China and WTO Review (CWR) Journal
2026, (13)
ISSN: 2384-4388
https://cwto.net

Research Article

Verified Smart Contract Generation with Agentic Testing and Gas Optimization

Fan Long [1], Daniel R. McAllister [1*] Chloe A. Bennett [1]

[1] Department of Computer Science, Faculty of Arts & Science, University of Toronto, Bahen, Canada
* **Corresponding Author**:fanl@cs.toronto.edu

ARTICLE INFO

ABSTRACT

Smart contract development requires strict correctness because small coding errors may lead to financial loss, contract lockup, or exploitable vulnerabilities. Large language models can generate Solidity contracts from natural-language requirements, but the generated code may contain reentrancy risks, integer-handling errors, access-control flaws, inefficient storage operations, and incomplete event logic. This study investigates verified smart contract generation through agentic testing and gas optimization. We propose ContractAgent-V, a multi-agent model including a requirement formalization agent, a Solidity generation agent, a vulnerability inspection agent, a property-based testing agent, and a gas optimization agent. The requirement formalization agent extracts contract roles, state variables, transaction rules, permission constraints, and failure conditions. The Solidity generation agent produces deployable smart contracts. The vulnerability inspection agent checks the code against common weakness patterns and symbolic execution warnings. The property-based testing agent creates randomized transaction sequences to verify balance consistency, ownership restrictions, state transitions, and revert behavior. The gas optimization agent reduces unnecessary storage writes, simplifies modifiers, optimizes data structures, and removes redundant computations. Experiments were conducted on 1,120 smart contract tasks covering token issuance, escrow, crowdfunding, voting, staking, auction, access control, and decentralized marketplace logic. The benchmark included 6,720 generated contracts, 43,500 property-based tests, and 18,200 vulnerability probes. Compared with a single-agent Solidity generator, ContractAgent-V improved deployment success from 69.3% to 84.7% and functional test pass rate from 61.8% to 79.6%. High-severity vulnerability findings decreased from 15.2% to 4.9%, while access-control violations decreased by 58.1%. Gas consumption for core transaction functions decreased by 23.4% on average after optimization. These results suggest that agentic verification and gas-aware repair can improve the reliability and efficiency of automated smart contract development.

Keywords: Smart Contract Generation, Solidity; Program Verification, Gas Optimization, Vulnerability Detection, Multi-Agent Systems, Automated Software Engineering.

INTRODUCTION

Smart contracts are self-executing programs that enforce predefined transaction rules on blockchain networks. They are widely used to manage digital tokens, deposits, access permissions, auctions, decentralized finance services, and other asset-related operations. Unlike conventional software, deployed smart contracts are often difficult to modify because their code and state transitions are permanently recorded on the blockchain. Consequently, even a small implementation error may lead to failed transactions, unauthorized asset transfers, locked funds, or irreversible financial losses. This characteristic places unusually high requirements on code correctness, security, and execution reliability. Existing reviews of smart-contract auditing tools have shown that different analyzers may produce substantially different findings because they target different vulnerability classes, adopt different execution assumptions, and use different detection strategies [1]. Formal verification and static analysis therefore remain important approaches for identifying defects before deployment. Refinement-type systems have been used to verify arithmetic safety and rule out selected classes of numerical errors [2], while other studies have explored syntactic features, semantic representations, and learning-based code analysis to

identify vulnerabilities in Solidity programs [3]. Efficiency-oriented research in other computational domains has further shown that globally informed pruning can remove redundant computation while preserving task-relevant information, indicating that optimization decisions benefit from global structural information rather than isolated local signals [4]. This principle is also relevant to smart-contract optimization, where reducing individual instructions is useful only when the overall transactional behavior of the contract remains unchanged. Hybrid approaches that combine language models with program analysis have consequently attracted increasing attention for code understanding, vulnerability identification, and automated repair [5].

