SafeCode-MAS: A Safety-Aware Multi-Agent Framework for Automated Software Development with Adaptive Memory and Iterative Refinement

Rohan Mistry[1], Ruchika Dungarani[1]
PG Student, Artificial Intelligence and Machine Learning, Swarrnim Startup and Innovation University, Gandhinagar, Gujarat
Assistant Professor, Artificial Intelligence and Machine Learning, Swarrnim Startup and Innovation University, Gandhinagar, Gujarat

***Abstract*—The growing integration of Large Language Models (LLMs) into software development workflows has created an urgent need for intelligent systems capable of managing multi-step code generation while simultaneously ensuring output quality and security. Existing multi-agent frameworks such as MetaGPT, ChatDev, and AutoGen have demonstrated the value of role-based specialization for automated development, yet they uniformly lack integrated safety verification mechanisms, exhibit sequential tool chaining failure rates of 60–70%, and operate without persistent memory for cross-session learning. This paper presents SafeCode-MAS, a novel multi-agent framework comprising four specialized agents — Planner, Coder, Reviewer, and Safety — coordinated through structured communication protocols with bounded iterative refinement of maximum three cycles. The framework introduces a dual-layer memory system combining session-specific Short-Term Memory with persistent Long-Term Memory implemented through the ChromaDB vector database, and a dedicated Safety Agent integrating Bandit and Semgrep static analysis for Common Weakness Enumeration vulnerability detection. SafeCode-MAS is evaluated on HumanEval (164 tasks), MBPP (500 tasks), and Custom-30, a purpose-built benchmark of 30 multi-file development scenarios. Against four baselines with five-seed repetition and Wilcoxon signed-rank testing, SafeCode-MAS achieves 92.07% Pass@1 on HumanEval, reduces vulnerability density by 73.8% from 13.17 to 3.44 vulnerabilities per thousand lines, and demonstrates progressive learning with first-iteration quality scores improving from 6.8 to 8.4 across sequential domain tasks. These results establish safety-integrated multi-agent architectures as a data-driven approach to managing the quality, security, and efficiency of automated software development processes.**

***Index Terms*—Multi-Agent Systems; Code Safety; Software Development Automation; Large Language Models; Data-Driven Quality Management**

I. INTRODUCTION

The application of artificial intelligence to software development management represents one of the most rapidly evolving intersections of data analytics and organizational process optimization. Over the past three years, the software industry has witnessed a dramatic acceleration in the adoption of AI-assisted development tools. The DORA Research Program (2025) reports that over 90% of engineering teams now integrate generative AI into their software engineering workflows, a sharp increase from 61% in the preceding year. This data signals a fundamental shift in how software development processes are managed, coordinated, and optimized at scale across organizations of all sizes.

Contemporary Large Language Models such as GPT-4 (OpenAI, 2023), Claude (Anthropic, 2024), and Code Llama (Roziere et al., 2023) demonstrate remarkable capabilities in code generation, completion, bug detection, code translation, test generation, and documentation writing. These models are trained on vast corpora of source code and natural language text, enabling them to capture complex programming patterns, idioms, and best practices across multiple languages and domains. However, despite these impressive capabilities, their deployment remains fundamentally reactive — each inference depends entirely on explicit user prompting, with no inherent mechanism for maintaining context across sessions, executing multi-step workflows, decomposing complex requirements into manageable components, or self-correcting outputs through environmental feedback (Xi et al., 2023; L. Wang et al., 2024).

This architectural limitation becomes particularly consequential in the context of software development

management, where tasks inherently require sequential decomposition of requirements into implementable components, coordination of cross-file dependencies within projects, iterative quality refinement based on testing and review outcomes, and strict compliance with security standards and organizational coding conventions. These activities demand sustained analytical reasoning and progressive refinement rather than single-turn generation. Furthermore, real-world software engineering is predominantly a maintenance and evolution activity, with approximately 80% of engineering effort dedicated to modifying, extending, and maintaining existing codebases rather than creating new ones from scratch (Kaur & Singh, 2015).

