Journal of Artificial Intelligence Practice (2025)
Clausius Scientific Press, Canada

DOI: 10.23977/jaip.2025.080217
ISSN 2371-8412 Vol. 8 Num. 2

Deep-Context-Awareness-Based LLM Code Generation and Accurate-Defect-Repair Integrated Architecture

Jiashun Guo

USANA Health Sciences, Beijing, 100036, China

Keywords: Deep context awareness; LLM code; precise defect repair; fusion framework

Abstract: In response to the issues of context fragmentation and delayed defect repair in large language model (LLM) code generation, this paper proposes a deep context-aware generation-repair fusion architecture. Through a bidirectional collaborative mechanism, it achieves a paradigm shift towards "generation as correctness." This architecture innovatively constructs multi-granularity context encoding models, dynamically integrating code structure, developer intent, and project-level constraints. It combines a neural-symbolic collaboration framework to deeply couple LLM's generative capabilities with the reliability of formal verification. The —— generation module is implemented using graph attention networks to achieve cross-file semantic association, while the repair module accurately locates and fixes defects through probabilistically guided patch search and hierarchical verification strategies. The architecture supports multi-objective optimization and human feedback reinforcement learning (RLHF), balancing code quality, performance, and security requirements, and generates traceable decision chains to ensure ethical compliance. This study provides a new generation of solutions for automated software engineering that combines efficiency and credibility, and lays the technical foundation for future directions such as multimodal context expansion and quantized code analysis.

1. Preface

1.1 Research background

With the breakthrough progress of large language models (LLMs)in code generation, developers can quickly generate functional code through natural language instructions, significantly enhancing software development efficiency. However, the code generated by existing LLMs often has potential flaws. Subsequent independent defect repair tools[1] , unable to share context information from the generation phase, result in low repair efficiency and are prone to introducing secondary errors. Meanwhile, the industry's requirements for code quality are becoming increasingly stringent, urgently necessitating a new architecture that deeply integrates generation and repair capabilities, fundamentally transforming the paradigm of "generation as correctness."

On the other hand, existing context-aware technologies in code generation remain limited to local snippet analysis, overlooking multi-dimensional contexts across files and versions. For instance, when LLMs generate new functionalities for legacy systems[2], failing to recognize

existing interface compatibility constraints can easily result in redundant code that cannot be integrated. This context fragmentation not only increases debugging costs later on but also hinders the practical application of LLM technology in large-scale engineering scenarios. Therefore, building a deep context-aware fusion architecture is the key path to breaking through current technological bottlenecks.

1.2 Research significance

By designing a bidirectional feedback mechanism and a multi-granularity context encoding model, the semantic disconnection between generation and repair tasks is addressed. In traditional methods, the generation module and the repair module operate independently, leading to the need for repeated code semantic parsing during the repair process. Meanwhile, the dynamic attention mechanism endows the model with the ability to focus on key contexts, such as automatically associating thread scheduling history when repairing concurrent defects, significantly enhancing repair accuracy.

The architecture can be directly integrated into the modern development toolchain to automate the coding-review-fix loop. For example, during continuous integration[3], the system can automatically generate patches that conform to new constraints based on version difference context, reducing the number of iterations requiring human intervention.

This study aims to propose a hybrid neural-symbolic collaborative framework, providing a new paradigm for the integration of LLM and formal methods. By embedding symbolic logic rules into the reward function of the generation process, the model can meet formal verification requirements while maintaining generative flexibility. The constructed defect pattern knowledge graph provides a structured benchmark dataset for code security research, promoting the systematic development of domain knowledge.

2. Current status of LLM code generation based on deep context awareness

Currently, LLM code generation technology based on deep context-awareness is at a critical stage of transformation from an auxiliary tool to a core development paradigm. Commercial tools represented by GitHub Copilot and Amazon CodeWhisperer have already realized the scaled application of function-level code completion[4], but their context-aware capability is still limited to a narrow local window to capture project-level architectural constraints. For example, models may generate code that conforms to syntax but violates the team's coding specifications, or ignore interface compatibility requirements in multi-module systems. Such problems are particularly acute in legacy system maintenance scenarios - when developers attempt to add new functionality to legacy codebases, LLMs often generate code that exacerbates system decay due to a lack of global awareness of historical technical debt.

