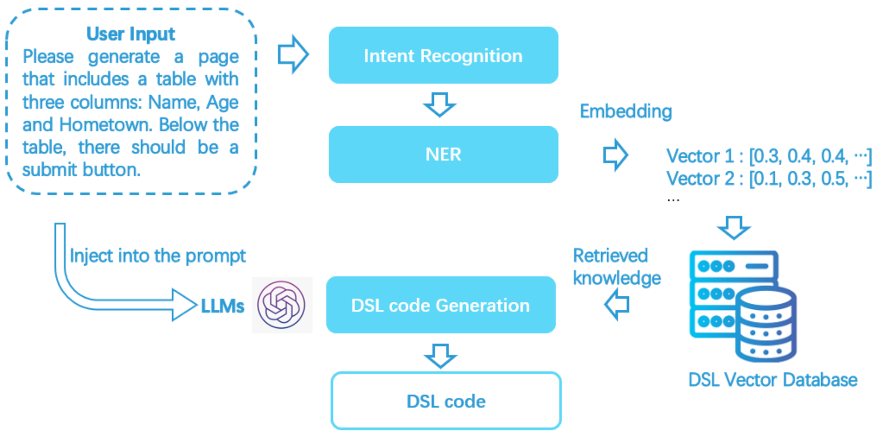

User Input
Please generate a page that includes a table with three columns: Name, Age and Hometown. Below the table, there should be a submit button.
Intent Recognition
NER
Embedding
Vector 1 : [0.3, 0.4, 0.4, ⋯]
Vector 2 : [0.1, 0.3, 0.5, ⋯]
...
DSL Vector Database
Retrieved knowledge
Inject into the prompt
LLMs
DSL code Generation
DSL code

Fig. 1. Pipeline Overview.

Nevertheless, it is still difficult for existing LLMs to produce satisfactory DSL [3] if only provided with textual descriptions without the procedure of training or fine-tuning on the DSL data. Fortunately, LLMs are distinguished largely from normal language models due to the emergent ability and Chain-of-Thought [4]. This introduces the construction of DSL prompts [3], [5] as a novel approach that eliminates the training process. Consequently, LLMs can perform a wide range of tasks proficiently with minimal examples.

In this work, we concentrate on generating DSL of front-end low-code platforms, where the DSL is subsequently compiled into visual pages. To achieve this objective, we aim to produce DSL of superior quality utilizing refined prompts and leveraging the capabilities of LLMs. The main challenge is that the quality of the generated DSL is largely determined by the provided prompts. It is imperative to tailor the prompts precisely to the user's intentions and incorporate the most relevant DSL knowledge.

To address this issue, we propose a pipeline-based solution that integrates the Intent Recognition (IR), Named Entity Recognition (NER), DSL vector database, and DSL generation modules, as depicted in **Fig. 1**. Within the pipeline, we identify the user's intentions from the textual description, match these intentions with relevant front-end DSL knowledge to construct a prompt, and finally, concatenate the prompt with the user's description, sending it to LLM to obtain the generated DSL. With GPT-3.5 [6] and Qwen-72B [7] as the LLMs in the pipeline, we achieve average page-generation scores of 95.92 and 94.5, respectively, to generate the DSL of the low-code front-end platform.

In summary, this study has made notable progress in the following areas. Firstly, we propose an innovative pipeline method for DSL generation leveraging LLMs, which integrates key components such as IR, NER, and vector databases. Furthermore, we validate the efficiency and accuracy of this approach, achieving a high score in DSL generation with reduced token usage.

2 Related Work

2.1 Code Generation

In recent years, numerous research efforts have focused on code-generation tasks employing deep-learning methods. TranX [8] studies a sequence-to-sequence methodology, leveraging the Long Short-Term Memory (LSTM) backbone and an abstract syntax description language. CodeT5 [9] is a code generation version of the Google T5 model, which uses an encoder-decoder transformer structure and applies pre-training fine-tuning technique. However, these works have limitations in the quality of DSL generation due to the lack of high-quality training data. For our DSL generation task, it is challenging to obtain such a training dataset.

With the rapid increase in training data and parameter scale, LLMs have demonstrated remarkable proficiency both in zero-shot and few-shot generation across various code types, including program languages and domain-specific languages [5], [10], [11]. The emergent and contextual understanding abilities brought by LLMs enable effective few-shot in-context learning for DSL generation without the necessity of a large training dataset. LLMs outperform traditional deep-learning models in this context.

