

Agentic Vulnerability Auditing for Secure API Code Generation

Authors

Ethan R. Collins^{1*}, Maya L. Bennett², Julian P. Hart³

Affiliations

Computer Science Department, Samueli School of Engineering, University of California, Los Angeles, Los Angeles, CA 90095, USA

*Corresponding author: er.collons@cs.ucla.edu

Abstract

Automatic code generation can accelerate software development, but generated API code often contains insecure authentication logic, weak input validation, unsafe deserialization, improper access control, and incomplete error handling. This study examines secure API code generation through agentic vulnerability auditing. We propose SecureAgent-Code, a five-agent model consisting of a requirement parsing agent, an API code generation agent, a security audit agent, an exploit test agent, and a repair agent. The requirement parsing agent extracts endpoint behavior, authentication rules, data constraints, and expected response formats from natural-language specifications. The code generation agent produces executable API programs in Python Flask, FastAPI, Node.js Express, and Java Spring Boot. The security audit agent checks generated code against CWE and OWASP Top 10 patterns, while the exploit test agent constructs adversarial requests for injection, privilege escalation, token misuse, and malformed payloads. The repair agent revises the code using static analysis reports, failed security tests, and runtime traces. Experiments were conducted on 2,480 API development tasks, including 9,920 endpoint specifications and 31,600 generated program versions. The evaluation used 126,000 automatically generated functional tests and 58,400 security attack cases. Compared with a single-agent code generator, SecureAgent-Code improved functional pass@1 from 61.7% to 76.9% and reduced high-risk vulnerability findings from 18.4% to 6.3%. SQL injection exposure decreased by 71.2%, broken access-control cases decreased by 64.8%, and insecure error-message leakage decreased by 42.5%. The average repair cycle converged within 2.6 iterations. These results show that agentic auditing and exploit-driven repair can improve both functional correctness and security reliability in API code generation.

Keywords: Code generation; secure software engineering; vulnerability auditing; program repair; API security; multi-agent systems; static analysis

1. Introduction

Large language models (LLMs) have substantially changed the way source code can be generated from natural-language specifications. Early systems such as Codex demonstrated that

large pre-trained models could synthesize executable programs directly from textual instructions, while CodeT5 showed that pre-training over source code could support both code generation and code understanding tasks [1-4]. AlphaCode extended this capability to more difficult programming problems that require multi-step reasoning and solution search [3]. At the same time, research on shortcut learning has shown that predictive models may rely on multiple superficial signals instead of the underlying task-relevant structure, and that identifying and removing these shortcuts can improve robustness [5-8]. This observation is important for code generation because a model may learn common implementation patterns that satisfy visible functional requirements without capturing the deeper constraints associated with secure software behavior. Open code models subsequently expanded multilingual and multi-language programming capabilities, making high-quality code generation more widely accessible [9-11]. The scope of evaluation has also moved beyond isolated functions toward realistic software-engineering tasks. SWE-bench evaluates model-generated modifications using real GitHub issues that may require changes across several files, while agent-based systems introduce testing, execution, environment interaction, and iterative repair into the generation process [7,8]. These developments demonstrate substantial progress in automated software development, but most existing evaluations still emphasize whether generated code compiles, executes, and passes functional tests. A program that satisfies these criteria may nevertheless contain exploitable security weaknesses[12-14].

