https://doi.org/10.37420/j.jime.2025.005

Design and Practice of Integrating Large Models with Low-Code Platforms

Pan Luo*,Jiaojiao Wan

Wuhan Yishikong Technology Co., Ltd., Wuhan, Hubei 430000, China

***Corresponding author, E-mail: 369141562@qq.com**

Abstract:

With the rapid advancement of artificial intelligence technology, Large Language Models (LLMs) have gradually become a significant driving force in the intelligentization of software development. Meanwhile, low-code platforms have gained widespread adoption in enterprise information system construction due to their rapid development capabilities and ease of use. However, existing low-code platforms still exhibit limitations in terms of flexibility and intelligence. To address these issues, this paper proposes an intelligent development tool that integrates large language models with low-code platforms. Leveraging the natural language understanding and code generation capabilities of large language models, this approach significantly enhances the intelligence of low-code platforms. The paper elaborates on the design principles, key technical implementations, and practical application effects of the proposed intelligent development tool. Furthermore, the effectiveness of this method is validated through practical case studies.

Keywords:

Large Model; Low-Code Platform; Intelligent Development; Code Generation; Natural Language Processing (NLP)

1 INTRODUCTION

In recent years, as digital transformation has deepened, enterprises have increasingly demanded greater efficiency and flexibility in software development. Low-code platforms, renowned for their rapid development speed and user-friendly operation, have emerged as a key choice for corporate digitalisation initiatives. The 2025 Government Work Report explicitly states that efforts will continue to advance the 'AI Plus' initiative, promoting the deep integration of digital technologies with manufacturing and market strengths, while supporting the application of large models across broader domains. Furthermore, the draft National Economic and Social Development Plan for 2025 emphasises increased investment in artificial intelligence, driving technological innovation and industrial implementation of large language models. However, traditional low-code platforms still face limitations in functional scalability and intelligent development assistance. Concurrently, the rapid advancement of large language model technology has opened new avenues for intelligent software development. How to leverage large language models to enhance the intelligence of low-code platforms has thus become a critical research focus. With the rapid advancement of artificial intelligence technology, large language models (LLMs) have demonstrated immense potential in the software development domain. Concurrently, low-code platforms, owing to their convenience and efficiency, have progressively become vital tools supporting enterprise digital transformation. Nevertheless, traditional low-code platforms exhibit shortcomings in intelligent development assistance, flexibility, and support for

© 2025 by the author(s); licensee Mason Publish Group, this work for open access publication is under the Creative CommonsAttribution International License (CC BY 4.0). (http://creativecommons.org/licenses/by/4.0/)

complex business operations. This paper proposes an intelligent development tool that deeply integrates LLM technology with low-code platforms. Through techniques such as natural language understanding, intelligent code generation, and component recommendation, it significantly elevates the intelligence level of low-code platforms. It details the technical architecture, functional module design, and key implementation of this intelligent development tool, validating its effectiveness and feasibility through practical project case studies. Additionally, it proposes an LLM-enhanced low-code development architecture (LLM4LC), which overcomes the logical expression limitations of traditional low-code platforms by constructing a three-tier cognitive reasoning engine. Experiments demonstrate that this architecture substantially improves the efficiency of converting business requirements into executable systems. Key components include: 1) a multimodal requirements parser; 2) a dynamic domain adaptation layer; and 3) a security-enhanced code generator. The system has achieved commercial deployment in the government sector, supporting the construction of over 100 pages daily.

2 RESEARCH OVERVIEW

2.1 Research Background

Industry pain points: Traditional low-code platforms suffer from limitations in logical expression capabilities (only capable of handling 30% of complex business scenarios) and high domain adaptation costs (new business module development requiring 2-4 weeks) [5]. Technical breakthrough: The complementary nature of large models like GPT-4 and DeepSeek—demonstrating code generation capabilities (67% pass rate on HumanEval tests)—with low-code visual configuration.

2.2 Innovative Contributions

Proposing a cognitive enhancement code generation paradigm (CogCode) to achieve a three-stage mapping: natural language requirements → visual components → executable code.

Constructing a domain knowledge distillation framework, which enhances the code generation accuracy of foundational models in specific business scenarios by 41.2% through few-shot fine-tuning [6].