2.2 In-Context Learning (ICL)

ICL which relies on natural language descriptions instead of neural network training, has demonstrated remarkable zero-shot and few-shot performance across various natural language processing (NLP) tasks. The progress is attributed to the exceptional emerging abilities of LLMs [12]. The ongoing research in ICL focuses on improving inference performance [13] and expanding example coverage structures [14]. This implies the potential to achieve satisfactory answers from LLMs with minimal examples. Despite the remarkable strides that LLMs have made, it remains challenging to construct adequate DSL prompts for topics the LLMs have not been trained for. A method proposed by [5] employs a BNF syntax, which enables LLMs to generate high-quality responses. Furthermore, vector database [15] or the knowledge base has been demonstrated to serve as reliable external references for DSL grammar, facilitating the generation of high-quality prompts.

In this work, we adopt a pipeline approach that integrates the NER technique, vector database, and LLMs. This strategy enables the construction of effective prompts with minimal redundancy, resulting in high accuracy for DSL generation.

3 Pipeline Methodology

The Overview of the pipeline is illustrated in **Fig. 1**, comprising four ordered modules: IR, NER, DSL vector database, and DSL Generation.

3.1 IR

IR serves as the foundational module for each conversation task, guiding the overall workflow, filtering out inputs irrelevant to front-end page construction, and significantly reducing computational costs and susceptibility to potential attacks. It utilizes the ICL capability of LLMs to construct prompts and classify user's natural language input into three predefined request types: creating a new interface, modifying the interface, and irrelevant requests. If the input is determined to be entirely irrelevant, the system prompts the user to restructure the input into a more contextually relevant format for front-end interface construction. Otherwise, the system processes the input and proceeds with subsequent operations. Let R denote the initial user's request and $\boldsymbol{IR}$ denote the process of IR, then the request category C can be denoted as (1):

$$\mathsf{C} = \boldsymbol{IR}(R),\ \mathsf{C} \in \{new, modify, irrelevant\}\,. \tag{1}$$

3.2 NER

After being verified by the IR module, the pipeline steps into the NER module, where the NER technique is utilized to parse and extract the entities related to the front-end page components. To formulate, given the user's initial textual request R, the NER process is to recognize and then extract a set of predefined DSL components entities ε as shown in (2)

$$\varepsilon = \boldsymbol{NER}(R), \varepsilon \subseteq \{e_1, \dots, e_n\} \tag{2}$$

3.3 DSL Vector Database

Throughout our pipeline, we incorporate a DSL vector database as the external knowledge reference of front-end low-platform DSL. This serves as a resource for high-quality prompt information, ensuring the reliability and quality of DSL generation results. Generally, DSL vector databases facilitate efficient vector retrieval and leverage distance functions to rank the retrieved results by similarity, making them a crucial component for providing external knowledge. As depicted in **Fig. 2**, we present an overview of the DSL vector database, which comprises two main components: vector database construction and data retrieval.

In the construction of the vector database, we establish knowledge vectors for all predefined components entities within the DSL. Each vector is represented by a comprehensive set of DSL descriptors, encompassing the entity's name and textual explanations. The precision of retrieved DSL knowledge from the vector database significantly impacts the accuracy of LLMs in generating DSL. Effective text embedding models play a pivotal role in augmenting retrieval accuracy. This study introduces text-embedding-ada-002 [16], which outperforms previous embedding models in text search and sentence similarity tasks. In (3), we designate text-embedding-ada-002 as Em_model, where the entity is denoted as e_i and the explanation text for the entity as e_{i-exp}. Subsequently, the entity and the explanation text are concatenated and the embedding model processes each concatenated pair to generate a 512-dimensional vector:

$$\overrightarrow{X_i} = Em\_model(\text{concat}(e_i,\ e_{i-exp}))\ , \tag{3}$$

where each entry within the vector database corresponds to a front-end component, comprising the embedding vector, the associated component name, textual explanations, and succinct examples of DSL. We construct the index based on the embedding vector to expedite similarity retrieval.