Static inspection, however, cannot fully determine whether a contract remains correct after a long sequence of state-changing transactions. Many vulnerabilities depend not only on individual statements but also on the order of function calls, changes in persistent storage, interactions among users, and the cumulative effect of earlier transactions. Invariant-generation methods provide one way to describe expected contract properties and detect abnormal state changes [6,7]. Dynamic analysis and transaction-sequence testing extend this idea by exploring execution paths and persistent contract states. SmarTest applies language-model-guided symbolic execution to search for vulnerable transaction sequences [8], SMARTIAN combines static and dynamic information to improve smart-contract fuzzing [9], and ItyFuzz explores stateful execution behavior to expose vulnerabilities that emerge only under specific transaction histories [10]. These techniques have substantially improved the analysis of existing smart contracts. Nevertheless, most of them assume that the contract code has already been written and focus on finding defects in a fixed implementation. They do not directly address a rapidly emerging development scenario in which executable Solidity programs are generated automatically from natural-language requirements. Large language models have made this scenario increasingly practical. A user can now describe a token system, auction mechanism, deposit rule, ownership policy, or payment process in natural language and obtain compilable Solidity code within a short period of time. Early studies examined the feasibility of generating smart contracts with general-purpose conversational language models [11,12]. More recent agent-based software-engineering systems distribute different responsibilities among specialized components for requirement interpretation, code generation, testing, static analysis, execution, and repair [13,14]. Such architectures are attractive for blockchain development because smart-contract correctness cannot be determined solely from surface-level code quality. A generated contract may compile successfully while still violating an access-control rule, allowing an invalid state transition, emitting an incorrect event, or producing an unexpected balance change [15,16]. Compilation success therefore represents only the lowest level of correctness. Evaluation methods designed specifically for generated Solidity code are still developing. SolEval introduced repository-level tasks and evaluated generated contracts using functional correctness, vulnerability occurrence, and gas-related measures [17]. Other work has compared smart-contract code generated from natural-language descriptions with implementations derived from more formal specifications [18]. These studies indicate that requirement representation strongly affects the quality of generated code. Natural-language requirements may omit implicit constraints, boundary conditions, actor permissions, transaction-order assumptions, or failure behavior that an experienced developer would normally infer. A generation model may therefore produce syntactically valid code that appears reasonable but does not faithfully implement the intended business process. This problem becomes more serious in stateful contracts because an error in one transaction can modify persistent storage and influence many later transactions. Property-based testing provides a useful mechanism for addressing this problem. Instead of checking only a small number of manually written test cases, property-based approaches repeatedly generate inputs and transaction combinations to determine whether predefined behavioral properties continue to hold [19]. Relevant properties may include conservation of balances, ownership restrictions, withdrawal limits, bidding rules, authorization requirements, expected event emission, and restrictions on invalid state transitions. Such testing is particularly suitable for generated smart contracts because the goal is not merely to determine whether one example succeeds, but to establish whether the implementation consistently respects the underlying rules across many possible inputs and execution sequences. However, property-based testing is more effective when the original natural-language requirement has already been transformed into explicit roles, states, permissions, preconditions, postconditions, and failure cases. Without such formalization, the testing process may reproduce the same ambiguity contained in the original prompt. Execution cost introduces an additional challenge. Every computational and storage operation performed on a blockchain consumes gas, and inefficient implementations can make otherwise correct contracts unnecessarily expensive to deploy or use. Storage writes, repeated state access, unnecessary loops, redundant calculations, poorly selected data structures, and avoidable memory operations can substantially increase transaction fees. Previous research has identified recurring gas-related code smells and optimization opportunities in Solidity programs [20]. Other studies have generated high-cost inputs to reveal expensive execution paths [21] and applied mutation-based strategies to evaluate gas consumption under alternative implementations [22]. Surveys have also summarized common optimization techniques for reducing smart-contract execution cost [23], while multi-agent approaches have recently been investigated for automated gas optimization [24]. These studies demonstrate that gas efficiency is an important quality dimension,

