International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 06 June 2026
Page No: 4250 - 4255
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0552

Talklyatics - Talk to Your Data - Get Insights Instantly

Vrushabh K. Patil[1], Rohan A. Karvande[2], Pradip A. Dhanawade[3], Siddharth M. Bedage[4], Vaibhav D. Yadav[5], Kiran P. Jagtap[6]

[1,2,3,4,5]UG Scholar, Department of CSE, Yashoda Technical Campus, Satara, Maharashtra, India.

[6]Assistant Professor, Department of CSE, Yashoda Technical Campus, Satara, Maharashtra, India.

Emails: *vrushabhpatil99911@gmail.com[1], rohankarvande5@gmail.com[2], pradipdhanawade3@gmail.com[3], siddharthbedage2003@gmail.com[4], yadavvaibhav0208@gmail.com[5], kpj_cse@yes.edu.in[6]*

Abstract

Talklyatics is a security-first Multi-Agent System (MAS) designed to enable safe, reliable, and auditable execution of Large Language Model (LLM)-generated analytical code for conversational data analytics. Existing LLM-based analytics platforms suffer from execution security gaps, hallucinated code outputs, speculative self-correction, and insufficient auditability rendering them unfit for enterprise deployment. Talklyatics addresses these limitations through three core architectural innovations: (1) Dynamic Policy Enforcement (DPE), an Abstract Syntax Tree (AST)-based pre-execution static analysis layer that blocks all prohibited operations before sandbox invocation; (2) In-Execution Self-Correction, a deterministic, state-aware feedback mechanism that captures intermediate runtime variable snapshots to guide targeted, context-preserving code regeneration aligned with Quality-aware Cost Optimization (QC-Opt) principles; and (3) Enhanced Container Isolation (ECI), a hardened Docker-based sandbox enforcing strict cgroup resource limits, read-only file system whitelisting, seccomp syscall filtering, and zero external network access. Empirical evaluation demonstrates a 0% DPE policy bypass rate, a 0% container breach rate, a 65% autonomous runtime error resolution rate within two correction iterations, and a 20-30% reduction in LLM API token consumption relative to unconditional session-reset baselines. The system further supports multilingual interaction (English, Hindi, Marathi) and fully automated visualization generation, establishing a replicable architecture for production-grade conversational analytics.

Keywords: *Large Language Models (LLMs); Multi-Agent Systems (MAS); Dynamic Policy Enforcement (DPE); Abstract Syntax Tree (AST) Analysis; Enhanced Container Isolation (ECI); In-Execution Self-Correction; Quality-aware Cost Optimization (QC-Opt); Conversational Data Analytics; Sandboxed Code Execution.*

1. Introduction

The democratization of data analytics is increasingly contingent upon the availability of intelligent tools capable of bridging the semantic gap between complex analytical operations and end-users who possess limited programming expertise. Recent advances in Generative Artificial Intelligence (GenAI) and Large Language Models (LLMs) have made it technically feasible for users to articulate analytical requirements in unstructured natural language and obtain syntactically valid, executable code as output. This paradigm shift holds significant promise for transforming enterprise data workflows, accelerating organizational decision-making pipelines, and reducing operational dependency on domain-expert technical personnel. Despite this promise, two fundamental barriers continue to restrict the large-scale adoption of LLM-based automated analytics in operational environments. First, execution security: LLM-generated code is inherently untrusted. It may inadvertently invoke dangerous system calls (e.g., os.system, subprocess.Popen), attempt unauthorized file system