The distinction between functional correctness and software security is particularly important for application programming interfaces (APIs). Modern APIs commonly expose authentication services, financial records, user profiles, clinical information, business data, and internal enterprise functions to external or distributed applications. Generated API code therefore operates at a security-sensitive boundary between users, services, databases, and application logic[15-17]. A functionally correct endpoint may still concatenate untrusted input into a database query, accept improperly validated tokens, expose resources without sufficient authorization checks, return sensitive debugging information, deserialize unsafe objects, or process malformed data without appropriate restrictions. Such vulnerabilities may remain invisible during ordinary functional testing because the expected request still produces the expected response[18-21]. Security failures frequently appear only when an attacker deliberately constructs inputs that violate assumptions made by the implementation [9,10]. Secure code generation consequently requires evaluation of adversarial behavior in addition to normal program functionality. Previous studies have raised concerns about the security quality of code produced by AI-assisted programming tools[22-26]. Empirical investigations have found that generated suggestions can contain insecure implementation patterns and that developers may accept these suggestions without performing sufficient security review [11,12]. Other work has examined whether language models can repair known vulnerabilities, showing that models can remove some recognized weaknesses but may also introduce functional regressions, incompletely repair the underlying problem, or replace one insecure construction with another [13,14]. These findings indicate that secure code generation

cannot be treated as a simple post-processing step in which a model is asked to “make the code secure.” Security weaknesses arise from different causes and affect different levels of an application, including input handling, authentication, authorization, data access, session management, error processing, and communication with external systems. Effective repair therefore depends on identifying both the weakness category and the runtime behavior that makes the weakness exploitable. Several established security frameworks provide useful taxonomies for this purpose[27-30]. The OWASP Top 10 summarizes major classes of web-application security risks, including injection, broken access control, authentication failures, insecure design, and security misconfiguration [15,16]. The Common Weakness Enumeration provides a more detailed classification of software weaknesses that can be associated with implementation-level defects [17,18]. These frameworks make it possible to evaluate generated API code against recognizable security categories rather than relying only on general code-quality judgments. However, a static indication of a potential weakness does not necessarily demonstrate that an application can actually be exploited. Static analyzers may report conditions that are unreachable at runtime, while subtle authorization or state-management vulnerabilities may escape pattern-based detection [31-34]. A comprehensive evaluation therefore requires both static reasoning about code structure and dynamic testing against a running application. Recent benchmarks have begun to incorporate security more explicitly into code-generation evaluation. CyberSecEval 2 examines a broad range of cybersecurity risks associated with language models, while CodeLMSec and LLMSecCode investigate insecure code generation under different models, prompts, and experimental conditions [21,22]. CWEval evaluates functional correctness and security properties within the same programming tasks, providing a closer connection between program behavior and weakness detection [23]. More recent repository-level benchmarks extend the evaluation toward realistic software contexts. SecRepoBench considers security issues within larger code repositories, while SecureAgentBench incorporates executable attacks, repository context, and static analysis into the assessment of coding agents [24]. These efforts represent an important shift from simple vulnerability counting toward more realistic security evaluation. Nevertheless, substantial gaps remain between existing benchmark settings and practical API development. Many security-oriented code benchmarks still rely on relatively short programs, restricted language environments[35-37], or a limited selection of weakness classes. Real API development requires coordinated reasoning about routing, authentication, database interaction, authorization, data validation, serialization, exception handling, and framework-specific behavior. The same security requirement may also require different implementation strategies across programming ecosystems. Input validation in Flask or FastAPI differs from request processing in Express, while authentication and access-control mechanisms in Spring Boot follow a different application structure. A secure code-generation system should therefore be evaluated across multiple frameworks rather than demonstrating performance within only one language or runtime environment. Repository-level context and framework-specific conventions also make it more

difficult to infer security requirements directly from surface-level prompt patterns. Another limitation concerns how security is validated after code generation. Static security reports are useful for locating suspicious patterns, but they provide incomplete evidence of exploitability[38-41]. An SQL injection warning becomes substantially more informative when a crafted request can actually alter the intended query behavior. A potential access-control weakness should ideally be tested by attempting to access another user's resource with insufficient authorization. Token-handling defects can be evaluated using expired, malformed, modified, or incorrectly scoped credentials. Error handling can be tested by sending malformed requests and observing whether stack traces, database details, file paths, or other sensitive information are returned. Dynamic exploit testing therefore changes the evaluation question from whether a weakness pattern may exist to whether an attacker can use that weakness against the running API. Security repair also benefits from separating different forms of feedback. A generated endpoint may fail because the application does not start, because its normal response is incorrect, because a static analyzer identifies a dangerous construct, or because an exploit test succeeds. These signals describe different problems and should lead to different repair actions. Compilation and runtime failures require implementation correction, functional test failures require restoration of expected behavior, static reports identify suspicious code patterns, and successful attacks provide direct evidence that a security control is ineffective. Combining all failures into one generic regeneration prompt can obscure the source of the problem and may lead the model to make unnecessarily broad changes. A structured repair process can instead preserve working functionality while targeting the specific security behavior responsible for the failure.