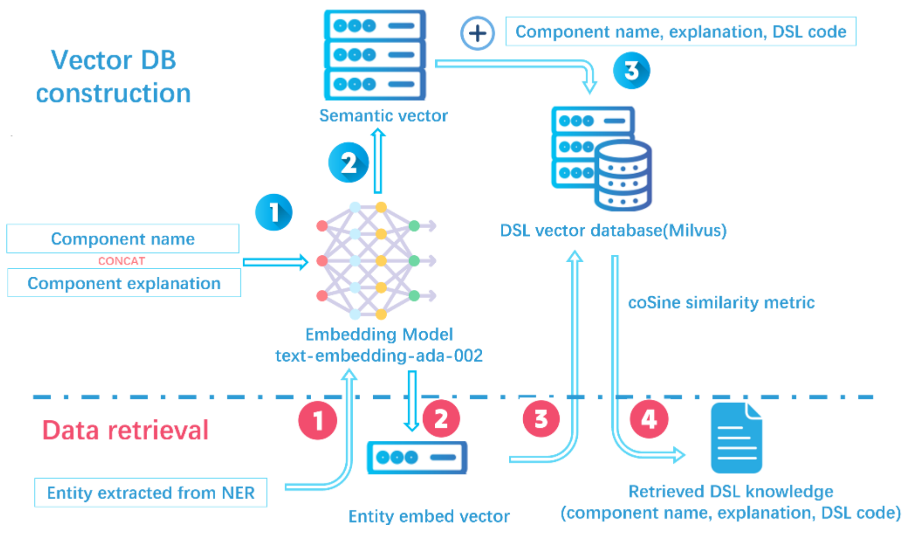

Vector DB construction
Component name, explanation, DSL code
Semantic vector
DSL vector database(Milvus)
Component name
CONCAT
Component explanation
Embedding Model text-embedding-ada-002
coSine similarity metric
Data retrieval
Entity extracted from NER
Entity embed vector
Retrieved DSL knowledge (component name, explanation, DSL code)

Fig. 2. DSL Vector database Overview. The construction phase consists of three key steps. First, we organize the component entity name and the corresponding explanation for each front-end page component, which will serve as input to the embedding model. Second, we use the embedding model to generate semantic vector. Finally, we create a collection in the Milvus [15], [19] vector database, storing the embedded vectors along with their associated component names, explanations, and DSL. Once the DSL vector database is established, the data retrieval phase is carried out in four steps. Initially, we utilize the same embedding model employed during the database construction phase to embed the entities extracted from the NER module. Subsequently, in steps 3 and 4, we leverage the entity embedding vector to retrieve the item with the most similar embedding vector. This retrieved knowledge is then utilized as a prompt to feed into LLMs for generating DSL.

In the querying process, we initially extract all entities from the user's request R using the NER module (2). As delineated in (4), for each entity $e_i \epsilon$ the set of extracted entities, we employ the same embedding model utilized during the construction of the vector database to transform the entity into a vector $\overrightarrow{e_i}$. Moreover, the choice of similarity algorithms employed to retrieve the most similar item from the vector database, such as the Inner Product function, Euclidean distance function, or Cosine distance function, is also a critical factor that impacts retrieval accuracy. Drawing upon previous studies [17], [18], we opt for cosine similarity due to its demonstrated effectiveness in comparing transformer embeddings. This metric offers several advantages, including scale invariance, robustness to varying text lengths, and suitability for high-dimensional spaces, making it a widely adopted choice in NLP applications. We use $cosSim$ to denote the cosine similarity function and $\overrightarrow{X_j}$ to represent each embedding vector in the database, calculating the similarity based on this metric to retrieve the most relevant item.

$$cosSim(\overrightarrow{e_i}, \overrightarrow{X_j}) = \frac{\overrightarrow{e_i} \cdot \overrightarrow{X_j}}{\|\overrightarrow{e_i}\|, \|\overrightarrow{X_j}\|}, \quad where \;\; \overrightarrow{e_i} = Em\_model(e_i) \quad e_i \in \varepsilon \tag{4}$$

3.4 DSL Generation

Given the vector database $\boldsymbol{VDB}$, each embedding vector $\overrightarrow{X_j}$ in $\boldsymbol{VDB}$ and an extracted entity embedding vector $\overrightarrow{e_i}$, the prompt construction aims to obtain the most similar DSL knowledge from the vector database. The retrieved knowledge content can be denoted as (5):

$$K(\overrightarrow{e_i}) = argmax_{\overrightarrow{X_j} \in \boldsymbol{VDB}} \left(cosSim(\overrightarrow{e_i}, \overrightarrow{X_j}) \right) \tag{5}$$

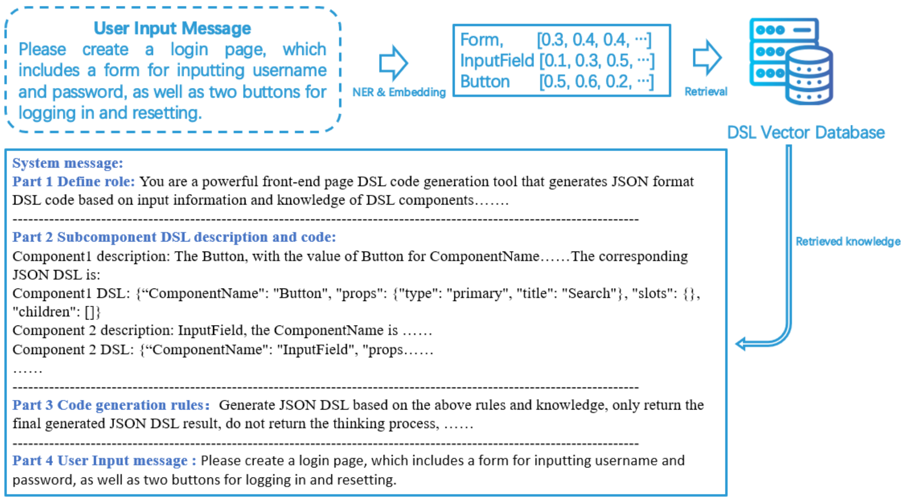
Hence, for each entity embedding vector extracted from the user's textual request, we select the DSL knowledge vector from the DSL vector database that exhibits the highest similarity with it for prompt construction. In (6), R denote the initial user's textual request, $K(\overrightarrow{e_i})$ be the DSL knowledge content (entity, explanation, and corresponding DSL) retrieved from the vector database. we directly concatenate R and $K(\overrightarrow{e_i})$ to form the prompt, which is represented by P. Let $\boldsymbol{LLM}$ denote the mapping function of the LLM, the generated DSL can be presented as $\boldsymbol{Code}$:

$$\boldsymbol{Code} = \boldsymbol{LLM}(P), \quad where \;\; P = concat(R, K(\overrightarrow{e_1}), \ldots K(\overrightarrow{e_i}) \ldots) \quad e_i \in \varepsilon \tag{6}$$

Fig. 3 illustrates a concrete example of the prompt generation process. Initially, user input is received and preprocessed by the NER module to extract entities from the input message. Subsequently, the most relevant DSL knowledge related to these entities' semantic embeddings is retrieved from the DSL vector database. This retrieved DSL knowledge, referred to as Part 2, is then combined with the predefined roles of the LLM system (Part 1) and DSL generation rules (Part 3) to form a well-organized prompt. Finally, this completed prompt is inputted into the LLMs for DSL generation.

3.5 Comprehensive Pipeline

We conclude the comprehensive pipeline process in **Algorithm 1**, which corresponds to the workflow in **Fig. 1**.


User Input Message
Please create a login page, which includes a form for inputting username and password, as well as two buttons for logging in and resetting.
NER & Embedding
Form, [0.3, 0.4, 0.4, ⋯]
InputField [0.1, 0.3, 0.5, ⋯]
Button [0.5, 0.6, 0.2, ⋯]
Retrieval
DSL Vector Database
Retrieved knowledge

System message:
Part 1 Define role: You are a powerful front-end page DSL code generation tool that generates JSON format DSL code based on input information and knowledge of DSL components.……

Part 2 Subcomponent DSL description and code:
Component1 description: The Button, with the value of Button for ComponentName……The corresponding JSON DSL is:
Component1 DSL: {"ComponentName": "Button", "props": {"type": "primary", "title": "Search"}, "slots": {}, "children": []}
Component 2 description: InputField, the ComponentName is ……
Component 2 DSL: {"ComponentName": "InputField", "props……
……

Part 3 Code generation rules: Generate JSON DSL based on the above rules and knowledge, only return the final generated JSON DSL result, do not return the thinking process, ……

Part 4 User Input message : Please create a login page, which includes a form for inputting username and password, as well as two buttons for logging in and resetting.

Fig. 3. Prompt Example for DSL Generation

Algorithm 1 DSL generation

```
Input: Textual Request $R$
C $\leftarrow \boldsymbol{IR}(R)$
if C is irrelevant then
        Inform the user to resubmit the request
else
        $\varepsilon \leftarrow \boldsymbol{NER}(R)$
        P $\leftarrow R$
        for all $e_i \epsilon \varepsilon$ do
                $\vec{e_i} = Em\_model(e_i) \quad e_i \ \epsilon \ \varepsilon$
                $K(\vec{e_i}) = argmax_{\vec{X_j}\ \epsilon\ \boldsymbol{VDB}}\left(cosSim(\vec{e_i}, \vec{X_j})\right)$
                $P \leftarrow concat(P, K(\vec{e_i}))$
        end for
        $\boldsymbol{DSL} \leftarrow \boldsymbol{LLM}(P)$
end if
Output: DSL
```

4 Experiments

We explore three primary issues in the experiments. Firstly, the effectiveness of the IR module is verified in Section 4.2. Secondly, the impact of the NER module that extracts front-end page entities is investigated in Section 4.3. Finally, a comprehensive performance of the prompt construction and DSL generation is illustrated in Section 4.4.

4.1 Experiments Settings

To evaluate the efficacy of the proposed pipeline, we establish a customized testing dataset, with examples shown in **Table 1**, tailored for each module in the pipeline. The absence of a dataset for front-end low-code platform DSL is the reason behind this. Moreover, we apply two LLMs, namely GPT-3.5 and Qwen-72B, for their exceptional generation capacities.

Table 1. Samples of the validation dataset.

We have constructed a front-end low-code platform DSL generation validation dataset, consisting of 100 natural language user-input requirements. The dataset includes 50 requests for front-end page creation, 20 requests for page modifications, and 30 irrelevant requests. For illustration purposes, we present only 9 sample items in this table. The Intent column indicates the type of request: 0 denotes page creation, 1 represents page modification, and 2 signifies an irrelevant request. In the Entities column, the ground truth entities extracted from the user requirements are provided. No entity extraction is performed for irrelevant requests.