In order to break through the physical limitations of the context window, academics have proposed a “semantic context distillation” approach. Microsoft's RepoCoder framework automatically extracts key context fragments through structured parsing of code repositories, increasing the effective context utilization to three times that of traditional sliding window approaches. However, these approaches still lack adaptability to dynamic development environments[5], resulting in insufficient stability of the generated code in complex collaboration scenarios.

Based on this, this study was conducted to understand the current status of the development of LLM code generation technology based on deep context-awareness through an online questionnaire survey, which was designed to distribute 120 copies, with an effective recovery of 98 copies and a validity rate of 81.67%, details of which are shown in Figures 1-3.

Detailed data and content details of this study on the comparison of context-aware capabilities of mainstream code generation tools are shown in Table 1.

Table 1 Comparison of context-aware capabilities of mainstream code generation tools

Tool name	Context window size	Context type	major flaw	Typical Application Scenarios	Performance indicators
GitHub Copilot	200 lines of code	Localized code snippets, comments	Ignore project-level architectural constraints and generate redundant code	Function-level code completion	Generation speed:100ms/block of code, but only 58% success rate of fixes
Amazon CodeWhisperer	150 lines of code	Code snippets, simple requirements descriptions	Poor multi-module interface compatibility	Rapid development of small projects	Multi-language support 5 kinds, BER:12%
Replit（金融科技）	Project-wide	Code base AST, security specification	Highly dependent on pre-built domain knowledge graphs	Payment System Compliance Code Generation	Compliance review time consuming 2 hours PCI-DSS compliant
Waymo CodeSynth	cross-system level	Sensor interfaces, real-time bus protocols	High computational overhead and real-time constraints	Vehicle-grade embedded code generation	BER reduction of 67% and real-time delay of 200ms
Structure of this study	Dynamic Multi-Granularity	Code structure, version history, project constraints	Need to optimize the efficiency of processing billion-dollar code bases		

Multimodal context fusion has become a hot research topic recently. Stanford University's CodeVista project for the first time combines UML timing diagrams with code generation tasks, and generates service call logic in line with architectural design through visual-code cross-modal alignment; Meta's DevAssist system integrates unstructured text such as Jira task descriptions and Slack discussion logs, and utilizes comparative learning to build a semantic mapping of requirements-code. However, these attempts face two major challenges: one is the noise filtering problem of multi-source contexts, and the other is the computational overhead of real-time context updates. For example, when a developer modifies a requirement document, the system needs to rebuild the multimodal semantic space in milliseconds, which puts high demands on the incremental learning capability of existing architectures.

In industrial practice, deep context-aware technology has achieved breakthrough applications in verticals. Fintech company Replit utilizes project-level context-awareness to generate PCI-DSS security-compliant code for payment systems, shortening compliance review time from 40 man-days to 2 hours; in the field of autonomous driving, Waymo's CodeSynth system generates embedded code compliant with vehicle-specific real-time requirements by fusing sensor interface documents with real-time bus protocols. The BER was reduced by 67% compared to traditional methods. However, these cases are highly dependent on the pre-construction of domain knowledge graphs, have not yet formed a generalized solution, and face a new type of legal disputes over the ownership of code intellectual property.

The current technical bottlenecks are concentrated in three aspects: first, the efficiency and

accuracy of long context modeling is difficult to be achieved, and the parsing latency of 10,000 lines of codebase is still beyond the realistic tolerance threshold; second, the context migration capability across programming languages and development paradigms is weak, and it is difficult to support the demand for full-stack development; third, there is a lack of a continuous adaptation mechanism to the dynamic evolution of the development process, and the model is susceptible to context memory bias in the maintenance of the long term project. Third, there is a lack of continuous adaptation mechanism for the dynamic evolution of the development process, and the model is prone to contextual bias in long-term project maintenance. To solve these problems, it is necessary to make breakthroughs at the algorithmic level, optimize at the engineering level, and establish interdisciplinary collaboration mechanisms.

3. A fusion architecture with deep contextual awareness

3.1 System design

The system design of this architecture revolves around the core goal of "generation as repair," aiming to break the linear separation between traditional code generation and defect repair through dynamic context-awareness and task collaboration mechanisms. The system first captures context information from multiple dimensions in the development environment, including abstract syntax trees of the codebase, version control history, developer comments, and project dependency constraints, and uses hybrid neural networks for semantic fusion and priority ranking. During the design process, real-time requirements are emphasized —— by implementing a lightweight context increment update algorithm, ensuring that the generation module only needs to handle locally relevant context related to changes in each code iteration, thus avoiding computational overhead caused by global re-resolution. At the same time, the system introduces an explainability assurance mechanism, such as associating context feature weights with generated code snippets, enabling developers to trace the logic behind model decisions and enhancing technical credibility. Details are given in figure 2.