particularly for contracts that execute frequently or process large numbers of transactions. Yet optimization introduces its own risk: a change that reduces gas consumption may also alter balances, authorization checks, emitted events, revert conditions, or state-transition logic. Gas reduction should therefore be treated as a constrained optimization problem rather than an independent code-compression task. Current research consequently leaves an important gap between smart-contract generation and smart-contract assurance. Existing datasets frequently emphasize isolated functions, previously reported vulnerable contracts, or relatively small collections of repositories. Such benchmarks are valuable for vulnerability detection, but they may not adequately represent contract-level business logic involving multiple actors, persistent states, dependent transactions, and failure conditions. At the same time, code generation, vulnerability detection, property testing, and gas optimization are commonly evaluated as separate tasks. This separation makes it difficult to determine whether improvements at one stage actually produce a safer and more useful contract. A model may generate compilable code but fail behavioral tests; a security analyzer may report no known vulnerability while missing a business-rule violation; and a gas optimizer may reduce execution cost while unintentionally modifying contract semantics. An effective evaluation framework therefore needs to examine these dimensions together and preserve traceability from the original requirement to the final optimized implementation.

This study proposes ContractAgent-V, a multi-agent framework that integrates requirement formalization, Solidity code generation, vulnerability inspection, property-based testing, iterative repair, and gas optimization within a unified workflow. Rather than sending an unstructured user request directly to a code generator, the framework converts natural-language requirements into explicit representations of participant roles, state variables, transaction conditions, access-control rules, expected state transitions, and failure cases. These structured constraints are then used to guide contract generation and to construct behavioral properties for subsequent testing. Generated contracts are subjected to vulnerability inspection and stateful property-based tests, and detected failures are returned to the relevant agents for revision before optimization is attempted. Gas optimization is performed only after functional and security requirements have been established, and the optimized contract is revalidated to determine whether cost reductions preserve its original semantics. The experimental study uses 1,120 tasks across eight smart-contract categories and evaluates deployment success, functional test performance, high-severity vulnerabilities, access-control violations, and gas consumption. By examining these indicators jointly, the study aims to determine whether agent-based generation can move beyond producing merely compilable Solidity code toward generating contracts that are behaviorally correct, security-aware, and economically efficient. The broader significance of the work lies in establishing an end-to-end evaluation paradigm for natural-language-driven smart-contract development, where requirement interpretation, implementation, verification, and optimization are treated as interdependent stages rather than isolated objectives. Such a framework can provide stronger evidence for the practical reliability of automatically generated contracts and support the safer adoption of large language models in blockchain software engineering.

MATERIALS AND METHODS

Study Sample and Task Description

This was a computational study; therefore, no geographic study area was defined. The sample included 1,120 smart contract tasks. The tasks covered token issuance, escrow, crowdfunding, voting, staking, auction, access-control, and decentralized marketplace functions. Each task was described by a natural-language requirement and included contract roles, state variables, transaction rules, permission limits, and failure conditions. The evaluation included 6,720 generated contracts, 43,500 property-based tests, and 18,200 vulnerability probes.

Experimental Design and Control System

ContractAgent-V was used as the experimental system. It included a requirement formalization agent, a Solidity generation agent, a vulnerability inspection agent, a property-based testing agent, and a gas optimization agent. The control system was a single-agent Solidity generator. Both systems received the same task requirements. The experimental group used security inspection, transaction testing, and gas optimization after code generation. The control group generated Solidity code without this multi-agent verification process. Deployment success, functional test results, vulnerability findings, access-control violations, and gas use were compared between the two systems.

Measurement and Quality Control

Deployment success was recorded when a generated contract could be deployed successfully. Functional correctness was measured using property-based tests. These tests used randomized transaction sequences to check balance consistency, ownership limits, state changes, and revert behavior. Security measurement included

high-severity vulnerability findings and access-control violations. The vulnerability inspection agent checked the generated code against common weakness patterns and symbolic execution warnings. Gas use was measured for core contract transactions before and after optimization. Generated contracts, test cases, vulnerability findings, and execution results were recorded for each task. Failed cases were checked again with the same contract version and test condition.

Data Processing and Model Formulae

Natural-language requirements were organized into contract roles, state variables, transaction rules, access conditions, and expected failure cases. Generated Solidity code, test outputs, vulnerability findings, and gas records were then grouped by task. The functional test pass rate was calculated as:

$$FTR=\frac{\sum_{i=1}^{N} n_i^{pass}}{\sum_{i=1}^{N} n_i^{test}}\times 100\%$$

The mean gas reduction after optimization was calculated as:

$$GR=\frac{1}{N}\sum_{i=1}^{N}\left(\frac{g_i^{before}-g_i^{after}}{g_i^{before}}\right)\times 100\% =\alpha E_i+\beta D_i+(1-\alpha-\beta)G_i$$

The first equation gives the proportion of passed functional tests. The second equation gives the mean percentage of gas saved after optimization.