ID	User requirements	Intent	Entities
1	I want to generate a login page that includes an account input box, a password input box, a login button, and a cancel button.	0	Form, Input-Field, Button
2	Build the registration page, including input fields for username, password, and email.	0	Form, Input-Field
3	Create a doctor display page for a medical system that showcases the doctor's photo, medical experience, and honors, and presents some of their academic achievements in a table.	0	Form, Input-Field, Button
4	Add an avatar upload component on the personal information page, allowing users to upload their own avatars and enhance personalization customization.	1	Form, Input-Field, Button
5	Update the style of the login page by removing the "Cancel" button and only keeping the "Login" button, simplifying the page and improving user efficiency.	1	Button
6	Add a date picker component on the form page to allow users to easily select a date and improve data accuracy.	1	Date-Field,Form
7	Plan and execute the marketing promotion for the company, increase the brand exposure and sales, and improve the market share and customer satisfaction of the company.	2	——
8	Learn how to design a data visualization page so that users can intuitively understand trends and changes in the data.	2	——
9	Great, web design is very useful in the workplace.	2	——

IR: To validate the accuracy of the IR module, we collected a total of 100 natural language user-input requirements, comprising 50 requests for front-end page creation, 20 requests for page modifications, and 3 irrelevant requests (with examples shown in **Table 1**). We utilized GPT-3.5 and Qwen-72B as the classifiers for this validation process.
Front-end Page Entity Extraction: We collected 50-page generation requests from front-end product managers and annotated 169 entities related to front-end pages. We used the LLM ICL method (GPT-3.5 and Qwen-72B) on this data set to perform the NER task, delivering promising results.
DSL Generation: When evaluating the effectiveness of the front-end low-code DSL generated, we use the dataset described in **Table 1**. This experiment comprises three distinct settings. Setting 1 serves as our baseline method, where only LLMs are used to generate the DSL without the involvement of the NER module or the DSL vector database. In this scenario, all knowledge of the front-end components, including textual explanations and DSL examples, are concatenated to form the prompt for LLMs. Setting 2 introduces the support of the DSL vector database compared to Setting 1, where the user's textual request is directly embedded without undergoing the NER process, and the embedding is matched in the vector database. We then concatenate the knowledge retrieved with the prompt provided to the LLMs. Setting 3 represents the pipeline approach proposed in the paper, which utilizes both the NER module and DSL vector database, following the process outlined in Algorithm 1. In general, we refer to the baseline Setting 1 as *S1, Setting 2 Setting 3 as S2 and S3 respectively, as indicated in **Table 2**.

Table 2. Front-end DSL generation results

Settings	Variance	Model	Token cost	Time cost(s)	Page generation score
*S1	LLM + raw knowledge	GPT-3.5	4902.57	9.95	77.55
S2	LLM + Vector DB	GPT-3.5	1067.82	**6.81**	87.24
S3	Pipeline method (LLM + NER + Vector DB)	GPT-3.5	**924.84**	7.46	**95.92**
*S1	LLM + raw knowledge	Qwen-72B	4902.57	61.25	81.00
S2	LLM + Vector DB	Qwen-72B	1067.82	51.96	84.50
S3	Pipeline method (LLM + NER + Vector DB)	Qwen-72B	**911.24**	**51.63**	**94.50**

4.2 Accuracy of IR

The detailed results of the IR exploration are provided in **Table 3**. The Qwen-72B model achieves a prediction accuracy of 94%, while the GPT-3.5 model attains a prediction accuracy of 99%. In particular, the GPT-3.5 model exhibits exceptional accuracy, with only one instance misclassified as a new creation type instead of a modification type. These findings underscore the module's proficiency in efficiently screening out extraneous user requests and precisely identifying requests for creating or modifying front-end pages. This outcome ensures the efficacy and robustness of subsequent stages in the pipeline.

Table 3. Intent classification

		Predict result		
		New class	Modify class	Other class
New class	GPT-3.5	50	0	0
New class	Qwen-72B	50	0	0
Modify class	GPT-3.5	1	19	0
Modify class	Qwen-72B	4	16	0
Other class	GPT-3.5	0	0	30
Other class	Qwen-72B	1	1	28