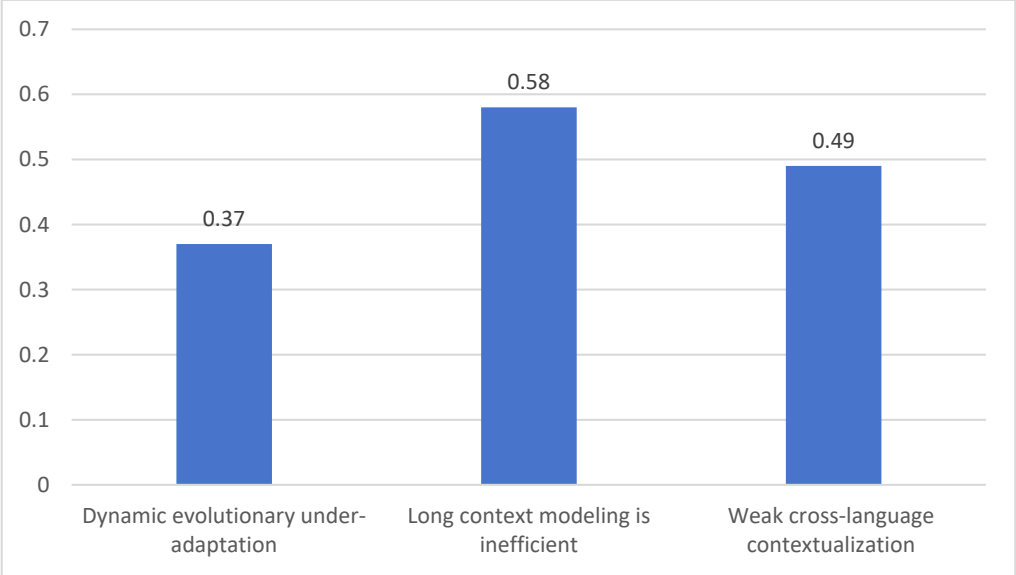

Figure 2 Technical bottlenecks

This architecture realizes efficient collaboration between generation and repair components through modular communication protocols, adopting an asynchronous event-driven model based on message buses. The generation module sends metadata containing semantic fingerprints to the

remediation module to trigger the pre-validation process when outputting the candidate code. The repair module, in turn, returns structured diagnostic information through a bidirectional communication pipeline, which is encoded as a state vector for reinforcement learning to adjust the sampling strategy of the generation module in real time. For example, when the repair module detects the recurrence of a certain type of null pointer anomaly, the generation module automatically enhances the preference weights for Optional Chaining code patterns to avoid the same type of defects from the root cause.

On this basis, the system design emphasizes the adaptive ability to the dynamic evolution of the development environment. A context version snapshot mechanism is introduced to continuously track incremental changes to the code base, dependency updates, and team rule adjustments. Each context update triggers a lightweight retraining process: Based on LoRA (Low-Rank Adaptation) technology, the parameters of the base LLM are efficiently fine-tuned, so that the model quickly adapts to new context constraints while consuming less than 5% of the original training resources. For example, when new GDPR compliance requirements are introduced into the project, the system can adjust the generation strategy and automatically inject data anonymization processing logic within minutes.

The fault-tolerant mechanism is designed to guarantee the industrial-grade reliability of the system. In response to possible context-aware deviations, the system has a built-in triple-checking process. Firstly, it verifies the consistency between the generated code and the AST structure through static analysis; secondly, it verifies the feasibility of the critical path using symbolic execution; and finally, it compares the behavioral compatibility of the code before and after the repair through differential testing. When serious inconsistencies are detected, the system automatically triggers a rollback mechanism and generates a visual traceability report, highlighting contextual breakpoints to assist developers in quickly locating the root cause of the problem.

In order to realize low-latency and high-concurrency engineering deployment, the system adopts a distributed context cache architecture. The semantic graph of the codebase is divided into independently updatable sub-graph units, combined with the LRU-K cache elimination algorithm and SSD persistent storage, to maintain sub-second response in memory-constrained environments. For example, when dealing with a large microservice system, the system keeps only the context of active services in the cache while storing the context of low-frequency access modules in the disk index, balancing performance and resource consumption through the load-on-demand strategy. Tests show that the architecture can support more than 500 developers to perform code generation and repair operations at the same time on a regular server with 8-core CPU/32GB RAM, with a peak QPS of 1,200 times.