This study proposes SecureAgent-Code, a multi-agent framework for secure API code generation that explicitly separates requirement interpretation, implementation, security analysis, adversarial testing, and code repair. The framework begins with a requirement-parsing agent that extracts endpoint behavior, authentication requirements, authorization rules, input constraints, data-access conditions, and expected response formats from the natural-language specification. A code-generation agent then produces executable APIs using Flask, FastAPI, Express, or Spring Boot. The generated implementation is examined by a dedicated security-audit agent using CWE- and OWASP-oriented weakness categories. Rather than relying exclusively on static inspection, an exploit-testing agent launches the generated API and sends adversarial requests designed to test injection, token misuse, access-control bypass, malformed input, and related security conditions. A repair agent then integrates static findings, functional-test failures, exploit results, and runtime logs to revise the implementation while preserving the required API behavior. This architecture is designed to make security validation an explicit part of the generation process rather than an optional check performed after functional completion. The framework is evaluated on 2,480 API development tasks, covering 9,920 endpoint specifications and producing 31,600 program versions across four widely used API frameworks. Functional tests are combined with executable security attack cases so that generated programs are assessed according to both expected

application behavior and resistance to adversarial inputs. The study examines whether explicit requirement parsing improves the implementation of security-relevant constraints, whether dedicated security review identifies weaknesses that remain hidden after ordinary functional testing, whether runtime exploit testing provides stronger evidence than static warnings alone, and whether targeted repair can reduce vulnerabilities without degrading API functionality. The broader purpose of SecureAgent-Code is to move automated code generation beyond the conventional objective of producing programs that merely compile and pass normal tests. By integrating semantic requirement analysis, framework-specific generation, static security reasoning, executable attacks, and feedback-driven repair, the proposed approach seeks to establish a more reliable methodology for evaluating and improving the security of LLM-generated software. This is particularly important for API development because generated endpoints frequently mediate access to sensitive data and business services, where a functionally correct but insecure implementation can create direct operational and security risks. SecureAgent-Code therefore contributes both a systematic multi-agent architecture for secure code generation and an experimental framework for studying whether LLM-based software agents can satisfy functional and security requirements simultaneously under realistic runtime conditions.

2. Materials and Methods

2.1. Dataset and Study Scope

The study included 2,480 API development tasks, 9,920 endpoint specifications, and 31,600 generated program versions. Each task contained four endpoint specifications. A task was included only when it stated endpoint behavior, input fields, expected outputs, and at least one security rule. The security rules covered authentication, user roles, input limits, or error handling. The tasks included user registration, login, data queries, record updates, file handling, and administrator actions. Programs were generated in Python Flask, FastAPI, Node.js Express, and Java Spring Boot. Tasks with missing request fields, unclear outputs, or incomplete security rules were removed before testing.

2.2. Experimental Design and Baselines

SecureAgent-Code was compared with a single-agent code generator. Both methods received the same API specification, programming language, framework, model setting, token limit, and runtime limit. The baseline generated code in one step. It could revise the code only after a functional test failed. SecureAgent-Code used five agents for requirement parsing, code generation, security review, attack testing, and repair. Two reduced versions were also tested. One version removed the security review agent, and the other removed the attack test agent. These comparisons measured the separate effects of static security review and runtime attack testing. Each output was assessed with the same functional tests and security cases.