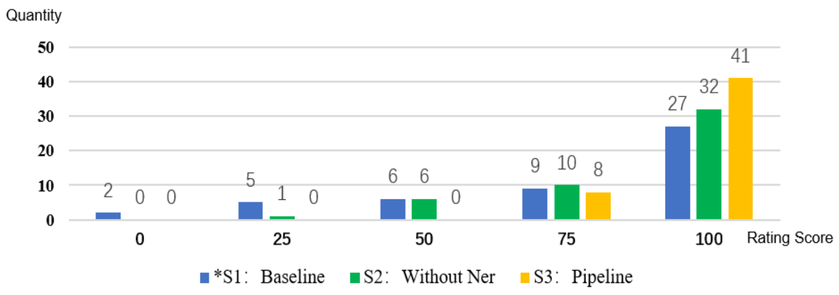
4.3 Impact of NER

For the NER module, the Qwen-72B model achieves an entity extraction precision of 88.00%, with a recall rate of 90.59% and an F1 score of 89.28%. The GPT-3.5 model achieves an impressive entity extraction precision of 92.82%, with a recall rate of 98.82%, and an F1 score of 95.73%. These results demonstrate excellent performance by both LLMs on the NER task, as indicated by their high F1 scores. A higher recall rate ensures the extraction of more components from natural language requirements, facilitating the retrieval of relevant component knowledge from the vector database. Moreover, high precision ensures that the knowledge retrieved is concise and relevant, thus enhancing the quality of generated DSL and reducing token costs.

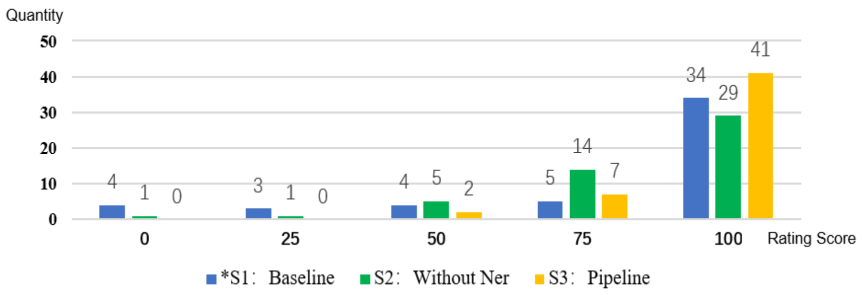
4.4 Efficacy of Front-end low-code platform DSL Generation

We investigate the token cost, time expenditure, and page generation quality across three distinct experimental setups: *S1, S2, and S3. The page generation quality is evaluated on a scale from 0 to 100, with intervals of 25 points, by a team comprising 5 front-end engineers and 5 product managers. They assess the front-end pages generated by our proposed pipeline and assign scores based on the degree of user requirement fulfillment. A score of 0 signifies complete non-compliance with user needs, while scores of 25, 50, 75, and 100 correspond to minimal, partial, substantial, and complete fulfillment, respectively. The distribution of scores for the three experimental settings is depicted in **Fig. 4**, it is apparent that the setting S3 pipeline method, employing both Qwen-72b and GPT-3.5, yields 41 out of 50 results that entirely meet user requirements. Notably, setting S3 exhibits fewer outputs in the low-score range ($\leq$50), distinguishing it from settings *S1 and S2, which produce a considerable number of low-score outputs.

Further insights into performance are presented in **Table 2**, revealing that both GPT-3.5 and Qwen-72B achieve the highest-rated pages under the setting S3 pipeline method, with average scores of 95.92 and 94.5, respectively. Moreover, S3 demonstrates the lowest average token consumption and the shortest API response time among the three settings. Conversely, setting S1, which excludes both the NER module and the vector database, exhibits significantly higher LLM token consumption, prolonged API response time, and diminished generation efficacy, with average scores of 77.55 for GPT-3.5 and 81.00 for Qwen-72B. Incorporating the vector database into the baseline (as in setting S2) reduces both time and token costs, while significantly improving the page generation scores. This comparison underscores the importance of the vector database. Furthermore, comparing the performance of settings S2 and S3 across the two LLMs, it is evident from **Table 2** that the NER process contributes to producing higher-quality pages while reducing token and time costs.



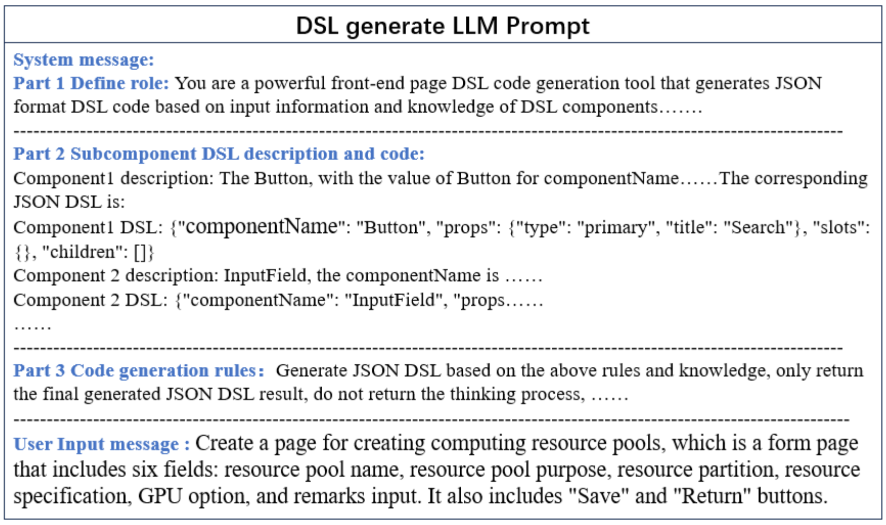
(a) GPT-3.5 generation results distribution



(b) Qwen-72B generation results distribution

Fig. 4. Rating distribution for DSL generation.