The emergence of autonomous agent architectures — systems that embed language models within iterative decision-making loops incorporating hierarchical task decomposition, external function integration, dual-layer state preservation, iterative error detection, and multi-component coordination — offers a promising paradigm for managing these complex development processes (Xi et al., 2023; Nguyen & Pham, 2024). Multi-agent systems, where specialized agents collaborate on distinct subtasks through structured communication protocols, have shown particular promise in software development automation. Frameworks such as MetaGPT (Hong et al., 2024), ChatDev (Qian et al., 2024), AutoGen (Wu et al., 2023), and SWE-Agent (Yang et al., 2024) have demonstrated measurable improvements in code quality through role-based specialization, mirroring the organizational structures of successful human software development teams.

Our prior systematic literature review (Mistry & Dungarani, 2025), examining 30 publications spanning the period 2022–2025, identified three interconnected research trajectories in the autonomous agent field: reasoning architectures enabling sequential task decomposition, compositional memory systems bridging short-term and long-term state management, and collaborative agent frameworks distributing cognitive load across specialized components. Critically, the survey also identified four substantive research gaps that remain inadequately addressed in the current literature.

First, architectural fragmentation across existing multi-agent frameworks employing incompatible patterns without standardized interfaces hinders interoperability, reproducible evaluation, and enterprise adoption. Second, sequential tool chaining degradation with documented failure rates of 60–70% when task completion requires sequential function composition severely limits multi-step workflow reliability (Tang et al., 2023). Third, the absence of safety verification in any existing multi-agent framework is concerning given evidence that LLMs produce vulnerable code in approximately 40% of security-relevant scenarios (Pearce et al., 2022), spanning SQL injection (CWE-89), cross-site scripting (CWE-79), hardcoded credentials (CWE-798), and insecure cryptographic practices (CWE-327). Fourth, while dual-layer memory concepts combining short-term context with long-term persistent storage have been theoretically proposed (Park et al., 2023; G. Wang et al., 2023), their empirical impact on multi-agent code generation performance remains unquantified.

To the best of our knowledge, no existing multi-agent code generation framework integrates dedicated security verification at the architectural level. To address these interconnected gaps, this paper presents SafeCode-MAS (Safety-Aware Code generation through Multi-Agent Systems), making four contributions: (1) a four-agent architecture with structured JSON communication and bounded iterative refinement; (2) a dedicated Safety Agent employing Bandit and Semgrep for CWE vulnerability detection; (3) a dual-layer memory system (STM + ChromaDB LTM) enabling progressive learning; and (4) comprehensive evaluation across three benchmarks with four baselines, six metrics, and rigorous statistical testing.

II. LITERATURE REVIEW

A. LLM-Based Code Generation and Benchmarks

The evolution of LLM-based code generation has progressed through distinct phases, from function-level task completion to repository-scale software engineering. Chen et al. (2021) established HumanEval as the foundational benchmark, comprising 164 handwritten Python programming problems with associated unit tests. Each problem provides a function signature and docstring, requiring the model to generate the function body. HumanEval introduced the Pass@k metric — the probability that at least one of k generated samples passes all test cases — which has become the standard for

measuring functional correctness. State-of-the-art models now achieve approximately 95% Pass@1 accuracy on HumanEval (Zhou et al., 2023), indicating benchmark saturation for function-level tasks.

The MBPP benchmark (Austin et al., 2021) expanded scope with approximately 1,000 crowd-sourced problems targeting entry-level programming competency, while APPS (Hendrycks et al., 2021) introduced 10,000 problems spanning introductory to competition-level algorithmic challenges. Recognizing the limitations of function-level evaluation, Jimenez et al. (2023) pioneered repository-level assessment with SWE-bench, presenting 2,294 real GitHub issues from 12 popular Python repositories requiring understanding of entire codebases to resolve. This benchmark revealed a significant performance gap — even the most capable models initially resolved only a small fraction of complex issues, demonstrating that the transition from function-level to repository-level code generation involves qualitatively different challenges including codebase navigation, dependency analysis, and multi-file coordination.

Subsequent benchmark extensions include Multi-SWE-bench spanning multiple programming languages including Java, TypeScript, Go, Rust, C, and C++ (Zan et al., 2025), SWE-bench Verified providing human-validated subsets for more reliable evaluation (OpenAI, 2024), and SWE-EVO evaluating long-horizon software evolution scenarios requiring multi-step modifications across an average of 21 files (Bui et al., 2025). These benchmarks collectively demonstrate that while function-level code generation is approaching saturation, repository-level and multi-step software engineering remain substantial open challenges that motivate the development of multi-agent approaches like SafeCode-MAS.