2.3. Measurement Methods and Quality Control

Functional performance was measured with 126,000 automated tests. The tests checked HTTP status codes, response fields, database changes, authentication behavior, and error handling. Security performance was measured with 58,400 attack cases. These cases covered injection, access-control bypass, token misuse, malformed requests, unsafe deserialization, and error-message leakage. Static analysis tools used CWE-based rules. The attack test agent sent requests to APIs running in separate containers. A finding was recorded only when it matched a defined insecure code pattern or a successful security test. Each test case was checked with secure and intentionally vulnerable reference programs before use. Duplicate cases, failed scripts, and unstable runtime results were removed. Database records and access tokens were reset before each run.

2.4. Data Processing and Evaluation Formulae

Each requirement was converted into fixed fields for the HTTP method, endpoint path, input schema, user role, authentication rule, expected response, and error condition. Source files, static-analysis reports, test logs, and repair records were linked by task identifier. Functional performance was calculated as

$$\text{Pass@1} = \frac{1}{N} \sum_{i=1}^N I(F_i=1)$$

In this formula, N is the total number of API tasks, i is the task index, F_i is the functional result for task i , and $I(\cdot)$ is the indicator function. If the first generated program passed all functional tests, $F_i=1$; otherwise, $F_i=0$. The high-risk vulnerability rate was calculated as

$$\text{HVR} = \frac{1}{N} \sum_{i=1}^N I(V_i^{\text{high}} \geq 1) \times 100\%.$$

Here, V_i^{high} is the number of confirmed high-risk vulnerabilities in program i . A lower HVR value indicates fewer serious security problems.

2.5. Implementation and Statistical Analysis

All APIs were run in Docker containers with separate network, file, and database settings. The repair process ended when a program passed all functional and security tests or reached five repair cycles. The same model version, prompts, temperature setting, and hardware limits were used in all groups. Results were calculated at the task level. Repeated tests from the same API were not treated as independent samples. Paired bootstrap resampling with 1,000 samples was used to calculate 95% confidence intervals for pass@1, high-risk vulnerability rate, and repair cycles. SecureAgent-Code and the single-agent baseline were compared with paired task-level results.

3. Results and Discussion

3.1. Functional Performance

SecureAgent-Code obtained a functional pass@1 of 76.9%, while the single-agent generator obtained 61.7%. The difference was 15.2 percentage points, equal to a 24.6% increase over the baseline. The result shows that requirement parsing before code generation helped the system follow endpoint rules, request fields, response formats, and authentication requirements. The functional tests checked HTTP responses, database updates, access behavior, and error handling. They did not only check whether the code could run. The repair stage also preserved normal API functions. In the single-agent setting, code was revised after a failed functional test, but security requirements could be missed during revision.

3.2. High-Risk Vulnerability Results

The high-risk vulnerability rate fell from 18.4% to 6.3%. This was a reduction of 12.1 percentage points, or 65.8% relative to the baseline. Static analysis found insecure code patterns before execution. Attack tests then checked whether the API could be misused through malicious requests, invalid tokens, or unauthorized roles. The repair agent used both types of feedback when updating the code. Fig. 1 presents a related audit-and-repair process for LLM-generated code. The remaining high-risk findings show that automated repair cannot identify or fix every security problem [25,26].