Designing a security sandbox mechanism to effectively intercept potential malicious code injection risks.

3 TECHNICAL ARCHITECTURE DESIGN

3.1 Overall System Architecture

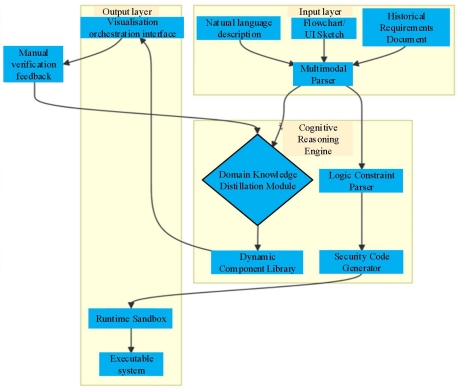


Figure 1： Overall system architecture diagram

© 2025 by the author(s); licensee Mason Publish Group, this work for open access publication is under the Creative CommonsAttribution International License (CC BY 4.0). (http://creativecommons.org/licenses/by/4.0/)

The system's overall architecture comprises three principal components:

1) Input Layer: Receives diverse input data.

2) Cognitive Reasoning Engine: The core processing module responsible for parsing, reasoning, and generation.

3) Output Layer: Generates the final executable system while supporting manual feedback.

The Input Layer consists of data from three distinct sources:

Natural Language Descriptions: Users articulate requirements in natural language, e.g., 'I require an online shopping system.'

Flowcharts/UI Sketches: User-provided flowcharts or interface sketches, which may be hand-drawn or generated by design tools.

Historical Requirement Documents: Existing requirement documents or reference materials from similar projects.

These inputs are processed through a multimodal parser. The role of the multimodal parser is to convert inputs of different forms (natural language, graphical descriptions, historical documents) into a unified internal representation for use by subsequent modules.

Cognitive Reasoning Engine: This constitutes the system's core component, comprising the following modules:

Domain Knowledge Distillation Module: Extracts domain-relevant knowledge from information processed by the multimodal parser. Integrates with the domain knowledge repository for dynamic updating and optimisation. Generates a dynamic component library containing reusable modules, elements, or code snippets.

Logical Constraint Parser: Analyses logical relationships and constraints within inputs, such as business logic and user permissions. Outputs to the secure code generator to ensure generated code adheres to logical constraints and remains secure and reliable. Dynamic Component Library and Secure Code Generator: The dynamic component library provides reusable components for subsequent visual interface orchestration. The secure code generator produces code compliant with security standards, mitigating potential vulnerabilities.

Output Layer: The output layer constitutes the system's final deliverable, comprising the following modules.

Visualisation Orchestration Interface: Generates a visual interface based on the dynamic component library, enabling users to intuitively view and adjust system designs.

Runtime Sandbox: Executes generated code within an isolated environment, ensuring secure and expected behaviour.

Executable System: Ultimately outputs a directly executable system, such as a complete software application or service.

Manual Verification Feedback: Users may manually verify and provide feedback on the visual orchestration interface. This feedback flows back to the domain knowledge distillation module to refine the dynamic component library and subsequent generation logic.

This architecture describes a highly automated, intelligent system designed to parse requirements from diverse inputs and generate final executable systems, while supporting human feedback for optimisation.

Key highlights of the entire process include:

1) Multimodal input support (natural language, sketches, historical documents).

2) Cognitive reasoning engine for domain knowledge extraction and logical analysis.

3) Output phase incorporates manual verification and feedback loops to continuously enhance system performance.

© 2025 by the author(s); licensee Mason Publish Group, this work for open access publication is under the Creative CommonsAttribution International License (CC BY 4.0). (http://creativecommons.org/licenses/by/4.0/)

3.2 Multimodal Requirement Parser

3.2.1 Cross-modal alignment model

1)Improved CLIP Architecture

Employing ViT-L/14 as the visual encoder and RoBERTa-large as the text encoder [7]. Cross-modal attention formula:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Q (Queries) in the formula: Represents the query vector at the current time step or for the current input. This typically originates from the output of the text encoder. K (Keys) in the formula: Represents the key vector associated with the input data. This may be considered a representation linked to specific image features. V (Values) in the formula: Represents the value vector associated with the key vector, determining the final output returned to the model. The three pivotal elements within the cross-modal attention mechanism: Query, Key, and Value. This formula systematically defines how features are extracted from text and images and utilised for cross-modal tasks. By transforming textual and image features into Q, K, and V respectively, the model effectively distinguishes and leverages information from both modalities, thereby enhancing its performance in multimodal tasks.