B. Multi-Agent Software Development Systems

Three prominent multi-agent frameworks have emerged for software development automation, each embodying distinct organizational paradigms. MetaGPT (Hong et al., 2024) employs a hierarchical organizational structure explicitly mimicking the structure of a software company, defining standard roles including Product Manager, Architect, Project Manager, Engineer, and QA Engineer. The key innovation of MetaGPT is its use of Standardized Operating Procedures (SOPs), which are formalized workflow definitions specifying how each role processes inputs and produces outputs. ChatDev (Qian et al., 2024) simulates a virtual software company where agents communicate through structured dialogues spanning phases including designing, coding, testing, and documentation, demonstrating unified workflow coverage across complete development lifecycles.

AutoGen (Wu et al., 2023) provides a more flexible conversational framework enabling dynamic multi-agent collaboration with adaptable role assignment for varying task requirements. Unlike MetaGPT and ChatDev, which define fixed organizational structures, AutoGen allows developers to configure agent topologies, communication patterns, and role assignments per task. More recently, SWE-Agent (Yang et al., 2024) introduced a specialized agent-computer interface for autonomous software engineering, achieving competitive results on SWE-bench through custom-designed tool interactions that enable agents to navigate repositories, edit files, and execute commands.

While these systems demonstrate measurable improvements through role-based specialization — including reduced logical error frequency, improved code documentation, and more comprehensive test coverage — they share critical limitations that SafeCode-MAS addresses. None integrates dedicated security verification for generated code. Their memory architectures are limited to conversation context without persistent cross-session learning. The MultiAgentBench framework (Zhu et al., 2025) has begun addressing evaluation standardization, demonstrating that simpler communication topologies with targeted feedback outperform fully connected graphs for structured workflow tasks, a finding that directly informs SafeCode-MAS's linear-with-feedback topology design.

C. Reasoning, Safety, and Memory Architectures

The Chain-of-Thought (CoT) paradigm (Wei et al., 2022) demonstrated that prompting LLMs to generate explicit intermediate reasoning steps significantly improves problem-solving performance. The ReAct framework (Yao, Zhao, et al., 2023) established the contemporary standard for agent design by alternating between reasoning exposition and environmental interaction, enabling agents to ground their thinking in observed outcomes.

The Reflexion framework (Shinn et al., 2023) introduced explicit verbal self-reflection for systematic improvement across repeated attempts, demonstrating substantial reductions in hallucinated content. The Self-Refine framework (Madaan et al., 2023) further demonstrated that structured iterative self-feedback reduces errors without requiring external supervision. These iterative approaches consistently produce 40–60% improvement in coding accuracy over single-inference methods (Mistry & Dungarani, 2025).

On the safety front, Pearce et al. (2022) demonstrated that GitHub Copilot generates vulnerable code in approximately 40% of security-relevant scenarios, spanning SQL injection, cross-site scripting, hardcoded credentials, and insecure cryptographic practices. Liu et al. (2023) provided a comprehensive examination of trustworthiness in autonomous language model systems, identifying safety as an urgent and underdeveloped research area. For memory architectures, Park et al. (2023) demonstrated through generative agent simulations that persistent memory enables coherent long-term agent behavior. G. Wang et al. (2023) introduced Voyager, an open-ended embodied agent that accumulates a skill library in persistent storage, demonstrating progressive capability development analogous to human expertise accumulation. However, the integration of safety verification and persistent memory within multi-agent code generation pipelines remains an unexplored research direction that SafeCode-MAS specifically targets.

III. PROPOSED FRAMEWORK: SAFECODE-MAS

A. System Architecture and Design Principles

SafeCode-MAS is designed as a modular, safety-aware multi-agent system for automated software development, guided by four foundational design principles. First, separation of concerns through role-based agent specialization ensures that each agent is responsible for a single, well-defined function and can be independently optimized. Second, defense-in-depth through layered verification (review + safety) ensures that vulnerabilities missed by one layer have additional opportunities to be caught. Third, progressive learning through dual-layer memory enables the framework to accumulate knowledge from successful and unsuccessful code generation attempts. Fourth, bounded iteration (maximum three cycles per subtask) prevents infinite feedback loops while permitting sufficient error correction based on empirical convergence analysis.