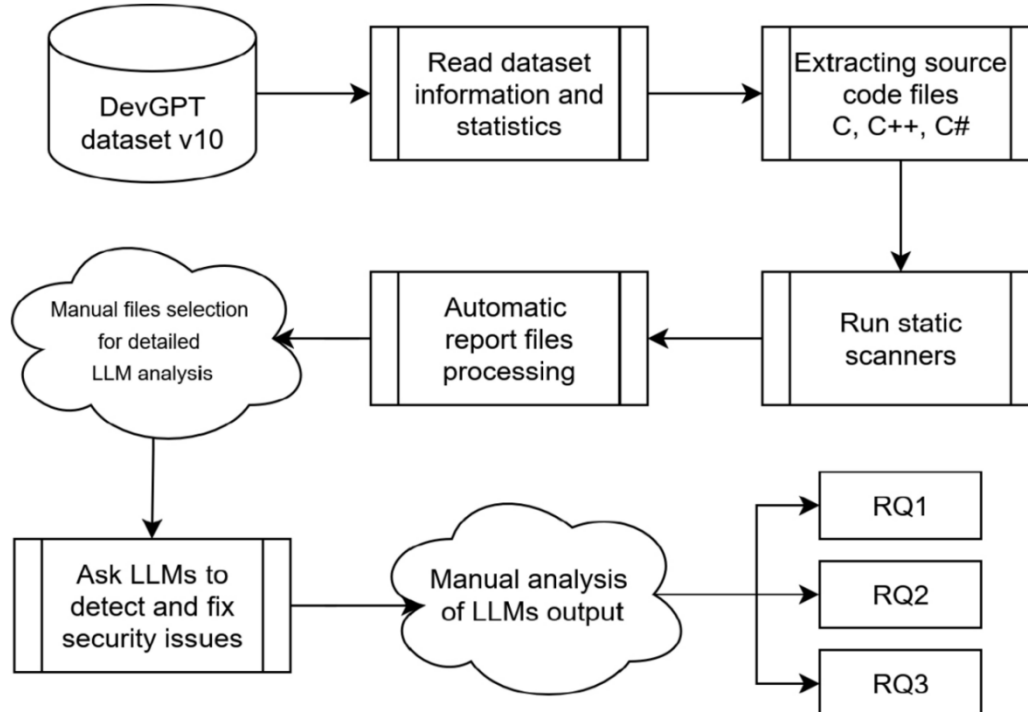


Fig. 1. Security review and repair steps for generated API code.

3.3. Results by Vulnerability Type

SQL injection exposure decreased by 71.2%, broken access-control cases decreased by 64.8%, and insecure error-message leakage decreased by 42.5%. SQL injection was reduced most because common input patterns and attack requests can reveal this problem. Access-control errors need information about user roles, resource ownership, and endpoint rules. They are harder to detect with one request or one source-code rule. Error-message leakage showed the smallest reduction because an API may return the correct status code but still expose internal details in its response [27,28]. Fig. 2 shows that vulnerability types occur at different rates in LLM-generated code. These results support separate reporting for each vulnerability type.