In summary, the experiments demonstrate that the vector database and NER module, integral components of our pipeline approach, significantly advance front-end low-code platform DSL generation. Nevertheless, it is noteworthy that Qwen-72B exhibits considerably higher time consumption compared to GPT-3.5, which attributed to differences in API response times between the two models. **Fig. 5** depicts an exemplary case of the structured prompt and the resulting DSL generated by our proposed pipeline. In **Fig. 5**(a), this prompt comprises three main components: the system role, pertinent DSL details, and DSL generation directives, appended with the original user input. Additionally, the example showcases the output DSL, subsequently utilized in rendering the visual page displayed in **Fig. 5** (b) through the front-end low-code platform. Notably, the generated page fully satisfies the user's input requirement.



DSL generate LLM Prompt

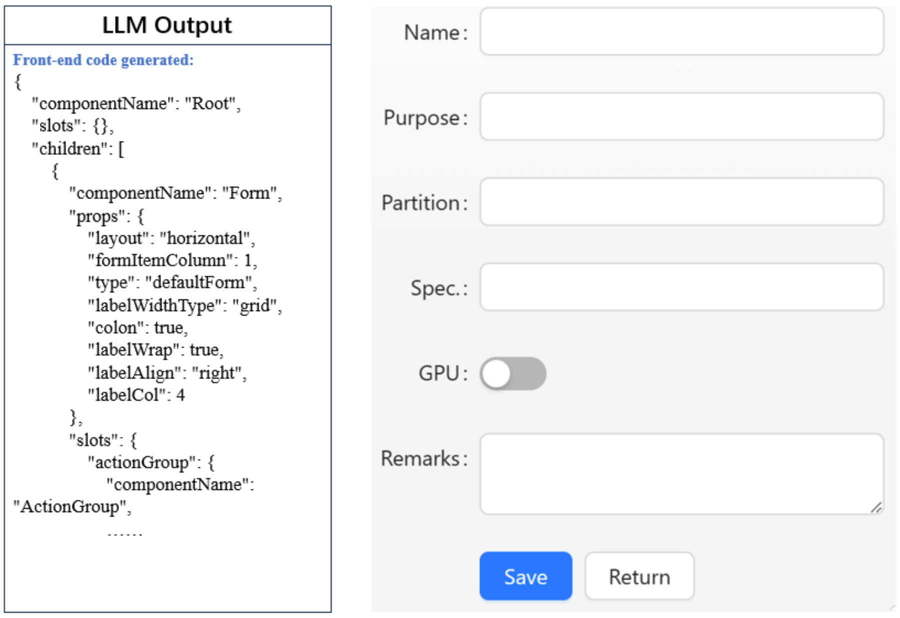
System message:
Part 1 Define role: You are a powerful front-end page DSL code generation tool that generates JSON format DSL code based on input information and knowledge of DSL components……

Part 2 Subcomponent DSL description and code:
Component1 description: The Button, with the value of Button for componentName……The corresponding JSON DSL is:
Component1 DSL: {"componentName": "Button", "props": {"type": "primary", "title": "Search"}, "slots": {}, "children": []}
Component 2 description: InputField, the componentName is ……
Component 2 DSL: {"componentName": "InputField", "props……
……

Part 3 Code generation rules: Generate JSON DSL based on the above rules and knowledge, only return the final generated JSON DSL result, do not return the thinking process, ……

User Input message : Create a page for creating computing resource pools, which is a form page that includes six fields: resource pool name, resource pool purpose, resource partition, resource specification, GPU option, and remarks input. It also includes "Save" and "Return" buttons.

(a) Prompt details



LLM Output

Front-end code generated:

```
{
  "componentName": "Root",
  "slots": {},
  "children": [
    {
      "componentName": "Form",
      "props": {
        "layout": "horizontal",
        "formItemColumn": 1,
        "type": "defaultForm",
        "labelWidthType": "grid",
        "colon": true,
        "labelWrap": true,
        "labelAlign": "right",
        "labelCol": 4
      },
      "slots": {
        "actionGroup": {
          "componentName":
"ActionGroup",
          ……
```

(b) Generated DSL and webpage

Fig. 5. DSL Generation Example

5 Discussion

Our proposed pipeline significantly enhances the capability of LLMs in front-end low-code platform DSL generation. Key observations and remarks regarding the pipeline are summarized below:

- The incorporation of the IR module enhances the precision of LLMs in understanding human instructions by filtering out irrelevant information.
- The NER module and the vector database aid in the concise construction of prompts. A precise matching of entities from the NER module with DSL knowledge from the vector DB results in more effective and concise prompts.
- The well-organized prompts stimulate the understanding and generation capabilities of LLMs for DSL generation, further bolstering the effectiveness of the pipeline.