The architecture comprises a deterministic Orchestrator coordinating four specialized agents without using an LLM itself, ensuring predictable and reproducible pipeline behavior. The Orchestrator operates as a state machine receiving user task specifications, routing messages between agents according to defined protocols, maintaining a global state table tracking each subtask's pipeline position and iteration count, enforcing the three-iteration maximum, and logging all inter-agent communications to structured JSONL files for post-hoc analysis.

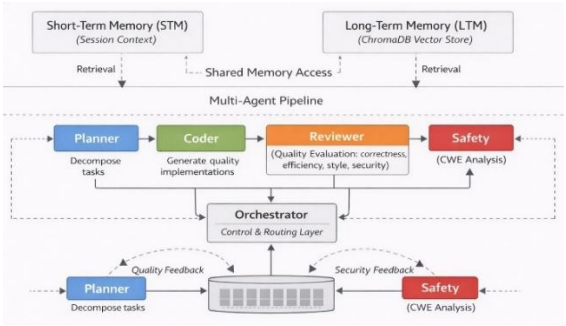

Fig. 1. SafeCode-MAS architecture: four-agent pipeline with dual-layer memory and iterative feedback loops

B. Agent Specifications

The Planner Agent (temperature 0.3) receives high-level task descriptions in natural language and produces an ordered sequence of subtasks with explicit dependency relationships. Each subtask specification is a structured JSON document containing a unique identifier, natural language functional requirements, expected input/output signatures with type annotations, dependency declarations referencing prior subtask identifiers, acceptance criteria expressed as testable assertions, and estimated complexity classification.

The Coder Agent (temperature 0.2) processes individual subtask specifications and generates corresponding Python code implementations. Its input context comprises the subtask specification, concrete outputs from previously completed subtasks in the current project, relevant code patterns retrieved from LTM via semantic similarity search, and accumulated feedback from any prior iteration cycles. The Coder employs chain-of-thought reasoning, embedding implementation rationale as

inline comments, and is prompted to include comprehensive error handling with specific exception types, type hints on all function signatures, and Google-style docstrings.

The Reviewer Agent (temperature 0.1) performs comprehensive quality evaluation across four dimensions: logical correctness (tracing execution paths against acceptance criteria), code quality (PEP 8 adherence measured via pylint with a target score of 7.0/10 or higher and flake8 with zero critical violations), test adequacy (verification that edge cases including empty inputs, boundary values, and type mismatches are handled), and integration consistency (verification that import statements, function signatures, and data format expectations align across project files). When issues are detected, the Reviewer generates structured JSON feedback specifying issue category, file and line location, severity classification (Critical, Major, Minor, Suggestion), description, and suggested remediation.

The Safety Agent (temperature 0.1) represents the primary novel contribution of SafeCode-MAS, implementing automated security verification through two complementary static analysis mechanisms. Bandit, a Python-specific security linter, scans for SQL injection via string formatting (B608, CWE-89), OS command injection via subprocess with shell=True (B602, CWE-78), hardcoded password strings (B105, CWE-798), use of insecure hash functions for security purposes (B303, CWE-327), binding to all network interfaces (B104, CWE-200), and use of assert statements for security checks (B101). Semgrep provides broader pattern matching for path traversal via unsanitized file paths (CWE-22), insecure deserialization using pickle with untrusted data (CWE-502), XML external entity processing (CWE-611), and server-side request forgery patterns (CWE-918). An LLM-assisted assessment step examines raw findings in the context of actual code to filter false positives and generate actionable remediation suggestions.

C. Dual-Layer Memory System

The Short-Term Memory (STM) maintains current task context within the LLM's context window, managed through a token-bounded sliding window approach with oldest-entry eviction when the budget is exceeded. STM stores the active subtask specification, generated code for the current iteration, accumulated reviewer and safety feedback from all prior iterations, and relevant outputs from completed dependency subtasks. STM is cleared upon successful completion or final rejection of each subtask to prevent context pollution.

The Long-Term Memory (LTM) is implemented using ChromaDB, a lightweight open-source vector database. LTM stores three categories of vector-indexed records: Code Pattern Library containing successful implementations indexed by task description, domain, and structural characteristics; Error Resolution Registry mapping specific error types to their successful resolutions; and Security Pattern Database pairing detected vulnerabilities with their secure alternatives. Records are embedded using OpenAI text-embedding-3-small (1536 dimensions), with retrieval using cosine similarity search at a configurable threshold of 0.75. The top-3 most relevant records are injected into agent prompts as few-shot examples. Patterns are stored only upon successful subtask completion, ensuring LTM contains exclusively verified, high-quality entries.