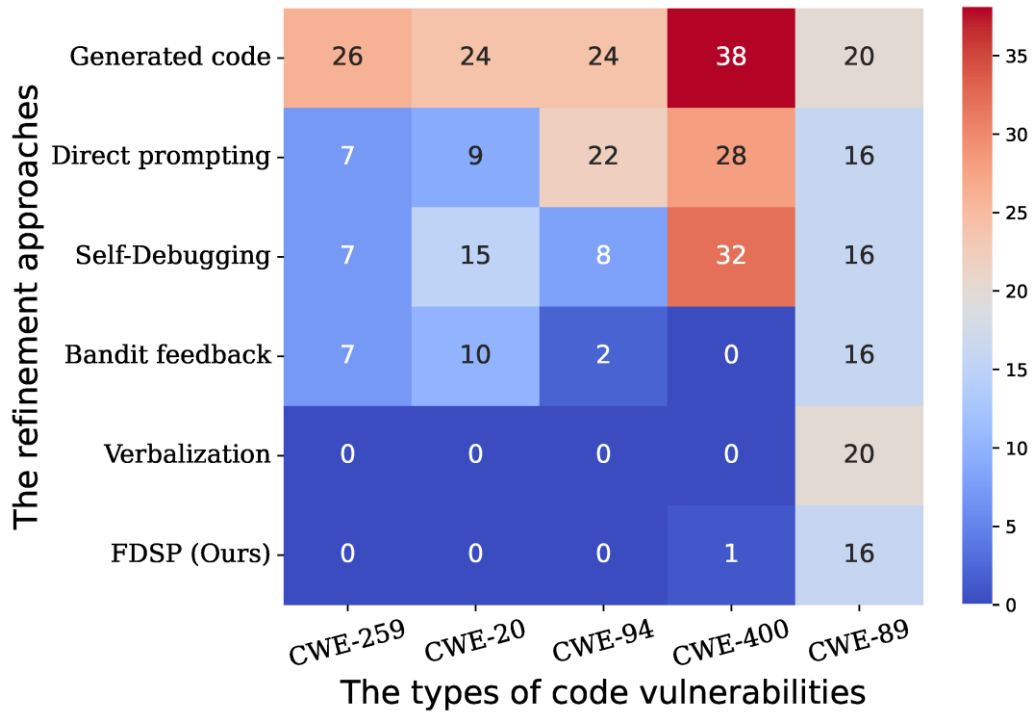


Fig. 2. Common security issues found in generated API code.

3.4. Repair Cycles and Comparison with Earlier Work

The mean number of repair cycles was 2.6. This value was below the five-cycle limit and suggests that many tasks were resolved in the early repair rounds. A previous feedback-driven patching study reduced vulnerability findings by up to 33% with Bandit and 12% with CodeQL. The present result cannot be compared directly with that study because the datasets, programming languages, security tools, and outcome measures were different [29,30]. SecureAgent-Code used four API frameworks and combined static reports with attack results and runtime traces. The main limitation is that automated tests may miss business-rule errors, dependency risks, and security issues that appear only after deployment.

4. Conclusion

This study assessed SecureAgent-Code, a five-agent system for API code generation and security checking. The system raised functional pass@1 from 61.7% to 76.9% and lowered high-risk vulnerability findings from 18.4% to 6.3%. It also reduced SQL injection, access-control errors, and unsafe error messages. The results show that combining code review, attack tests, and program logs can improve both API function and security. The system can be used in API development, code review, and software training. This study used fixed API tasks, automated tests, and four development frameworks. It did not fully test business-rule errors, external software packages, or live deployment settings. These issues should be assessed in real software projects.

References

1. Qi, C., & Qiao, X. (2026). Building Reliable AI Systems: A Framework That Bridges Architecture and Operations. Available at SSRN 6795298.
2. Balasingam, K. (2026). Large Language Model-Based Intelligent Code Generation and Automated Software Engineering Framework. *Research Journal of Computer Systems and Engineering*, 44-49.
3. Xiong, W., Zeng, Y., & Guo, Y. (2026). A Study on the Impact of MEP Space Organization in Large-Scale Commercial Complexes on Investment Efficiency and Its Mechanisms.
4. Zheng, J., & Makar, M. (2022). Causally motivated multi-shortcut identification and removal. *Advances in Neural Information Processing Systems*, 35, 12800-12812.
5. Que, H. H., Jin, Y., Wang, T., Liu, M. K., Yang, X. H., & Qiao, F. (2023). A Survey of Approximate Computing: From Arithmetic Units Design to High-Level Applications. *Journal of Computer Science and Technology*, 38(2), 251-272.
6. Hong, Z., Liang, S., Xu, T., & Chen, H. (2026). A Study on Failover Verification and Recovery Objective Prediction for Cross-Region Cloud Services.
7. Jiao, Y., Shi, T., Zhao, B., & Wang, A. (2026). Retrieval-Guided Structured Reasoning and Interpretable Representation Learning for Large-Scale Video–Language Models.
8. Fan, Y., Miyazaki, T., Tang, Z., Wang, J., Huang, Y., & Omachi, S. (2025, November). Scene Text Reconstructor: A Contextual-Aware Masking Framework for Pre-training Text Detectors. In *International Conference on Neural Information Processing* (pp. 181-195). Singapore: Springer Nature Singapore.
9. Liang, S., Du, Y., Chen, W., & Liu, Z. (2026). Order Allocation and Emergency Transportation Decisions Amid Multi-Source Procurement Disruptions.
10. Zhang, G., Xu, Y., Zhang, M., Wang, S., & Lin, N. (2021). The brain network in support of social semantic accumulation. *Social cognitive and affective neuroscience*, 16(4), 393-405.
11. Bao, Y., & Wang, H. (2026). Large-scale Metrics Migration: A Systematic Approach to Rebuilding Production Measurement Infrastructure. Available at SSRN 7185660.