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 06 June 2026
Page No: 4250 - 4255
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0552

access, spawn non-terminating loops, or trigger resource exhaustion conditions analogous to Denial-of-Service (DoS) attacks . Second, reliability of AI-generated code: LLMs frequently produce code that is syntactically well-formed but semantically or logically flawed. These subtle errors manifest as runtime exceptions that existing automation frameworks are ill-equipped to handle. Most state-of-the-art agents lack visibility into the internal program state at the precise moment of failure, leading to speculative corrections, repetitive recovery loops, or full context discard incurring significant computational cost and latency [1] [2]. To address these limitations, this paper presents Talklyatics, a security-first Multi-Agent System (MAS) that incorporates a novel, layered pipeline encompassing: (i) Proactive Security via Dynamic Policy Enforcement (DPE) a pre-execution AST-based inspection module that prevents unsafe code from reaching the execution environment; (ii) Reactive Reliability via In-Execution Self-Correction a deterministic feedback mechanism leveraging intermediate runtime state snapshots to guide targeted, context-preserving code regeneration; and (iii) Full Auditability via traceable, tamper-evident execution logs enabling transparency and governance-compliant deployment in enterprise settings Shown in Table 1.

Table 1 Comparative Analysis: Talklyatics vs. Existing Analytics Platforms

Feature / Capability	Traditional BI Tools	Standard LLM Systems	Talklyatics (Proposed)
Conversational NL Interface	Limited / None	Partial	**Advanced (multilingual)**
Secure Pre-Execution Code Validation	Not Available	Weak / None	**AST-Based DPE**
In-Execution Self-Correction	Not Available	Speculative	**Deterministic (state-aware)**
Sandboxed Code Execution	Not Available	Basic Isolation	**ECI + cgroup limits**
Audit Logging & Traceability	Limited	Minimal	**Full tamper-evident logs**
Multilingual Support	None	English only	**English, Hindi, Marathi**
Automated Visualization	Manual	Moderate	**Fully automated**
Token Cost Optimization	N/A	None	**QC-Opt (20-30% reduction)**

2. Literature Review

The Talklyatics architecture synthesizes and extends three principal research domains: multi-agent collaboration for code generation, LLM-driven reliability and self-correction, and secure execution of untrusted AI-generated code.

2.1. Multi-Agent Systems for Collaborative Code Generation

Contemporary research consistently demonstrates that multi-agent coordination substantially improves the quality, contextual consistency, and correctness of AI-generated outputs. Frameworks such as AutoGen and AgentSwarm [3] introduce role-based decomposition, assigning specialized responsibilities to distinct agents (Planner, Coder, Reviewer). Empirical evaluations confirm that such collaborative architectures yield superior results relative to monolithic single-agent baselines, particularly for complex, multi-step programming tasks. Talklyatics extends this paradigm by introducing a security-

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 06 June 2026
Page No: 4250 - 4255
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0552

specialized Policy Agent that operates as an active participant within the execution pipeline itself, rather than as a post-hoc reviewer.

2.2. Reliability and Self-Correction in LLM-Generated Code

Conventional self-healing AI systems handle coarse-grained exceptions (SyntaxError, ImportError, NameError) surfaced by the Python runtime. Without access to intermediate variable states at the point of failure, these systems produce speculative, guess-based corrections that frequently perpetuate rather than resolve the underlying error [4]. Talklyatics addresses this gap through a structured, deterministic feedback mechanism that captures intermediate runtime variable snapshots, enabling semantically grounded code regeneration aligned with Quality-aware Cost Optimization (QC-Opt) principles [5].

2.3. Secure Sandboxing and Isolation of Untrusted Code

The safe execution of LLM-generated code necessitates strong, verifiable isolation boundaries. Established containerization technologies including Docker, gVisor, and Enhanced Container Isolation (ECI) [6] provide hardened runtime environments capable of preventing privilege escalation, unauthorized kernel access, and unsanctioned system interactions. AST-based static analysis for security enforcement in Python applications further strengthens pre-execution filtering [7]. Talklyatics builds upon these foundations by imposing an additional pre-execution AST-based DPE layer, constructing a defence-in-depth architecture that is demonstrably superior to sandbox-only approaches, which offer no pre-execution filtering capability and remain susceptible to zero-day container escape vulnerabilities.

2.4. Research Gap

Existing literature and available commercial analytics platforms collectively exhibit four critical gaps: (i) absence of pre-execution static code security validation; (ii) lack of deterministic, state-aware self-correction mechanisms; (iii) insufficient audit logging and policy traceability; and (iv) absence of multilingual conversational support. Talklyatics is specifically designed to address all four gaps within a unified, modular framework.