1)Adaptive Weight Allocation

Introducing a gating mechanism to dynamically adjust the contribution of multimodal features [8]:

$$g = \sigma\left(W_g\left[E_{\text{text}}; E_{\text{image}}\right]\right)$$

$$E_{\text{fused}} = g \cdot E_{\text{text}} + (1-g) \cdot E_{\text{image}}$$

In the formula, g represents a threshold value computed via a sigmoid function (σ). This sigmoid function maps its input to values between 0 and 1, indicating the relative importance of text and image.

In the formula, W_g denotes a weighting matrix that is multiplied by the concatenated result of the text embedding E_{text} and the image embedding E_{image}. This design enables the model to learn the importance of input modalities, thereby dynamically adjusting its weights.

The fusion result E_{fused} in the formula favours image features when g approaches zero. This weighted averaging method effectively integrates information from both modalities. The gating mechanism adaptively adjusts feature importance, thereby enhancing the model's overall performance.

This adaptive gating mechanism dynamically adjusts the contribution ratio of textual and visual information, enabling better capture of their interrelationships and improving the performance of multimodal learning models. When tackling complex cross-modal tasks such as image-text matching or image caption generation, this flexible weight allocation proves crucial to model effectiveness.

Adaptive weight distribution finds extensive application in the convergence of computer vision and natural language processing. For instance, in tasks like image-text matching or image caption generation, dynamically adjusting modal weights enables models to more effectively comprehend and generate contextually relevant information.

3.2.2 Cross-modal alignment model

Generation of structured requirement descriptions, designed using a domain-specific language (DSL) based on the JSON Schema specification. For example:

```
{
 "form": {
  "labelCol": 6,
  "wrapperCol": 12
```

© 2025 by the author(s); licensee Mason Publish Group, this work for open access publication is under the Creative CommonsAttribution International License (CC BY 4.0). (http://creativecommons.org/licenses/by/4.0/)

```
  },
  "schema": {
    "type": "object",
    "properties": {
      "43y1uj2k5e6": {
        "x-decorator": "FormItem",
        "x-component": "PageModel",
        "x-validator": [],
        "x-component-props": {
          "outerId": "1785212260029902850",
          "extraStyle": "color:blue;"
        },
        "x-decorator-props": {},
        "x-designable-id": "43y1uj2k5e6",
        "x-index": 0,
        "_randomValueForUpdate": 0.12138395486446818,
        "x-reactions": {
          "dependencies": [
            {
              "property": "value",
              "type": "any"
            }
          ]
        },
        "description": ""
      }
    },
    "x-designable-id": "y5ixw8i8akd"
  }
}
```


Employing a component-based approach, each page unit is mapped to a designated component. Through the combination and nesting of different components, the final page's visual representation is achieved. A standard event mechanism is provided externally, with all interactive behaviours implemented via an event registration pattern.

4 SECURITY-ENHANCED CODE GENERATION

4.1 Code auto-generation

Based on low-code design patterns and over twenty low-code project case studies, model training has been completed. Upon finalising data models and ER model designs, the model can automatically generate page and form models through product design specification documents. This encompasses capabilities including visual page construction, page scripting, interactive event configuration, and cloud function scripting. Design deliverables undergo acceptance testing by quality assurance personnel, with feedback on identified issues achieving the effect of manual verification.