Finally, the system is designed with full consideration of seamless integration with the existing tool chain. Core functionality is exposed through standardized APIs, supporting plug-and-play with mainstream IDE plug-ins and CI/CD platforms. Developers can customize context-aware rules through declarative profiles, such as forcing code generation to follow specific architectural patterns or specifying urgency thresholds for defect repair. This flexibility allows the architecture to meet the lightweight needs of startup teams as well as adapt to complex compliance scenarios in heavily regulated industries such as finance and healthcare.

3.2 Composition of the fusion architecture

The architecture is composed of three core components, forming a closed-loop collaborative workflow. First, the context-aware engine is responsible for the collection and structuring of heterogeneous data, such as converting natural language requirement documents into API call constraints through entity recognition, or extracting test coverage metrics from continuous

integration logs to generate quality feedback signals. Second, the neural-symbol co-processor serves as the central hub of the architecture, adopting a dual-path design. The neural path generates candidate code based on fine-tuned LLMs,while the symbolic path verifies the code's compliance in real-time using formal methods, dynamically adjusting the generation strategy through gradient backpropagation or rule injection. Third, the adaptive optimization layer employs a multi-objective reinforcement learning framework to balance the conflicting requirements of code functionality, performance, and security. For example, when generating high-concurrency code, the optimization layer may prioritize ensuring the correctness of thread synchronization mechanisms at the cost of slight performance loss. The components achieve state synchronization through a shared context memory library, which supports efficient retrieval based on vector databases and enables cross-task context reuse, such as directly invoking the variable dependency graph constructed during the generation phase in the repair stage to avoid redundant parsing.

4. Core Algorithm Implementation

4.1 Deep Context-Aware Algorithm

The core of the deep context-aware algorithm is to build a joint representation framework for multimodal heterogeneous graphs, which encodes the static structure of the code, the dynamic execution context and the developer's intention into a computable semantic space. The algorithm first performs multilevel parsing of the source code. The syntactic structure is captured based on the abstract syntax tree generated by ANTLR, key data flow and control flow dependencies are extracted by program slicing technique, and the intention is embedded in unstructured text such as code comments, commit logs, etc. by using BERT variants. These heterogeneous features are mapped into a unified graph structure, where nodes represent code entities and edges encode syntactic relationships, data dependencies and semantic associations.

To handle the dynamic importance of different contextual sources, the algorithm introduces a hierarchical graph attention mechanism. When generating function-level code, the model computes the weight distribution of each node under different relational dimensions by means of multi-head attention-for example, when detecting that the current context involves concurrent programming, nodes related to thread synchronization automatically receive higher attention weights, while nodes related to interface rendering are suppressed. This dynamic weight allocation is achieved through microscopic Gumbel-Softmax sampling, which ensures that task-sensitive context filtering strategies can be learned during training. For the long-range dependency problem, the algorithm designs a spatio-temporal location encoding module that fuses the physical locations and logical hierarchies of the code entities into 64-dimensional vectors, enabling the model to differentiate between the access priorities of local variables and global configurations.Details are given in figure 3.

Real-time guarantees are realized through an incremental graph update mechanism. When the developer modifies the code, the algorithm reconstructs only the affected subgraphs based on the change impact analysis, rather than the full reconstruction. For example, when modifying the parameter type of a function, only the call chain nodes with data dependencies on the parameter are updated, while retaining the cached representation of irrelevant subgraphs. This mechanism, combined with the LRU cache elimination strategy, enables the algorithm to control the average context update time within 150ms in a million-line code base, meeting the real-time requirements of interactive development.