Data Reporting and Reproducibility

All results were summarized across the 1,120 tasks. Deployment success, functional test pass rate, high-severity vulnerability rate, and access-control violations were reported as percentages. Gas use was compared before and after optimization. The same task set was used for ContractAgent-V and the single-agent control system. The original task requirements, generated contracts, test cases, audit findings, and execution records were retained for each task so that the reported results could be checked again.

RESULTS AND DISCUSSION

Deployment and Functional Performance

ContractAgent-V achieved a deployment success rate of 84.7%, while the single-agent generator achieved 69.3%. The difference was 15.4 percentage points. The functional test pass rate increased from 61.8% to 79.6%, a gain of 17.8 percentage points. Higher deployment success was therefore accompanied by better contract behavior in the functional tests. Earlier studies showed that language models can generate Solidity code with valid syntax but may fail to meet contract rules or handle links between functions correctly [25,26]. The results show that requirement checking and repeated testing reduced these problems. Fig. 1 presents a related comparison of language-model-assisted verification methods.

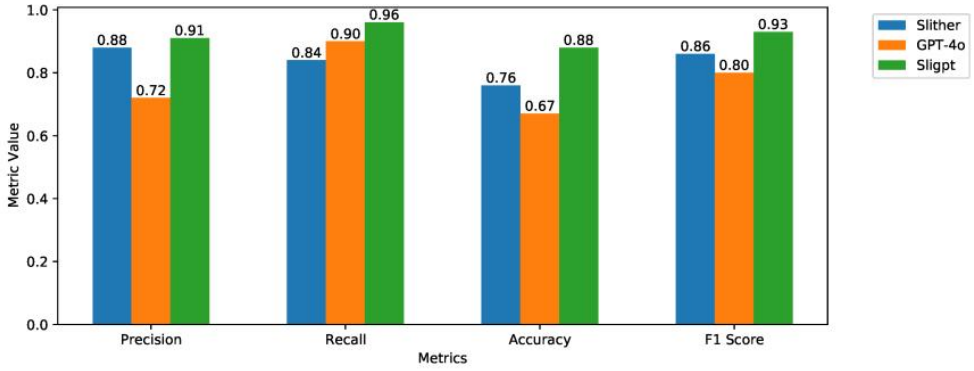

Figure 1. Comparison Of Slither, Gpt-4o, and Sligpt In Solidity Data-Dependency Analysis

Security Results

The rate of high-severity vulnerabilities decreased from 15.2% to 4.9%. This was a decrease of 10.3 percentage points, or 67.8% relative to the baseline. Access-control violations also decreased by 58.1%. These errors are important because an unauthorized user may change contract data or transfer assets. Earlier tools, including SmarTest, SMARTIAN, and ItyFuzz, mainly test contracts that already exist [27,28]. ContractAgent-V checks security during code generation and tests transaction sequences before selecting the final contract version. The lower vulnerability rate shows the value of combining requirement checks, security inspection, and transaction testing.

Effect of the Multi-Agent Design

The results improved across deployment, functional testing, and security testing. The experiment compared the full ContractAgent-V system with a single-agent generator. It therefore measured the effect of the complete workflow rather than the effect of each agent alone. Previous multi-agent code systems used testing, static analysis, and repair for general programming tasks [29,30]. This study applied these steps to Solidity contracts, where errors may affect balances, permissions, events, and contract states. The benchmark included eight contract types, which reduced the effect of one task category on the results. However, it did not include every type of contract dependency or on-chain condition. Larger contract systems and external protocol calls should be tested in later work.