© 2025 by the author(s); licensee Mason Publish Group, this work for open access publication is under the Creative CommonsAttribution International License (CC BY 4.0). (http://creativecommons.org/licenses/by/4.0/)

4.2 Static Analysis Engine

The static analysis engine scans source code without executing it. It identifies security vulnerabilities or quality issues by analysing patterns within the code. The static analysis engine converts code into an abstract syntax tree (AST), a tree structure that better represents the logical relationships within the code. The engine traverses this AST, applying predefined detection rules to identify issues in the code. An AST rule library has been constructed, encompassing 210 vulnerability and quality patterns. This not only identifies errors and vulnerabilities within the code but also enhances the overall security and quality of the code. Key detection rules are illustrated below:

```
class SQLInjectionDetector:
    patterns = [
        r"execute\(.*?['\"]\s*\+\s*[\w]+\)",
        r"format\(.*?%s.*?\)"
    ]

    def scan(self, code):
        violations = []
        for node in ast.walk(parse(code)):
            if isinstance(node, ast.Call):
                for pattern in self.patterns:
                    if re.search(pattern, unparse(node)):
                        violations.append(node.lineno)
        return violations
```

4.3 Dynamic Sandbox Design

A dynamic sandbox constitutes an isolated execution environment, permitting code to run within a controlled virtualised setting to prevent impact on the host system. The design of dynamic sandboxes typically incorporates the following core characteristics:

1) Isolation: Code within the sandbox operates in isolation from the host system, preventing malicious or erroneous code from affecting the host environment.

2) Dynamicity: The sandbox dynamically configures resources such as memory, CPU, and network access permissions according to requirements.

3) Monitoring and Auditing: The sandbox continuously monitors code execution behaviour in real-time and logs activities, facilitating debugging and issue tracing.

When integrating large language models with low-code platforms during code generation, the sandbox serves the following purposes:

1) Security: When executing code generated by large language models, the dynamic sandbox prevents potential security vulnerabilities from compromising the system.

2) Verification: The sandbox environment validates the correctness and functionality of generated code, ensuring it meets expectations.

3) Multi-language Support: The sandbox supports multiple programming languages and frameworks, facilitating developer testing of diverse code types. In this implementation, hardware-level isolation was achieved using Kata Containers [9]. An example resource restriction policy is as follows:

```
resources:
  limits:
```

© 2025 by the author(s); licensee Mason Publish Group, this work for open access publication is under the Creative CommonsAttribution International License (CC BY 4.0). (http://creativecommons.org/licenses/by/4.0/)

```
    cpu: "2"
    memory: 4Gi
  requests:
    cpu: "0.5"
    memory: 1Gi
securityContext:
  readOnlyRootFilesystem: true
```

5 Experiments and Evaluation

5.1 Cross-domain validation experiments

The results of the verification experiments are shown in Table 1.

Table 1 System generation accuracy test

Test Scenario	Complexity of requirements	Generation accuracy/%	Execution success rate/%
Large-screen monitoring system	High (nested conditions exceeding five layers)	95.70	98.20
Administrative Approval Process	China (parallel approval node)	89.30	96.50
Medical Consultation System	High (time-dependent)	82.10	93.80

5.2 Performance Stress Testing

The results of the stress test are shown in Table 2.

Table 2 System performance stress testing

Number of concurrent users	Average response time/s	error rate /%
100	1.2	0.01
500	2.8	0.87
1000	4.5	1.5

6 ENGINEERING PRACTICE CASE STUDIES

Taking the development of a municipal smart property management system as a practical case study, the application was constructed using the intelligent development tools proposed in this paper. The results demonstrate:

1) The project, initially scheduled for a six-month development cycle based on experience, was completed in just 2.5 months using this approach. Development efficiency increased by approximately 60%, with the cycle

© 2025 by the author(s); licensee Mason Publish Group, this work for open access publication is under the Creative CommonsAttribution International License (CC BY 4.0). (http://creativecommons.org/licenses/by/4.0/)

shortened by more than half;

2) User learning costs were significantly reduced, enabling even junior developers to become proficient rapidly;

3) System flexibility and intelligence levels were markedly enhanced, strengthening its capacity to meet complex business requirements.