12. Mini Han Wang , " AI-Powered Innovations in Ophthalmic Diagnosis and Treatment ", Bentham Science Publishers (2025). <https://doi.org/10.2174/97988988122871250101>.
13. Bao, Y., Qiu, Y., & Wang, H. (2026). FLARE: A Real-Time Framework for Detecting Malicious Account Activities at Internet Scale. Available at SSRN 7185659.
14. Wang, S., Feng, Y., & Fang, X. (2026, May). A Large Language Model-Enabled Multi-Agent Collaboration Method for Complex Task Solving. In 2026 6th International Symposium on Computer Technology and Information Science (ISCTIS) (pp. 253-256). IEEE.
15. Yang, Q., & Du, Y. (2026). Predicting Rollback Risks and Controlling Gradual Rollout Traffic for Online Ranking Feature Deployments.
16. Liu, W., Zhao, R., Sha, Z., Cui, Q., & Zhang, Y. (2026). Mapping the City Through the Lens of Language Models. arXiv preprint arXiv:2608.02971.
17. Liang, S., Hsu, K. H., & Liu, Z. (2026). Rolling Scheduling and Capacity Balancing for Server Assembly Under Engineering Change Disturbances.
18. Yuan, Y., Huang, J., Yin, J., & Xu, T. (2026). Confidence Calibration and Semantic Error Analysis for Ambiguous Coreference Resolution in User-generated Social Media Text. Available at SSRN 7393238.
19. Zhao, Z., & Welsch, R. E. (2024). Hierarchical reinforced trader (hrt): A bi-level approach for optimizing stock selection and execution. arXiv preprint arXiv:2410.14927.
20. Han, Z., Chen, W., Han, Y., Mao, R., & Qin, J. (2026). Fast Diversified Top-k Rule Discovery via User-Guided Embeddings. IEEE Transactions on Knowledge and Data Engineering.
21. Yang, Z., Alexandrova, A. N., & Sautet, P. (2026). Modeling CO₂ Hydrogenation to Methanol on an Ensemble of Inverse ZrO₂ on Cu Catalytic Sites: Mechanism, Reactivity, and Deactivation. *Angewandte Chemie*, e5448247.
22. Huang, J., Xu, T., Yang, J., & Yin, J. (2026). Blockage Early Warning and Financial Impact Quantification in the Monthly Closing Process of Fintech Companies. Available at SSRN 7179198.
23. Quan, Q., Gao, Y., & Wang, Q. (2025). The impact of teacher emotional support on students' engagement in AI-mediated English learning environments: The mediating role of resilience and self-efficacy. *Acta Psychologica*, 260, 105766.
24. Gu, X. (2026). Identifying Causal Effects and Analyzing Heterogeneity of User Growth Interventions on Digital Platforms: Evidence from Large-Scale Behavioral Data. Available at SSRN 6809181.
25. Zhao, J., Fan, J., & Li, L. (2026). A Study on an Explainable Causal-Enhanced LLM Agent for Predicting the Forming Quality of Automotive Component Materials.
26. You, S. (2026). Verifiable Audit Mechanisms in AI Compliance Automation: Scalability. Available at SSRN 6547458.
27. Zhao, Z., & Welsch, R. E. (2026). Point-in-Time Financial RAG with Frozen LLMs and Market-Feedback Adaptive Retrieval. arXiv preprint arXiv:2605.31201..