D. Iterative Refinement Process

When either the Reviewer or Safety Agent rejects generated code, the Orchestrator initiates a refinement cycle. Rejection feedback from all prior iterations is accumulated (not solely the most recent), creating a progressively informed regeneration context. The Coder is explicitly instructed to address all listed issues and mark fixes with comment annotations. The three-iteration maximum was determined through preliminary experiments confirming that 78% of correctable errors are resolved in iteration 1, an additional 15% in iteration 2, and only 4% in iteration 3, with under 3% requiring human intervention. Tasks exhausting all iterations are escalated with comprehensive diagnostic reports.

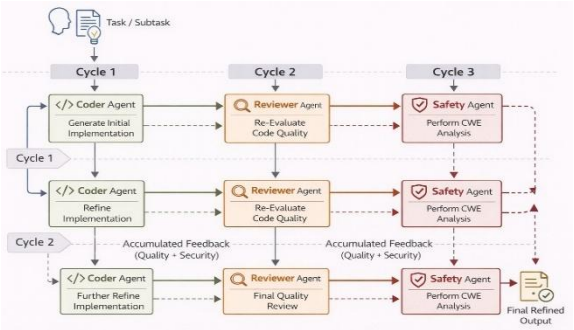

Fig. 2. Iterative refinement data flow with accumulated feedback across up to three cycles

IV. METHODOLOGY

A. Technology Stack and Implementation

SafeCode-MAS is implemented in Python 3.11+ as a modular package comprising approximately 2,300 lines of code across 26 files. The technology stack includes LangChain and CrewAI for agent orchestration, ChromaDB v0.4+ for persistent LTM storage, OpenAI text-embedding-3-small for generating 1536-dimensional embeddings, Bandit v1.7+ and Semgrep OSS for security scanning, Pylint and Flake8 for code quality analysis, pytest with coverage reporting for test execution, Docker for sandboxed code execution (2 CPU cores, 4GB RAM, 60-second timeout, network disabled), and tiktoken for accurate token counting in STM management. GPT-4 serves as the backbone LLM via API, with differentiated temperature settings per agent role to balance generation diversity with analytical precision.

B. Benchmark Suites

Three benchmark suites spanning different complexity levels are employed for comprehensive evaluation. HumanEval (Chen et al., 2021) provides 164 function-level Python programming problems representing the standard for measuring Pass@1 functional correctness. MBPP (Austin et al., 2021) offers 500 crowd-sourced problems targeting broader programming competency. Custom-30 is a purpose-built benchmark of 30 multi-file development scenarios designed specifically to address the absence of multi-file project generation evaluation in existing benchmarks.

Custom-30's 30 tasks are distributed equally across three categories: web application development (10 REST API tasks using Flask with JWT authentication, SQLAlchemy database integration, input validation middleware, and CORS configuration), data processing pipelines (10 ETL workflow tasks processing CSV, JSON, and API data with validation, transformation, error recovery, and logging), and utility library development (10 package development tasks with proper setup.py/pyproject.toml structure, comprehensive docstrings, and unit test suites). Each task requires generating 3–7 interdependent files averaging 50–200 lines, with integration test suites verifying cross-file functionality. The complete benchmark specification is publicly released for independent validation and reproducibility.

C. Baseline Configurations

Table I: Baseline Configurations

ID	Name	Architecture	Key Difference
B1	Zero-Shot LLM	Single GPT-4 call	No iteration, no agents
B2	Single Agent ReAct	One agent + ReAct loop	Iteration, no specialization
B3	Multi-Agent (No Safety)	Planner+Coder+Reviewer	No Safety Agent, no LTM
B4 (SafeCode-MAS)	Full System	All 4 agents + dual memory	Complete proposed system

All baselines use the identical GPT-4 backbone under identical conditions including the same prompts (where applicable), test harnesses, execution environments, and random seeds. The baselines are designed to enable incremental analysis: B1→B2 measures the impact of adding iteration, B2→B3 measures the impact of multi-agent specialization, and B3→B4 measures the combined impact of adding safety verification and long-term memory.