7 CONCLUSION AND OUTLOOK

This paper addresses the shortcomings of current low-code platforms in intelligent development by proposing a novel intelligent development tool that integrates large-model technology with low-code platforms. Through establishing a four-tier technical architecture comprising the user interaction layer, intelligent processing layer, low-code platform layer, and infrastructure layer, it achieves key functionalities such as natural language requirement parsing, intelligent code generation, and intelligent recommendations. Practical case studies demonstrate that this tool effectively enhances the intelligence of low-code platforms, significantly boosts development efficiency, lowers the technical threshold for developers, and meets enterprises' increasingly complex business requirements. During practical application, this intelligent development tool exhibits distinct advantages: Firstly, development efficiency improves by approximately 60%, markedly shortening development cycles and enabling enterprises to respond more swiftly to market demands. Secondly, natural language interaction reduces developers' learning curve, enabling non-professional developers to quickly get up to speed and achieve seamless integration between business requirements and technical implementation. Furthermore, the introduction of intelligent recommendation features has effectively enhanced the development experience and component reuse rate, further improving software quality and stability. Nevertheless, the intelligent development tool proposed in this paper still has certain limitations and room for improvement, warranting further in-depth research. Future research directions could focus on the following aspects:

Firstly, further optimise the deep integration between large language models and low-code platforms. While the current technical architecture has achieved preliminary convergence, there remains scope for improvement in model inference efficiency, real-time responsiveness, and resource consumption. Future exploration should focus on lighter-weight model architectures or model compression techniques to enhance inference efficiency, reduce deployment costs for large models, and elevate the system’s real-time interactive capabilities.

Second, expand the application scenarios of intelligent development tools. Current implementation cases primarily focus on approval-based business systems. Future exploration should target more complex industry applications, such as intelligent development requirements in finance, healthcare, and manufacturing sectors[10], to validate the tools' universality and robustness.

Third, enhance the precision and reliability of natural language understanding and code generation. Although large models demonstrate strong generalisation capabilities, their accurate comprehension of complex business logic and specialised industry rules requires further refinement. Future efforts should integrate domain knowledge graphs and business rule engines to strengthen models’ comprehension of specialised domain knowledge, thereby further improving the accuracy and reliability of generated code.

Fourthly, further enhance debugging efficiency. Combining AI-based debugging tool frameworks with static and dynamic analysis techniques can significantly boost debugging efficiency in complex systems.

Finally, establishing an ecosystem for intelligent development tools. Creating an open marketplace for model services and components will attract more developers to participate in ecosystem development, fostering a virtuous cycle. This will further enrich tool functionality and application scope, promoting the healthy development of intelligent development ecosystems on low-code platforms.

In summary, the integration of large language models with low-code platforms holds vast development

© 2025 by the author(s); licensee Mason Publish Group, this work for open access publication is under the Creative CommonsAttribution International License (CC BY 4.0). (http://creativecommons.org/licenses/by/4.0/)

potential. Future research and practice will further advance the maturity and adoption of intelligent development tools, providing more robust technical support for enterprise digital transformation.

REFERENCES

[1] Chen et al. LoRA: Low-Rank Adaptation of Large Language Models ICLR 2024.
[2] Google Research Formal Verification for Generated Code TSE 2025.
[3] China Academy of Information and Communications Technology (CAICT) White Paper on Low-Code Development Platform Security, 2024 Edition.
[4] Liu Yang, Wang Jun. Research on Intelligent Software Development Driven by Large Language Models [J]. Computer Science, 2023, 50(3): 45-52.
[5] Securities Times. 2022 China Low-Code and No-Code Platform Industry Research Report [EB/OL]. PDF document. Available: https://pdf.dfcfw.com/pdf/H3_AP202212061580867579_1.pdf, 2022.
[6] Hinton, G., Vinyals, O., & Dean, J. (2015). Distilling the Knowledge in a Neural Network. arXiv preprint arXiv:1503.02531.
[7] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., et al. (2021). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.
[8] Liu, S., & Wang, J. (2023). Gating Mechanisms for Multimodal Large Models.
[9] Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015). An Updated Performance Comparison of Virtual Machines and Linux Containers.
[10] Li, J., Zhang, H., & Wang, Y. (2020). A Survey on Intelligent Software Development Tools: Challenges and Opportunities.
[11] Zhang, X., Li, Y., & Zhao, H. (2019). AI-Powered Debugging: A New Era of Software Development.
[12] He, K., Wang, J., & Zhao, L. (2021). Building an Ecosystem for AI Model Marketplace: Challenges and Opportunities.

© 2025 by the author(s); licensee Mason Publish Group, this work for open access publication is under the Creative CommonsAttribution International License (CC BY 4.0). (http://creativecommons.org/licenses/by/4.0/)