However, our pipeline does have some limitations. As it is not an end-to-end method, the quality of the generated DSL is directly influenced by each component in the pipeline. Incorrect identification by the NER module or insufficient DSL knowledge in the vector DB will impact DSL generation. Therefore, considerable effort is required to ensure the effectiveness of each module within the pipeline.

6 Conclusion

In this study, we present an innovative DSL generation pipeline for front-end low-code platforms, which incorporates IR, NER module, and DSL vector database along with LLM-based DSL generation. The NER module precisely extracts front-end page components from user requirements, facilitating an accurate match with the DSL vector database. The retrieved DSL knowledge is tailored for LLM to produce satisfactory results. Consequently, the pipeline method allows users to swiftly construct desired pages with simple natural language descriptions.

Acknowledgments. This work was supported by the National Natural Science Foundation of China (Grant No. U22A6001), the National Key Research and Development Program of China (No. 2023YFE0108600), and the Pioneer and "Leading Goose" R&D Program of Zhejiang Province (No. 2024SSYS0002).

References

1. Cai, Yuzhe, Shaoguang Mao, Wenshan Wu, Zehua Wang, Yaobo Liang, Tao Ge, Chenfei Wu et al. "Low-code llm: Visual programming over llms." arXiv preprint arXiv:2304.08103 2 (2023).
2. Zhao, Wayne Xin, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang et al. "A survey of large language models." arXiv preprint arXiv:2303.18223 (2023).
3. Yang, Zhen, Jacky Wai Keung, Zeyu Sun, Yunfei Zhao, Ge Li et al. "Improving domain-specific neural code generation with few-shot meta-learning." Information and Software Technology 166 (2024): 107365.

4. Wei, Jason, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph et al. "Emergent abilities of large language models." arXiv preprint arXiv:2206.07682 (2022).
5. Wang, Bailin, 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 36 (2024).
6. Ouyang, Long, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright et al. "Training language models to follow instructions with human feedback." Advances in neural information processing systems 35 (2022): 27730-27744.
7. Bai, Jinze, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang et al. "Qwen technical report." arXiv preprint arXiv:2309.16609 (2023).
8. Yin, Pengcheng, and Graham Neubig. "Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation." arXiv preprint arXiv:1810.02720 (2018).
9. Wang, Yue, Weishi Wang, Shafiq Joty, and Steven CH Hoi. "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation." arXiv preprint arXiv:2109.00859 (2021).
10. Liu, Aiwei, Xuming Hu, Lijie Wen, and Philip S. Yu. "A comprehensive evaluation of ChatGPT's zero-shot Text-to-SQL capability." arXiv preprint arXiv:2303.13547 (2023).
11. Tang, Xiangru, Bill Qian, Rick Gao, Jiakang Chen et al. "BioCoder: a benchmark for bioinformatics code generation with contextual pragmatic knowledge." arXiv preprint arXiv:2308.16458 (2023).
12. Brown, Tom, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan et al. "Language models are few-shot learners." Advances in neural information processing systems 33 (2020): 1877-1901.
13. Li, Jiazheng, Runcong Zhao, Yulan He, and Lin Gui. "Overprompt: Enhancing chatgpt capabilities through an efficient in-context learning approach." CoRR (2023).
14. Levy, Itay, Ben Bogin, and Jonathan Berant. "Diverse demonstrations improve in-context compositional generalization." arXiv preprint arXiv:2212.06800 (2022).
15. Wang, Jianguo, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu et al. "Milvus: A purpose-built vector data management system." In Proceedings of the 2021 International Conference on Management of Data, pp. 2614-2627. 2021.
16. Neelakantan, Arvind, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han et al. "Text and code embeddings by contrastive pre-training." arXiv preprint arXiv:2201.10005 (2022).
17. Steck, Harald, Chaitanya Ekanadham, and Nathan Kallus. "Is cosine-similarity of embeddings really about similarity?." In Companion Proceedings of the ACM on Web Conference 2024, pp. 887-890. 2024.
18. Sitikhu, Pinky, Kritish Pahi, Pujan Thapa, and Subarna Shakya. "A comparison of semantic similarity methods for maximum human interpretability." In 2019 artificial intelligence for transforming business and society (AITB), vol. 1, pp. 1-4. IEEE, 2019.
19. Guo, Rentong, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi et al. "Manu: a cloud native vector database management system." arXiv preprint arXiv:2206.13843 (2022).