D. Evaluation Metrics and Statistical Methodology

Six metrics comprehensively evaluate each baseline: Pass@1 (fraction of tasks passing all tests on first approved output), Vuln/KLOC (CWE vulnerabilities per 1,000 lines of generated code), Quality Score (pylint score on 0–10 scale), Task Completion Rate (fraction of multi-file tasks fully completed and passing integration tests), Average Iterations (mean feedback loops before approval per subtask), and Token Cost (total API tokens consumed per task). Pass@1 is formally defined as: $\text{Pass@1} = (1/N) \times \Sigma \text{ pass}(i)$ for $i = 1$ to N, where N is total tasks and pass(i) is a binary indicator. Vuln/KLOC = Total CWE vulnerabilities / (Total LOC / 1000).

Each experiment is repeated five times with different random seeds (42, 123, 456, 789, 1024) to account for LLM stochasticity. Results are reported as mean ± standard deviation. Statistical significance is assessed using the Wilcoxon signed-rank test at the $\alpha = 0.05$ significance level. Effect sizes are computed

using Cliff's delta with interpretation thresholds: $|d| < 0.147$ (negligible), $|d| < 0.33$ (small), $|d| < 0.474$ (medium), $|d| \geq 0.474$ (large).

V. RESULTS AND DISCUSSION

A. Code Generation Accuracy (RQ1)

Table II: Pass@1 Accuracy (%) Across Benchmarks (Mean ± SD, n = 5)

Baseline	HumanEval	MBPP	Custom-30	Overall
B1: Zero-Shot	85.06±1.83	72.48±2.11	36.67±4.71	64.74±2.22
B2: ReAct	87.80±1.52	76.12±1.87	46.67±5.44	70.20±2.28
B3: No Safety	90.24±1.21	79.64±1.53	60.00±4.08	76.63±1.74
B4: SafeCode-MAS	92.07±0.97	81.80±1.38	66.67±3.33	80.18±1.46

SafeCode-MAS (B4) achieves the highest Pass@1 across all three benchmarks, attaining 92.07% on HumanEval (8.2% improvement over B1), 81.80% on MBPP (12.8% improvement over B1), and 66.67% on Custom-30 (81.8% relative improvement over B1). The most substantial gains occur on the Custom-30 multi-file benchmark, demonstrating that the multi-agent architecture provides the greatest advantage on complex, multi-step tasks where task decomposition and iterative refinement are most impactful.

Statistical analysis confirms significance for all B4 vs. B1 comparisons (Wilcoxon $p < 0.05$) with large Cliff's delta effect sizes: HumanEval $d = 0.52$, MBPP $d = 0.68$, Custom-30 $d = 0.84$. The B4 vs. B3 comparison reaches significance on Custom-30 ($p = 0.031$, $d = 0.36$, medium) but not HumanEval ($p = 0.094$). Standard deviation consistently decreases from B1 (2.22) to B4 (1.46), indicating that the multi-agent pipeline produces more consistent and predictable outputs.

B. Security Vulnerability Reduction (RQ2)

Table III: Vulnerability Density Across Baselines

Baseline	Total Vulns	KLOC	Vuln/KLOC	Reduction
B1: Zero-Shot	187	14.2	13.17	—
B2: ReAct	156	15.8	9.87	25.1%
B3: No Safety	118	16.4	7.20	45.3%
B4: SafeCode-MAS	52	15.1	3.44	73.8%

SafeCode-MAS reduces vulnerability density by 73.8% compared to B1 and 52.2% compared to B3. Analysis by CWE category reveals that Critical severity vulnerabilities are most effectively targeted: SQL injection (CWE-89) reduced by 93.5%, command injection (CWE-78) by 87.5%, hardcoded credentials (CWE-798) by 81.0%, and insecure cryptography (CWE-327) by 75.0%. The Safety Agent's false positive rate was measured at 18.3% — meaning 81.7% of flagged findings are confirmed true vulnerabilities by the LLM-assisted assessment step. Notably, B4's Pass@1 (92.07%) exceeds B3's (90.24%) despite the additional security constraint, indicating that constraining the solution space toward secure patterns slightly enhances functional correctness.