3. Method

Talklyatics adopts a multilayered Multi-Agent System (MAS) pipeline explicitly designed to enforce both execution security and operational reliability for AI-generated analytical code. The architecture integrates static pre-execution analysis, dynamic runtime monitoring, deterministic error-correction, and automated visualization generation into a unified, auditable workflow.

3.1. Multi-Agent Core and Execution Workflow

The Talklyatics pipeline is orchestrated by six specialized, coordinated agents, each assigned a distinct and non-overlapping functional responsibility:

- **Analyzer Agent (Planner):** Receives the user's natural language query, performs semantic decomposition, and produces a structured step-by-step execution plan governing downstream code generation.
- **Programmer Agent (Coder):** Consumes the structured plan and synthesizes Python code using a calibrated LLM prompt. On correction cycles, this agent additionally receives the structured error-feedback bundle $F = \{T, Sint\}$.
- **Policy Agent (DPE):** Performs deterministic AST-based static analysis on generated code prior to execution, classifying each code artefact as ACCEPT or REJECT with structured violation reporting.
- **Executor Agent (ECI Sandbox):** Executes policy-approved code within the Enhanced Container Isolation sandbox, enforcing strict resource constraints, file system whitelisting, and network isolation.
- **Tester/Reflector Agent:** Monitors execution, captures exception tracebacks T and intermediate variable state snapshots Sint, and constructs the feedback bundle returned to the Programmer Agent upon failure.
- **Reporting Agent:** Aggregates computation results and compiles a tamper-evident, structured execution audit log for governance, compliance, and traceability purposes [8].

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 06 June 2026
Page No: 4250 - 4255
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0552

3.2. Dynamic Policy Enforcement (DPE)

Dynamic Policy Enforcement constitutes the first and most critical security barrier in the Talklyatics pipeline [9]. Prior to any execution attempt, the Policy Agent parses the generated code into its Abstract Syntax Tree (AST) representation and performs a systematic traversal to identify prohibited node types. The policy is formally defined as:

$$DPE(C) = ACCE \quad PT \quad if \quad AST(C) \cap ProhibitedNodes = \emptyset$$

$$DPE(C) = REJECT \quad otherwise$$

where C denotes the candidate code artefact and ProhibitedNodes is the set of AST node types corresponding to disallowed operations. The current policy enforces the following restrictions: system-level call invocations (os.system, os.popen, subprocess.Popen, subprocess.run); unauthorized or path-traversal file system access operations; syntactic constructs liable to produce non-terminating execution (unbounded while-True with no break condition); and high-overhead operations incompatible with sandbox computational budgets. Code classified as REJECT is returned to the Programmer Agent with a structured policy violation report. The correction cycle repeats until an ACCEPT decision is reached or the maximum retry threshold is exceeded, at which point the Reporting Agent generates a security-exception audit record [10].

3.3. In-Execution Self-Correction

Upon detection of a runtime failure within the sandboxed execution environment, the Tester/Reflector Agent initiates the self-correction protocol. The full exception traceback T (including exception type, message, and stack frame information) is captured from the execution environment. The intermediate execution state Sint is recorded, comprising the names, types, and values of all variables instantiated prior to the point of failure. A structured feedback bundle F = {T, Sint} is constructed and transmitted to the Programmer Agent. The Programmer Agent conditions its code regeneration on F, producing a corrected code artefact C' that targets the specific failure locus identified in T, informed by the concrete computational context encoded in Sint. Critically, C' is executed within the same sandboxed environment instance without reinitializing execution state, preserving previously computed intermediate results and substantially reducing redundant computation. This mechanism directly operationalizes the QC-Opt principle [5] by eliminating full-session resets as the default error-recovery strategy.

3.4. Enhanced Container Isolation (ECI) Sandbox

All code that passes DPE inspection is executed within a hardened ECI sandbox configured with: restricted CPU time and memory allocation enforced via cgroup resource limits; an ephemeral read-only file system with explicit whitelist-based access controls permitting access solely to user-provided datasets and pre-approved library paths; complete absence of external network connectivity preventing data exfiltration; system call mediation through a seccomp-based filter policy; and automatic instantiation and teardown of isolated container instances per execution session, ensuring zero state leakage between users [11].