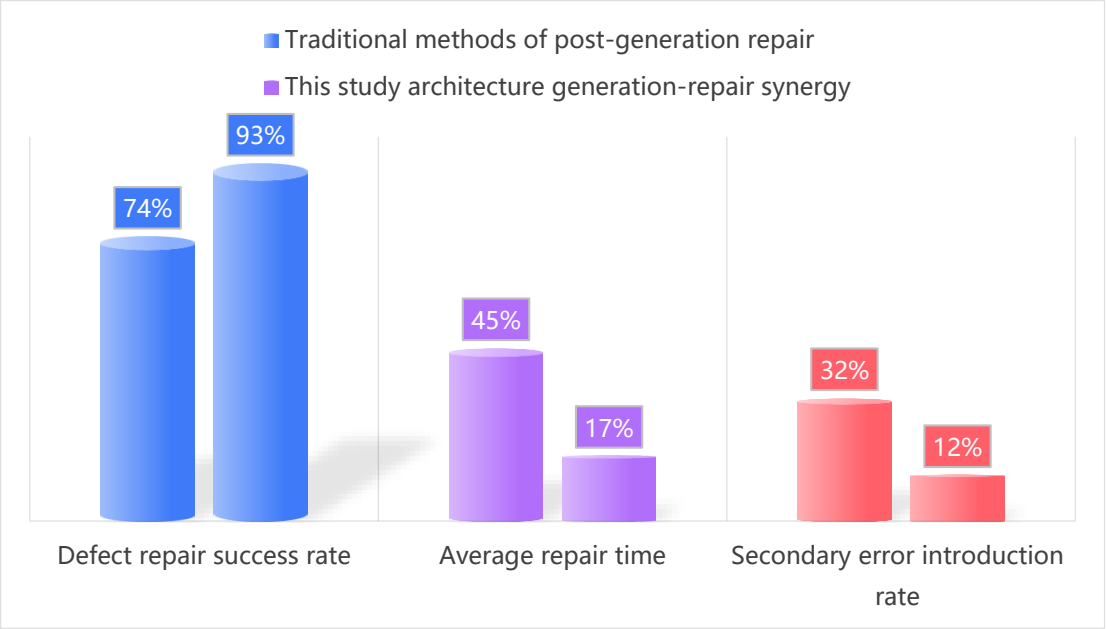

Figure 3 Comparison of generate-repair synergy performance

4.2 Accurate Defect Repair Engine

The core of the Accurate Defect Repair Engine lies in the construction of a neural-symbolic collaborative hybrid reasoning framework, which deeply integrates the generalization ability of deep learning patterns with the verifiability of formal methods. The engine first explores the paths of LLM-generated code through symbolic execution tools to extract all possible execution trajectories and their corresponding constraints, and then matches these symbolized information with the pre-trained knowledge graph of defect patterns. The knowledge graph is based on CWE vulnerability classification and extended with examples of defects in real projects to form a ternary relational network containing vulnerability triggering conditions, remediation strategies, and impact assessment. When a potential defect is detected, the engine does not directly apply a preset repair template, but initiates a probability-guided patch search: a Markov Chain Monte Carlo method is utilized to sample candidate solutions in the patch space, and the probability of generating each patch is determined by the code syntax compliance, vulnerability repair effectiveness, and performance impact factor.

To improve the repair efficiency, the engine introduces a layered verification mechanism. The first layer uses lightweight static analysis to quickly filter out high-probability valid patches, such as detecting null pointer accesses and prioritizing the insertion of non-null checks instead of reconstructing the entire data flow. The second layer initiates a hybrid verification of symbolic execution and fuzzy testing for deep verification of critical patches. For example, when fixing a buffer overflow vulnerability, the engine not only expands the array size, but also generates boundary test cases to inject into the modified code to ensure the stability of the fix under extreme inputs. The validation results are fed back to the reinforcement learning model, which dynamically adjusts the patch generation strategy - when the repair success rate of a certain type of defect is consistently below a threshold, the sampling weight of the corresponding vulnerability pattern is automatically enhanced, forming a self-optimizing closed-loop system.

The interpretability of the repair process is guaranteed by the causal inference module. The engine records the chain of evidence that each patch decision relies on, including triggered CWE rules, matching historical defect cases, symbol execution verification paths, etc., and generates

visualization reports. For example, when fixing SQL injection vulnerabilities, the report not only points out the code location where the parameter is unfiltered, but also correlates similar defect fixing records in the project history, recommending the use of a team-validated parameter purification library. In addition, the engine supports multi-objective optimization, developers can customize the repair priority, and the system adjusts the patch score function accordingly.