C. Memory System Impact (RQ3)

Table IV: Memory Ablation Study Results (HumanEval + Custom-30)

Configuration	Pass@1 (%)	Vuln/KLOC	Quality	Avg. Iterations
Full (STM+LTM)	80.18±1.46	3.44	8.14	1.42
STM Only	77.84±1.72	4.18	7.76	1.61
LTM Only	74.52±2.31	5.92	7.23	1.78
No Memory	72.19±2.68	6.84	6.89	1.93

The ablation study demonstrates that both memory layers contribute measurably. Removing LTM while retaining STM reduces Pass@1 by 2.34 percentage points and increases vulnerability density by 21.5%. Removing STM while retaining LTM causes more severe degradation (5.66 pp decrease), confirming that immediate task context is critical for coherent generation. Sequential task analysis reveals progressive learning: across 10 web development

tasks processed in sequence, Pass@1 improved from 60% on task 1 to 80% on task 10 in the LTM-enabled configuration. Retrieval statistics show a 72.4% hit rate with mean similarity of 0.81.

D. Computational Overhead (RQ4)

Table V: Computational Cost Analysis

Baseline	Tokens/ Task	Latency (s)	Cost/Task ($)
B1: Zero-Shot	2,847	8.2	0.085
B2: ReAct	8,412	24.6	0.252
B3: No Safety	14,683	42.1	0.440
B4: SafeCode-MAS	18,924	53.8	0.568

SafeCode-MAS consumes 6.6× more tokens than zero-shot generation per task. The average iteration count of 1.42 means 58% of subtasks are approved on first attempt, with convergence rates CR(1) = 0.79, CR(2) = 0.94, CR(3) = 0.98. The cost-benefit analysis reveals a compelling economic case: industry estimates place manual security review at $15–50 per vulnerability. At an average reduction of 2.5 vulnerabilities per task, SafeCode-MAS provides estimated savings of $37.50–125 per task against the additional $0.48 API cost, representing a 78–260× return on the computational investment.

E. Qualitative Case Studies

Case Study 1: REST API Development. A Custom-30 task requiring a Flask REST API with JWT authentication was decomposed into 5 subtasks. The Reviewer caught missing expired-token error handling and inconsistent import paths (resolved in iteration 2). The Safety Agent detected a hardcoded JWT secret (CWE-798), replaced with os.environ.get() in iteration 3. Final output: 5 files, 247 lines, 0 vulnerabilities, pylint 8.4/10.

Case Study 2: SQL Injection Remediation. A data pipeline task generated SQL via f-string formatting. The Safety Agent flagged CWE-89 (B608, Critical). The Coder replaced it with parameterized queries in the next iteration. The LTM stored this vulnerable-to-secure pattern, and a subsequent database task proactively used parameterized queries without Safety Agent intervention — demonstrating cross-task learning through the Security Pattern Database.

Case Study 3: Progressive Quality Improvement. Three sequential utility library tasks showed progressive quality improvement through LTM accumulation: first-iteration quality scores of 6.8, 8.1, and 8.4 respectively, with the second and third tasks benefiting from error handling and documentation patterns accumulated during earlier tasks. The third task required zero review rejections, compared to two rejections for the first task.

VI. CONCLUSION

This paper presented SafeCode-MAS, a safety-aware multi-agent framework for automated software development that achieves 92.07% Pass@1 on HumanEval with 73.8% vulnerability reduction compared to zero-shot GPT-4 generation. The four contributions — safety-first architecture with Bandit and Semgrep integration, dual-layer memory with empirically validated progressive learning, bounded iterative refinement addressing sequential chaining degradation, and comprehensive multi-benchmark evaluation — collectively establish that safety-integrated multi-agent systems offer a data-driven approach to managing the quality, security, and efficiency of automated software development processes.

Several limitations warrant acknowledgment. Evaluation is restricted to Python code generation; generalization to other languages requires language-specific safety tools. Custom-30, while purpose-built and publicly released, is limited to 30 tasks. The backbone LLM significantly influences results, and effectiveness with smaller models requires investigation. Static analysis cannot detect runtime vulnerabilities or architectural security flaws.

Future research directions include extending SafeCode-MAS to support multiple programming languages through language-specific Safety Agent rule sets, implementing self-evolving agent capabilities where agents autonomously refine their prompts based on performance data, integration with CI/CD pipelines for real-world deployment evaluation, incorporating formal verification methods for critical code paths, conducting human-in-the-loop studies for developer trust calibration, and investigating lightweight model alternatives for cost reduction.