Gas Optimization and Practical Meaning

Gas use in core contract functions decreased by 23.4% after optimization. This decrease occurred with higher functional test results and fewer security findings. The gas-saving changes therefore passed the tests used in this study. GasAgent reported a mean deployment-gas saving of 9.97% for 100 verified contracts. The two values should not be treated as a direct comparison. ContractAgent-V measured gas use for core transaction functions, whereas GasAgent measured deployment gas. Earlier studies identified storage operations, repeated calculations, and costly execution paths as common causes of high gas use. Fig. 2 shows the distribution of gas savings reported for real-world contracts. Future work should test ContractAgent-V on larger deployed contracts under different network and contract-state conditions.

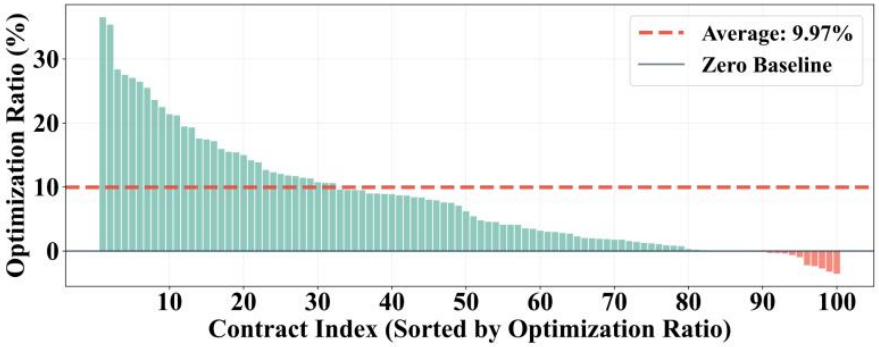

Figure 2. Gas Savings Across 100 Real-World Smart Contracts

CONCLUSION

ContractAgent-V improved the quality of generated Solidity contracts by combining requirement analysis, code generation, security checks, transaction tests, and gas optimization. Compared with the single-agent generator, deployment success increased from 69.3% to 84.7%, and the functional test pass rate increased from 61.8% to 79.6%. High-severity vulnerabilities decreased from 15.2% to 4.9%, access-control violations decreased by 58.1%, and gas use in core functions decreased by 23.4%. These results show that generated contract code should be checked for function rules, security risks, and gas cost before deployment. The system can support the development of token, escrow, voting, staking, and marketplace contracts. The benchmark covered eight contract types and controlled test conditions. It did not measure the effect of each agent separately or include all links with external deployed contracts. Future studies should test larger contracts under changing blockchain states and real network conditions.

REFERENCES

Chen, F., Li, S., Liang, H., Xu, P., & Yue, L. (2025). Optimization Study of Thermal Management of Domestic SiC Power Semiconductor Based on Improved Genetic Algorithm.

Lehmann, N., Kurashige, C., Akiti, N., Krishnakumar, N., & Jhala, R. (2025). Generic Refinement Types. Proceedings of the ACM on Programming Languages, 9(POPL), 1446-1474.

Jiao, Y., Zhao, B., Wang, A., & Shi, T. (2026). Construction and Empirical Study of a Modularized Teaching System for Art Courses Based on a Unified Training Pathway.

Yao, G., Zheng, J., Wang, Z., Zhang, W., Han, R., Zhao, C., ... & Liu, R. (2026, March). V-pruner: A fast and globally-informed token pruning framework for vision transformer. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 40, No. 40, pp. 34396-34404).

Lyu, M. R., Ray, B., Roychoudhury, A., Tan, S. H., & Thongtanunam, P. (2025). Automatic programming: Large language models and beyond. ACM Transactions on Software Engineering and Methodology, 34(5), 1-33.

Xiong, W., Guo, Y., & Zeng, Y. (2026). A Study on the Energy Efficiency of MEP Systems and Coordinated Dispatch Mechanisms for Multi-energy Systems in High-density Urban Buildings. Available at SSRN 7120518.

Hong, Z., Liang, S., Xu, T., & Chen, H. (2026). A Study on Failover Verification and Recovery Objective Prediction for Cross-Region Cloud Services.

Yuan, Y., Xu, T., Yin, J., & Huang, J. (2026). Modeling Loading Anomalies and Identifying Root Causes in Media Consumption Workflows on Large Social Media Platforms. Available at SSRN 7393318.