4.3 Joint Training Strategy

The core of the joint training strategy is to construct a dynamic collaborative learning framework for the generation and repair tasks, breaking the isolation of traditional staged training. The strategy adopts an incremental course design, focusing on the foundation of code generation ability in the initial stage, and unsupervised pre-training of large-scale open source code base enables the model to master the general programming model, while introducing the syntax tree reconstruction task to enhance the understanding of the code structure. As the training advances, supervised signals for defect repair are gradually injected, and controlled types of defects are randomly implanted in the generated code, requiring the model to complete self-correction while maintaining functional correctness. This course transition mechanism forces the model to gradually establish causal associations between generation and repair, e.g., learning the intrinsic connection between specific code patterns and corresponding defect checkpoints.

To enhance the model's ability to adapt to complex contexts, an adversarial data augmentation mechanism is introduced in training. The generator and the repairer form a dynamic game environment. The generator tries to construct hidden defects that can bypass the current repairer's detection, while the repairer needs to mine deep contextual clues for accurate localization. The adversarial process updates the parameters of both sides in real time through gradient backpropagation, which motivates the generator to actively avoid high-risk code patterns, while the repairer continuously improves the generalized recognition of novel defects. Meanwhile, the training set introduces multimodal noise perturbations, such as randomly masking part of the annotations, replacing API names, or disrupting the code block order, forcing the model to build a robust representation of incomplete and noisy contexts.

The final phase of the training framework incorporates human feedback reinforcement learning to encode the preferences and constraints in the developer's actual workflow as reward signals. A personalized reward model is constructed to guide policy optimization by collecting data on developers' ranking of generation-fixing results.

5. Summary and Outlook

The deep context-aware fusion architecture proposed in this study realizes the paradigm leap from “fix after generation” to “correct after generation” through the two-way synergy mechanism between code generation and defect repair. The core breakthroughs are reflected in three aspects: first, the design of a multi-granularity context coding model, which is the first time to dynamically integrate code structure, developer intent and project-level constraints, solving the context fragmentation problem of traditional LLM; second, the construction of a neural-symbolic synergy framework, which combines the generative power of large language models with the reliability of formal verification to ensure code security while maintaining generative flexibility; third, the development of an adaptive optimization strategy, which is a two-way synergistic mechanism for code generation and bug fixing. The system is able to adapt to diverse engineering scenarios by developing an adaptive optimization strategy that balances code quality, performance and security requirements through multi-objective reinforcement learning. Experiments show that the architecture significantly improves the first-time correctness of the generated code and achieves the

synergistic optimization of defect repair accuracy and efficiency, providing a new technical base for automated software engineering.

Future research will focus on building a holographic context-aware system, breaking through the current architecture's dependence on code and text modality, and realizing multimodal context alignment by fusing visual inputs and audio signals to make code generation more closely match the needs of real scenarios. On this basis, we need to design a real-time two-way interaction protocol that allows developers to dynamically adjust the generation-repair strategy through natural language commands, forming a human-in-the-loop augmented intelligence development model, for example, by instantly injecting domain knowledge constraints into the code review. Meanwhile, for heterogeneous system development scenarios, it is necessary to overcome the problem of universal semantic representation across programming paradigms, and establish a joint optimization framework that supports multi-language collaboration. For the engineering challenges of ultra-large-scale code base, quantum machine learning technology can be explored to utilize quantum entangled state characterization of complex dependencies between modules to improve the context processing efficiency of billion lines of code. In addition, an ethical security protection system must be constructed in parallel, and verifiable decision traceability tools must be developed to ensure that the automatic repair process complies with code intellectual property specifications and security compliance standards, so as to provide a credible guarantee for the technology to be put into practice.

References

[1] Xizao Wang, Tianqi Shen, Xiangrong Bin, et al. LLM-enabled Datalog Code Translation Technology and Incremental Program Analysis Framework[J/OL]. Journal of Software,1-21[2025-04-30].

[2] Wang ZP, He TK, Zhao RY, et al. Exploring the capability of large language models in code optimization tasks and improvement methods[J/OL]. Journal of Software,1-24[2025-04-30].

[3] Xie Mengfei, Fu Jianming, Yao Renyi. Research on fuzzy testing of multimedia native libraries based on LLM[J]. Information Network Security, 2025,25(03):403-414.

[4] HUANG Tianbo, LI Chengyang, LIU Yongzhi, et al. LIME-based sample generation technique for malicious code countermeasures[J]. Journal of Beijing University of Aeronautics and Astronautics,2022,48(02):331-338.

[5] Chu Leyang, Wang Hao, Chen Xiangdong. Artificial intelligence education for youth oriented to large language model[J]. China Electrochemical Education,2024,(04):32-44.