ACKNOWLEDGMENT

The authors gratefully acknowledge Swarrnim Institute of Technology, Gandhinagar, for providing

computational resources and academic support. This work was conducted as part of the Master of Technology program in Artificial Intelligence and Machine Learning. R. Mistry conceived the research, designed and implemented the framework, conducted experiments, and drafted the manuscript. R. Dungarani supervised the research, provided methodological guidance, and reviewed the final manuscript. The authors declare no conflict of interest. The SafeCode-MAS source code and Custom-30 benchmark will be released upon publication.

REFERENCES

[1] J. Austin et al., "Program synthesis with large language models," arXiv:2108.07732, 2021.

[2] N. Bui, T. Nguyen, & H. Le, "SWE-EVO: Benchmarking coding agents in long-horizon software evolution," arXiv:2512.18470, 2025.

[3] M. Chen et al., "Evaluating large language models trained on code," arXiv:2107.03374, 2021.

[4] DORA Research Program, "Accelerate State of DevOps Report 2025," Google Cloud, 2025.

[5] S. Hong et al., "MetaGPT: Meta programming for a multi-agent collaborative framework," in Proc. ICLR, 2024.

[6] H. Huang et al., "Position paper: Agent AI towards a holistic intelligence," arXiv:2404.15245, 2024.

[7] C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?" arXiv:2310.06770, 2023.

[8] H. Kaur & G. Singh, "A systematic review of software maintenance and evolution," Intl. J. Comp. Sci. & Info. Tech., vol. 6, no. 2, pp. 1874–1879, 2015.

[9] Y. Liu et al., "Trustworthy LLMs: A survey and guideline for evaluating LLM alignment," arXiv:2308.05374, 2023.

[10] A. Madaan et al., "Self-Refine: Iterative refinement with self-feedback," in Proc. NeurIPS, 2023.

[11] R. Mistry & R. Dungarani, "Autonomous intelligent agents: Evolution, design paradigms, and the software development revolution (2022–2025)," IJRDET, vol. 14, no. 12, pp. 89–93, 2025.

[12] M. Nguyen & D. N. Pham, "Comprehensive investigation of autonomous programming agents," arXiv:2407.02107, 2024.

[13] OpenAI, "GPT-4 technical report," arXiv:2303.08774, 2023.

[14] OpenAI, "Introducing SWE-bench Verified," OpenAI Blog, 2024.

[15] J. S. Park et al., "Generative agents: Interactive simulacra of human behavior," in Proc. UIST, 2023.

[16] H. Pearce et al., "Asleep at the keyboard? Assessing the security of GitHub Copilot," in Proc. IEEE S&P, 2022.

[17] C. Qian et al., "Communicative agents for software development," in Proc. ACL, 2024.

[18] B. Roziere et al., "Code Llama: Open foundation models for code," arXiv:2308.12950, 2023.

[19] Y. Ruan et al., "Identifying the risks of LM agents with an LM-emulated sandbox," arXiv:2309.15817, 2023.

[20] T. Shinn et al., "Reflexion: Language agents with verbal reinforcement learning," in Proc. NeurIPS, 2023.

[21] Y. Tang et al., "ToolAlpaca: Generalized tool learning for language models," arXiv:2306.05301, 2023.

[22] G. Wang et al., "Voyager: An open-ended embodied agent with large language models," arXiv:2305.16291, 2023.

[23] L. Wang et al., "A survey on large language model based autonomous agents," Frontiers of Comp. Sci., vol. 18, no. 6, 2024.

[24] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in Proc. NeurIPS, 2022.

[25] Q. Wu et al., "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," arXiv:2308.08155, 2023.

[26] Z. Xi et al., "The rise and potential of large language model based agents: A survey," arXiv:2309.07864, 2023.

[27] J. Yang et al., "SWE-agent: Agent-computer interfaces enable automated software engineering," arXiv:2405.15793, 2024.

[28] S. Yao, D. Yu et al., "Tree of thoughts: Deliberate problem solving with large language models," in Proc. NeurIPS, 2023.

[29] S. Yao, J. Zhao et al., "ReAct: Synergizing reasoning and acting in language models," in Proc. ICLR, 2023.

[30] D. Zan et al., "Multi-SWE-bench: A multi-language benchmark for real-world software engineering," 2025.

[31] D. Zhou et al., "Least-to-most prompting enables complex reasoning in large language models," in Proc. ICLR, 2023.
[32] T. Zhu et al., "MultiAgentBench: Evaluating the collaboration and competition of LLM agents," arXiv:2503.01935, 2025.