Almaghthawi, A., & Yafooz, W. M. (2025, February). Comparative analysis of large language models in solidity smart contract vulnerability detection. In 2025 2nd International Conference on Advanced Innovations in Smart Cities (ICAISC) (pp. 1-5). IEEE.

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.

Jiao, Y., Shi, T., Zhao, B., & Wang, A. (2026). Retrieval-Guided Structured Reasoning and Interpretable Representation Learning for Large-Scale Video–Language Models.

Liang, S., Du, Y., Chen, W., & Liu, Z. (2026). Order Allocation and Emergency Transportation Decisions Amid Multi-Source Procurement Disruptions.

Akbar, M. A., Khan, A. A., Hamza, M., Ghaffar, A., & Hajikhani, A. (2025). Agentic AI in software engineering: Practitioner perspectives across the software development life cycle. Software Engineering: Practitioner Perspectives Across the Software Development Life Cycle (September 16, 2025).

Liang, S., Zhang, X., Du, Y., & Chen, W. (2026). Supply Network Restructuring and Capacity Ramp-Up Limits in the Localized Expansion of High-Performance Computing Equipment Production. Available at SSRN 7339738.

Bao, Y., Qiu, Y., & Wang, H. (2026). FLARE: A Real-Time Framework for Detecting Malicious Account Activities at Internet Scale. Available at SSRN 7185659.

Li, J., Ma, R., & Gu, Y. (2026). From Audit Requirements to Computable Software Architecture: A Rule-Engine and Evidence-Indexing Method for Trusted Digital Infrastructure.

Hensel, F., Banerjee, A., Ebrahimi, E., & Schulte, S. (2025, October). An AI-powered Auto-Completion Tool for Solidity Smart Contracts. In 2025 IEEE International Conference on Blockchain (Blockchain) (pp. 171-180). IEEE.

Huang, J., Yin, J., Xu, T., & Yang, J. (2026). Quality Assessment and User Adoption Prediction for Enterprise-Level Financial Reporting Data Products. Available at SSRN 7179118.

Zhang, Z., Wang, J., Li, Z., Wang, Y., & Zheng, J. (2025). Anncoder: A mti-agent-based code generation and optimization model. Symmetry, 17(7), 1087.

Salzano, F., Marchesi, L., Antenucci, C. K., Scalabrino, S., Tonelli, R., Oliveto, R., & Pareschi, R. (2026). Bridging the gap: a comparative study of academic and developer approaches to smart contract vulnerabilities. Empirical Software Engineering, 31(2), 37.

You, S. (2026). Verifiable Audit Mechanisms in AI Compliance Automation: Scalability. Available at SSRN 6547458.

Bao, Y., & Wang, H. (2026). Orion: A High-Throughput, Fault-Tolerant Event Routing Architecture for Hyperscale Data Streams. Fault-Tolerant Event Routing Architecture for Hyperscale Data Streams (January 01, 2026).

Riyadi, M. A. Q., Al Mujahidin, S. M., Rustan, N. A., Aulia, A. I. B. F., Ningtyas, M. A., & Amiroh, K. (2026). Experimental Evaluation of Smart Contract Gas Optimization Techniques: A Case Study of a Blockchain-Based Reporting Application System. ITEGAM-JETIA, 12(60), 615-628.

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.

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.

Chatterjee, S., & Ramamurthy, B. (2025, March). Efficacy of various large language models in generating smart contracts. In Future of Information and Communication Conference (pp. 482-500). Cham: Springer Nature Switzerland.

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.

Qi, C., & Qiao, X. (2026). Building and Operating a Large Scale Multi-Agent System: A Case Study from Industry. Available at SSRN 6795198.

Kumar, B., & Manas, T. L. (2026). Agentcodereview: A Multi-Agent Framework For Explainable Code Review And Automated Bug Repair. International Journal of Computer Information Systems and Industrial Management Applications, 18(14s), 1121-1129.

Du, Y., Liu, Q., Dong, Y., & Chen, Z. (2026). Cross-Domain Transfer and Few-Shot Recognition of Defect Images in Complex Industrial Settings.

ETHICAL DECLARATION

Conflict of interest: No declaration required. **Financing:** No reporting required. **Peer review:** Double anonymous peer review.