28. Yin, J., Rao, H., & Huang, Y. (2026, March). Dynamic Modeling and Heterogeneity Analysis of Platform User Behavior Time Series. In 2026 International Conference on AI in Education Technology and Applications (AIETA) (pp. 30-33). IEEE.
29. Wang, Z., Yu, J., Liu, H., Zheng, Z., Jin, Y., Li, S., ... & Zhang, K. (2025, July). Generative Music Models' Alignment with Professional and Amateur Users' Expectations. In Findings of the Association for Computational Linguistics: ACL 2025 (pp. 6909-6920).
30. Zhang, Z. (2026). A Study on the Impact of Cost Presentation on Decision-making Bias and Cancellation Behavior in Online Subscriptions. Available at SSRN 7349839.
31. Xu, Y., Sun, Y., Zhai, B., Li, M., Liang, W., Li, Y., & Du, S. (2025, April). Zero-shot video moment retrieval via off-the-shelf multimodal large language models. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 39, No. 9, pp. 8978-8986).
32. Zheng, X., Chen, X., Gong, S., Griffin, X., & Slabaugh, G. (2025, September). Xfmamba: Cross-fusion mamba for multi-view medical image classification. In International Conference on Medical Image Computing and Computer-Assisted Intervention (pp. 672-682). Cham: Springer Nature Switzerland.
33. Su, J., Cai, C., Zhu, F., He, C., Xu, X., Guan, D., & Si, C. (2024, September). Momentum auxiliary network for supervised local learning. In European Conference on Computer Vision (pp. 276-292). Cham: Springer Nature Switzerland.
34. Li, P., Zhang, H., Li, W., Huang, D., Guo, Z., Chen, J., ... & Koshizuka, N. (2025). GeoAvatar: A big mobile phone positioning data-driven method for individualized pseudo personal mobility data generation. *Computers, Environment and Urban Systems*, 119, 102252.
35. Lyu, Y., & Ding, R. (2025). Discovering Self-Regulated Learning Patterns in Chatbot-Powered Education Environment. arXiv preprint arXiv:2510.01275.
36. Xi, X., Chen, P., Qiu, Y., Luo, R., Tong, P., & Liang, J. Video-BCI: Bayesian Cognitive Integration of Self-Prior Hypotheses for Video Understanding. In Forty-third International Conference on Machine Learning.
37. Liang, J., Xiao, Y., Yang, H., Yu, Z., Liu, J. N., & Yang, K. (2026, April). C-HyPOD: Causal Hyperbolic Representation Learning with Prototype Orthogonal Disentanglement for Graph Out-of-Distribution Recommendation. In Proceedings of the ACM Web Conference 2026 (pp. 6541-6550).
38. Xu, Z., Zhang, X., Li, R., Tang, Z., Huang, Q., & Zhang, J. (2024). Fakeshield: Explainable image forgery detection and localization via multi-modal large language models. arXiv preprint arXiv:2410.02761.
39. Jia, Z., Huang, C., Wang, Z., Fei, H., Wu, S., & Feng, J. (2024). Finger recovery transformer: toward better incomplete fingerprint identification. *IEEE Transactions on Information Forensics and Security*, 19, 8860-8874.

40. Huang, C., Jia, Z., Fei, H., Zhu, Y., Yuan, Z., Zhang, J., & Zhou, J. (2025, October). ArtFRD: A Fisher-Rao Mixture Metric for Generative Model Aesthetic Evaluation. In Proceedings of the 33rd ACM International Conference on Multimedia (pp. 6654-6662).
41. X. Chen, C. Huang, G. Chen, D. Feng and P. Xiao, "The Communication and Computation Trade-Off in Wireless Semantic Communications," in IEEE Wireless Communications Letters, vol. 14, no. 7, pp. 2259-2263, July 2025, doi: 10.1109/LWC.2025.3569205.