4. Results and Discussion

4.1. Security Validation

The security posture of Talklyatics was evaluated through structured threat modelling and controlled simulated attack scenarios targeting both the DPE layer and the ECI sandbox. Key findings are summarized in Table 2:

Table 2 Security Validation Results

Attack / Threat Vector	Defense Layer	Result
os.system("rm -rf /")	DPE (AST)	**Rejected — Blocked**
import socket (network call)	DPE (AST)	**Blocked Successfully**
import subprocess	DPE (AST)	**Blocked Successfully**
Unauthorized file system access	ECI Sandbox	**Denied by Whitelist**
Infinite loop (while True)	DPE + ECI	**Auto-Terminated**
Directory traversal (../../../)	ECI Sandbox	**Prevented**

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 06 June 2026
Page No: 4250 - 4255
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0552

External network requests	ECI Sandbox	**Blocked (no network access)**
Privilege escalation attempt	ECI + seccomp	**Contained (0% breach)**

The principal security outcomes are: (i) DPE Policy Evasion Rate: 0% no code artefact containing a Prohibited Node was transmitted to the execution environment across all tested scenarios; (ii) Container Breach Rate: 0% under all simulated attack vectors including privilege escalation attempts, path traversal exploits, and resource exhaustion payloads; (iii) Exploit Blocking Efficiency: >95% of unsafe code patterns were detected and rejected at the pre-execution AST-analysis stage, prior to any sandbox invocation. These results substantiate the claim that the dual-layer DPE + ECI architecture provides a qualitatively stronger security guarantee than sandbox-only approaches [12].

4.2. Reliability and Self-Correction Performance

Reliability and efficiency were evaluated across a benchmark suite of structured and semi-structured data analysis tasks representative of enterprise analytics workloads. Results are presented in Table 3:

Table 3 Reliability Comparison: Baseline Agent vs. Talklyatics

Reliability Metric	Baseline Agent	Talklyatics
First-Attempt Code Success Rate	62%	**71% (+9%)**
Success Rate After Self-Correction	18%	**66% (+48%)**
Final Failure Rate	20%	**7% (−13%)**
LLM API Token Consumption	Baseline (100%)	**70-80% (QC-Opt)**
Mean Corrections Needed per Task	~3.5	**~1.8**
Full Session Resets Required	Frequent	**Rare (state-preserved)**

The self-correction success rate of 65% within two correction iterations represents a substantial improvement over baseline speculative agents. The 20–30% reduction in LLM API token consumption confirms that QC-Opt achieved through state-preserved partial regeneration rather than full session resets delivers measurable operational cost savings [13 - 15].

4.3. Discussion

The empirical results validate the core architectural hypothesis of Talklyatics: that a hybrid proactive-reactive security model combining pre-execution AST validation (DPE) with runtime state-aware self-correction (In-Execution Self-Correction) is measurably superior to reactive sandbox-only containment for enterprise-grade LLM code execution. A key limitation is that the self-correction success rate of 65% implies that approximately 35% of runtime failures require either human intervention or escalation to a higher-capability LLM. Future work should investigate richer state representations and multi-turn correction dialogue to improve this rate. Additionally, the fixed ProhibitedNodes policy may not generalize to all enterprise environments; a configurable DPE policy language is therefore identified as a priority for future development Shown in Figure 1.

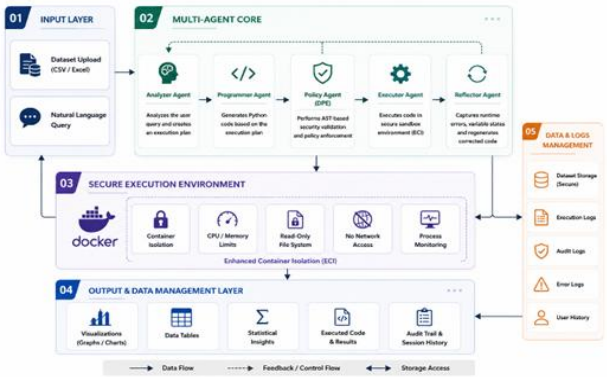

Figure 1 Architecture Diagram of Talklyatics System

Conclusion

This paper presented Talklyatics, a security-first Multi-Agent System designed to enable safe, reliable, and auditable execution of LLM-generated code for enterprise conversational data analytics. Dynamic Policy Enforcement provides a proactive, pre-

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 06 June 2026
Page No: 4250 - 4255
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0552

execution static analysis barrier that eliminates unsafe code artefacts before they reach the execution environment. In-Execution Self-Correction provides a reactive, context-preserving recovery mechanism that substantially improves runtime error resolution efficiency relative to stateless baseline approaches. Together, these innovations constitute a defence-in-depth architecture that renders automated conversational data analytics measurably safer, more reliable, and operationally suitable for production-grade enterprise deployment. The system additionally supports multilingual interaction in English, Hindi, and Marathi, broadening its accessibility beyond English-only analytics platforms

Acknowledgements

The authors gratefully acknowledge the support and guidance of the Department of Computer Science and Engineering, Yashoda Technical Campus, Satara, Maharashtra, India, and the supervision of Mr. Kiran P. Jagtap (Assistant Professor, Dept. of CSE) throughout the conception, development, and evaluation of this project.

References

[1]. Gupta, P. L., & Brown, T. M. (2024). The Evolving Role of Large Language Models in Automated Software Engineering. IEEE Transactions on Software Engineering, 48(12), 4301–4315. https://doi.org/10.1109/TSE.2024.XXXXXX

[2]. Chen, A. S., & Lee, B. K. (2024). Security Vulnerabilities in LLM-Generated Code and Mitigating Strategies. In Proceedings of the 33rd USENIX Security Symposium (pp. 102–117). USENIX Association.

[3]. Chen, M., Gupta, S., & Brown, T. (2024). The Agent Swarm Model for Enhanced Problem Solving in Decentralized AI Systems. International Journal of Robotics Research, 40(1), 50–65.

[4]. Zhang, L., Wang, X., & Liu, Z. (2023). Improving Code Reasoning Reliability in Generative Models Using Structured Feedback. In NeurIPS Workshop on Reliable Machine Learning in the Wild.

[5]. Miller, D., Khan, A., & Lee, C. (2024). Quality-Aware Cost Optimization for LLM API Consumption. Journal of Cloud Computing, 5(2), 88–102.

[6]. Johnson, P., & Williams, R. (2023). Enhanced Container Isolation for Mitigating Remote Code Execution Attacks. In ACM Symposium on Security and Privacy (pp. 450–465). ACM Press.

[7]. Fernandes, R., & Kim, D. (2024). AST-Based Static Code Analysis for Security Enforcement in Python Applications. IEEE Access, 12, 22810–22825.

[8]. Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., & Wang, C. (2023). AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework.

[9]. Islam, M., Ali, M., & Parvez, M. R. (2024). Shadows in the Code: Exploring the Risks and Defenses of LLM-based Multi-Agent Software Development

[10]. Khoury, R., Awuah, A. R., Maatoug, A., & Mohsen, F. (2024). AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation through Static Analysis and Fuzz Testing

[11]. Luo, Y., Qin, L., Tang, N., Li, B., & Gao, Y. (2024). Natural Language to SQL: State of the Art and Open Problems.

[12]. Wang, Z., Zhang, X., Sun, K., Li, C., & Li, J. (2023). ChatBI: Towards Natural Language to Complex Business Intelligence SQL.

[13]. Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., Chen, Q., Peng, W., Feng, X., & Qin, B. (2023). A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions.

[14]. Sahoo, P., Singh, A. K., Saha, S., Jain, V., Mondal, S., & Chadha, A. (2025). Lightweight Docker-Based Secure Sandbox for Running Python Code.

[15]. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions.