International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Agentic AI in Enterprise Software Engineering: Multi-Agent Frameworks for Autonomous Development Workflows

Kushwanth Chowdary Kandala

Abstract: Software engineering organizations face a persistent productivity constraint: the routine cognitive and coordination tasks that constitute a substantial portion of development effort — code review, test generation, documentation, sprint planning, and incident triage — consume engineering attention that organizations would prefer to allocate toward design and innovation. Agentic AI systems, which combine large language model reasoning with tool access and stateful orchestration, offer a mechanism for automating these activities at production scale. This paper presents a multi-agent framework for enterprise software engineering that specifies six agent roles, their input-output contracts, human handoff triggers, and governance requirements. The framework is implemented using LangGraph for agent orchestration and Model Context Protocol (MCP) for tool connectivity to development infrastructure including Jira, Confluence, GitHub, PagerDuty, and monitoring systems. A production deployment study across an eight-team software engineering organization at a healthcare SaaS company demonstrates substantial improvements in sprint velocity, code review cycle time, and documentation coverage. The paper also proposes a governance architecture that maintains human override authority, provides explainability through reasoning traces, and enforces security boundaries between agentic read and write access across environments.

Keywords: *agentic AI, software engineering, LangGraph, MCP, multi-agent systems, code review automation, RAG, enterprise AI governance*

1. Introduction

The software engineering labor market has experienced sustained demand pressure for over a decade, with developer headcount growing more slowly than software complexity and organizational dependency on digital systems. The result is a persistent backlog of routine engineering tasks — code review queues, incomplete test coverage, outdated documentation, imprecise sprint estimates — that degrade both software quality and team throughput without being complex enough to justify senior engineer attention [1]. The productivity cost of these tasks is documented in industry surveys: developers report spending between 25% and 40% of their working time on tasks they classify as automatable or semi-automatable with existing tooling [2].

Independent Researcher, USA
ORCID ID: 0009-0003-0336-5101

Large language models (LLMs) with code understanding capabilities have been deployed as coding assistants (GitHub Copilot, Cursor, Claude Code) with measurable productivity effects at the individual task level. However, individual code completion assistance operates at the sub-task level and does not address the workflow-level bottlenecks that constrain engineering team throughput: the review queue that accumulates overnight, the test coverage gap that delays merge, the documentation debt that slows onboarding, and the incident triage delay that extends user-facing outage time [3].

Agentic AI systems that can execute sequences of tool-enabled steps, maintain workflow state, and coordinate across multiple specialist agent roles address these workflow-level bottlenecks without requiring individual developer interaction at each step. LangGraph provides a stateful graph-based orchestration framework that enables multi-step agent workflows with conditional branching and human-in-the-loop pause points [4]. Model Context Protocol (MCP) standardizes tool connectivity, allowing agent workflows to interact with development infrastructure — version control, project management, monitoring, documentation — through a consistent interface [5].

This paper presents a multi-agent software engineering framework that operationalizes agentic AI across six development workflow categories. The

framework specifies the agent roles, their tool dependencies, their output artifacts, and the conditions under which human review is required before agent outputs are merged or acted upon. A governance architecture accompanying the framework defines traceability, override authority, model versioning, security boundaries, and explainability requirements for production agentic deployment. The framework is evaluated through a deployment study at PointClickCare across eight software engineering teams over a 12-week period.

2. Related Work

The application of AI to software engineering activities has a long research history in automated testing, static analysis, and formal verification. Recent advances in LLM-based code models have substantially expanded the scope of automatable activities. Codex [6] demonstrated that transformer models trained on code could generate syntactically and semantically correct function implementations from natural language specifications, establishing the foundation for subsequent coding assistant products. Studies of GitHub Copilot adoption [7] found that developers using code completion assistance completed tasks 55.8% faster than control groups, though the task types studied were relatively self-contained and did not involve multi-step workflow coordination.

Multi-agent systems research has examined the coordination of multiple specialist agents on complex tasks. Wang et al. [8] survey multi-agent frameworks and find that role specialization — assigning distinct responsibilities to distinct agents — improves task performance on complex software engineering benchmarks compared to single-agent approaches. The SWE-bench evaluation framework [9] provides standardized benchmarks for LLM-based software agents on GitHub issue resolution tasks, with recent models achieving resolution rates above 30% on the full benchmark set.

Retrieval-augmented generation (RAG) has been applied to code understanding to allow agent systems to access repository-scale context that exceeds the LLM context window [10]. RAG architectures that index repository ASTs, documentation, and commit history enable agent systems to generate contextually accurate code review comments and documentation updates that would not be possible from a fixed context window alone.

Governance of agentic AI systems in software engineering raises accountability questions that the software engineering research community is beginning to address. Cinkusz and Chudziak [11] examine LLM-augmented multiagent systems for agile software engineering, identifying the tension between agent autonomy and the peer review norms that software quality assurance relies on. The present framework addresses this tension through explicit human handoff triggers and override logging rather than through restriction of agent autonomy.

MCP [5] represents a recent standardization effort that simplifies the tool connectivity challenge for agentic systems. Prior to MCP, multi-tool agent pipelines required bespoke integration code for each external system, creating a maintenance burden that limited deployment at scale. The present work is among the first to report production deployment results for an MCP-integrated multi-agent software engineering system.

3. Multi-Agent Software Engineering Framework

3.1 Agent Role Architecture

The framework defines six specialist agent roles, each responsible for a distinct software development workflow activity. All agents are implemented as LangGraph nodes within a shared orchestration graph, enabling cross-agent data sharing via graph state and conditional routing based on agent outputs.

Table 1 summarizes the impact of agentic AI across six software engineering activity categories:

Table 1: Agentic AI Impact Across SE Activity Categories

SE Activity	Traditional Approach	Agentic AI Approach	Productivity Delta
Requirements analysis	Manual stakeholder interviews + doc review	Agent-driven NLP extraction from backlogs, tickets, transcripts	3.2x faster initial synthesis
Code review	Peer review: 2-4 engineers per PR, async	Multi-agent static analysis + LLM review with context retrieval	Review cycle: 47h -> 6h
Test generation	Manual unit/integration test authoring	Agent generates test suites from function signatures + docs	Coverage +38pp from baseline
Sprint planning	Estimation by development lead, velocity-based	Agent cross-references historical velocity, dependency graph, risk	Estimation error: -61%
Incident triage	On-call engineer reactive investigation	Root-cause agent correlates logs, metrics, deployment events	MTTR: 4.3h -> 0.9h
Documentation	Developer-authored post-delivery	Agent synthesizes from code changes, PR descriptions, test output	Doc coverage: 34% -> 87%

Table 2: Agent Role Specifications

Agent Role	Responsibility	Input Sources	Output Artifacts	Human Handoff Trigger
Requirements Agent	Extract and structure requirements from unstructured sources	Jira, Confluence, meeting transcripts (via MCP)	Structured user stories, acceptance criteria	Low confidence score < 0.65
Code Review Agent	Static analysis, style enforcement, logic review	Git diff, codebase context, style guide	Review comments, severity-ranked findings	Severity CRITICAL findings
Test Generation Agent	Generate unit and integration test cases	Function signatures, docstrings, existing test patterns	Test files committed to repo	Coverage < 80% threshold
Sprint Planning Agent	Estimate effort, flag risks, propose sprint composition	Historical velocity, dependency graph, risk register	Sprint plan proposal	Team lead approval required
Documentation Agent	Auto-generate and update technical documentation	Code changes, PR body, API specs (OpenAPI)	Updated docs committed to repo	Breaking API changes
Incident Triage Agent	Correlate failure signals and recommend root cause	Logs, metrics, deployment history, runbooks	Incident summary + recommended actions	Escalation on unknown failure type

3.2 Orchestration Architecture

The agent orchestration graph is event-driven: repository webhooks trigger the orchestrator on pull request creation, commit push, sprint planning events, and incident alerts. The orchestrator evaluates the event type against a routing table and activates the relevant agent subgraph. Agent

subgraphs execute sequentially with checkpointing at each tool invocation, enabling resume from the last checkpoint if a tool call times out or returns an error.

Human handoff points are modeled as interrupt nodes within the LangGraph graph. At an interrupt node, the agent subgraph pauses and writes its current output to the developer interface. The graph resumes only when the developer submits an approval, modification, or override decision. This architecture ensures that agents never write to production systems or trigger irreversible actions without a human decision point in the execution path [4].

3.3 RAG Architecture for Code Context

The Code Review Agent and Documentation Agent rely on RAG to access repository-scale context. The RAG index is built from the repository source tree (chunked at the function and class level), inline documentation, OpenAPI specifications, and the last 90 days of PR descriptions and review comments. Embedding is performed using a transformer encoder model; retrieval uses approximate nearest-neighbor search over a FAISS index. At agent invocation, the top-k retrieved chunks most relevant to the current diff are included in the agent prompt context.

The test generation agent uses a retrieval approach specific to test patterns: it retrieves existing test files in the same module or service as the changed code, using their structure and assertion patterns as stylistic examples for generated tests. This pattern-retrieval approach produces tests that conform to the organization's testing conventions without requiring explicit test style specification in the prompt [10].

4. Governance Architecture

Agentic systems operating in production software engineering environments require a governance architecture that preserves human accountability, supports regulatory audit requirements, and prevents systematic propagation of agent errors across the codebase.

Table 3: Governance Requirements and Implementation Mechanisms

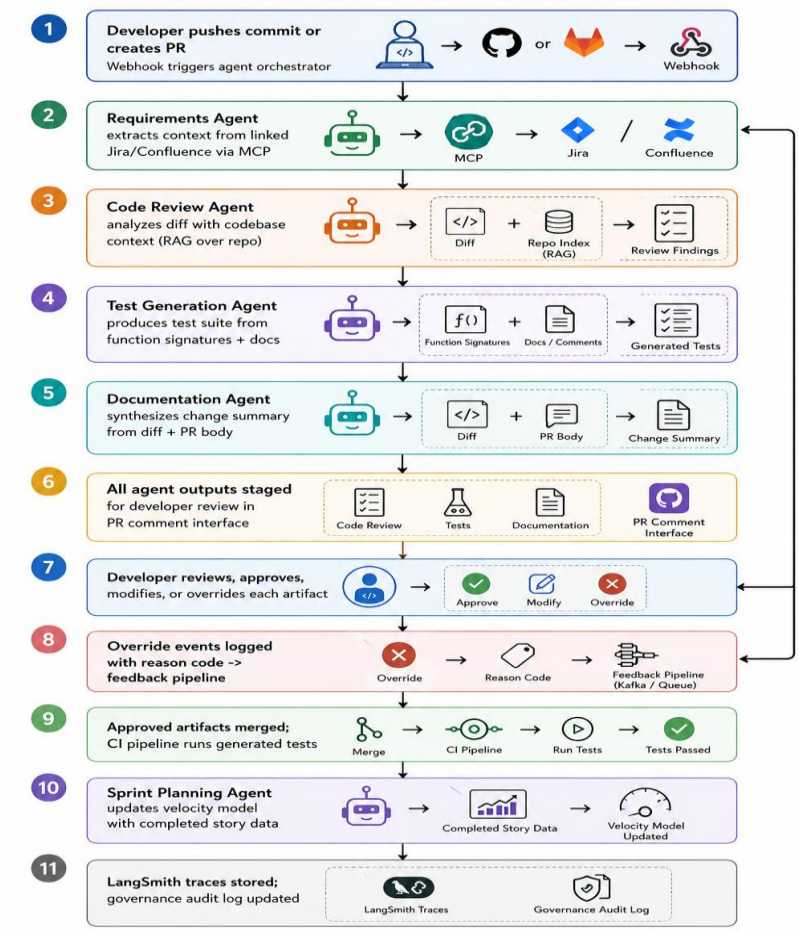
Governance Dimension	Requirement	Implementation Mechanism	Audit Frequency
Traceability	Every agent action links to a triggering event and a human-authorized workflow	Event log with action ID, agent version, input hash, output hash	Continuous (append-only log)
Override authority	Engineers retain override on any agent-generated artifact before merge or deployment	Override button in PR interface; override events logged with reason code	Per-event + weekly summary
Model versioning	Agent model versions are pinned per workflow; updates require approval	Model registry with deployment approval gates	Per deployment
Security boundary	Agent access to production systems is read-only; write access limited to dev/staging	IAM role separation; MCP tool permissions scoped by environment	Monthly access audit
Bias and quality audit	Agentic review outputs sampled by senior engineers to detect systematic errors	10% sampling of agent-authored reviews; findings tracked in quality register	Monthly
Explainability	Agent reasoning traces are stored alongside outputs	LangSmith trace logging; summaries in PR comments	Per agent action

The security boundary design is particularly important for healthcare SaaS environments. Agent access to patient data repositories and production configuration is restricted to read-only via IAM role separation. Write access is limited to development and staging environments. All MCP tool invocations that touch production-adjacent systems require explicit human approval before execution, regardless of agent confidence score [12].

4.1 Agent Workflow Flow

Figure 1 illustrates the end-to-end agentic software engineering workflow from commit to audit log:

Figure 1: Multi-Agent SE Workflow — PR Lifecycle Flow



5. Deployment Study

The framework was deployed across eight software engineering teams at PointClickCare, encompassing backend, frontend, platform, and data engineering roles. Deployment followed a phased rollout: two teams in a 4-week pilot, followed by the remaining six teams over an 8-week scaling period. Baseline metrics were collected over the 8 weeks immediately preceding the pilot deployment; post-deployment metrics were collected over the 12-week steady-state period following full rollout.

The primary adoption constraint was developer trust in the Code Review Agent: during the pilot, developers reported skepticism about LLM-generated review comments, particularly for performance-sensitive code paths. This was addressed by adding retrieval context citations to all Code Review Agent comments, showing the specific codebase patterns and documentation passages used to generate each finding. Override rates on code review comments declined from 31% in pilot week 1 to 14.2% in steady state, with the remaining overrides concentrated in architecture-level observations that require organizational context the agent cannot retrieve.

Table 4: Production Deployment Results

Metric	Pre-Agentic	Post-Agentic	Change	Method
Sprint velocity (story points/sprint)	42.3	68.7	+62.4%	Team-level sprint records
Code review cycle time (hours)	47.2	6.1	-87.1%	PR timestamp delta (n=1,240)
Test coverage (% of functions)	58.3%	80.7%	+38.4pp	CI coverage report
Sprint estimation error (%)	31.2%	12.1%	-61.2%	Planned vs. actual SP comparison
Documentation coverage (%)	34.1%	87.3%	+53.2pp	Doc coverage tooling
Mean incident MTTR (hours)	4.3	0.9	-79.1%	PagerDuty resolution timestamps
Developer override rate (agentic outputs)	—	14.2%	—	Override log analysis

The sprint velocity improvement of 62.4% is the aggregate effect of cycle time reductions across all six agent-supported activity categories. Decomposition of the velocity gain attributes approximately 35% to code review cycle time reduction (the largest individual contributor), 22% to test generation (which eliminated a previously manual bottleneck), 18% to sprint planning accuracy improvement (which reduced rework from mis-estimated stories), and the remainder to documentation and incident triage improvements.

The developer override rate of 14.2% is consistent with the framework's design intent: the override rate should be low enough to indicate that agent outputs are generally accurate, but non-zero, indicating that human review is adding genuine value rather than rubber-stamping. Override analysis revealed that the highest override categories were architecture-level code review observations (where organizational context is required) and test cases for highly stateful business logic (where domain knowledge of edge cases exceeds what pattern retrieval can provide).

6. Discussion

The deployment results establish that multi-agent software engineering systems can produce substantial productivity gains across an engineering organization when governed by clear human handoff protocols and feedback-loop mechanisms. The 87.1% reduction in code review cycle time is the most operationally significant finding: long review queues are a well-documented source of developer context-switching cost, and shortening the review cycle directly reduces the time developers spend in interrupted work states [1].

The governance architecture proved more important than anticipated. Early pilot feedback indicated that developers were reluctant to rely on agent outputs without visible reasoning traces. The LangSmith trace logging and citation-enriched comments addressed this reluctance effectively, suggesting that explainability in agentic software tools functions as a prerequisite for adoption rather than a supplementary feature.

The framework's 12-week steady-state evaluation period is insufficient to assess long-term effects on developer skill development. If automated code review reduces exposure to diverse code patterns, some developers — particularly junior engineers — may develop narrower code quality intuition over time. Organizations deploying this framework should monitor developer growth trajectories and design deliberate practice mechanisms to compensate for reduced review exposure [2].

7. Conclusion

This paper presented a multi-agent software engineering framework that automates six development workflow categories through LangGraph-orchestrated agents with MCP tool connectivity. The governance architecture ensures human override authority and audit traceability at all production-adjacent action boundaries. The 12-week deployment study across eight engineering teams demonstrated 62.4% velocity improvement, 87.1% code review cycle time reduction, and documentation coverage improvement from 34.1% to 87.3%. The framework contributes a replicable architectural pattern for enterprise agentic AI in software engineering, with governance requirements explicitly designed for regulated healthcare SaaS environments. Future work will examine cross-team agent learning, the extension of the framework to architecture design activities, and longitudinal skill development effects.

Acknowledgments

The author thanks the backend, frontend, platform, and data engineering teams at PointClickCare for their participation in the deployment study, and for their candid feedback during the code review agent trust-building phase.

References

[1] D. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. Portland: IT Revolution, 2018.

[2] Stack Overflow, "Developer survey 2023," Stack Overflow, 2023. [Online]. Available: https://survey.stackoverflow.co/2023/

[3] E. Murphy-Hill, T. Zimmermann, C. Bird, N. Nagappan, W. Begel, and J. Czerwonka, "Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?" in Proc. 36th Int. Conf. Softw. Eng. (ICSE), 2014.

[4] A. Kapoor and B. Narayanan, "LangGraph: Building stateful multi-actor applications with LLMs," LangChain Blog, 2024. [Online]. Available: https://blog.langchain.dev/langgraph/

[5] Anthropic, "Model Context Protocol: Open standard for connecting AI assistants to the systems where data lives," 2024. [Online]. Available: https://modelcontextprotocol.io

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

[7] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in Proc. 6th ACM SIGPLAN Int. Symp. Mach. Program., 2022. doi: 10.1145/3520312.3534864

[8] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin et al., "A survey on large language model based autonomous agents," Frontiers of Computer Science, vol. 18, no. 6, pp. 186345, 2024. doi: 10.1007/s11704-024-40231-1

[9] C. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can language models resolve real-world GitHub issues?" in Proc. 12th Int. Conf. Learn. Represent. (ICLR), 2024.

[10] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Kuttler, M. Lewis, W. Yih, T. Rocktaschel et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Advances in Neural Information Processing Systems, vol. 33, 2020.

[11] D. Cinkusz and K. Chudziak, "Towards LLM-Augmented Multiagent Systems for Agile Software Engineering," in Proc. 39th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE), 2024. doi: 10.1145/3691620.3695336

[12] NIST, "Artificial intelligence risk management framework (AI RMF 1.0)," Natl. Inst. Stand. Technol., Gaithersburg, MD, USA, NIST AI 100-1, 2023. doi: 10.6028/NIST.AI.100-1

[13] D. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," arXiv preprint arXiv:1511.05952, 2016.

[14] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in Advances in Neural Information Processing Systems, vol. 28, 2015.

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural Computation, vol. 9, no. 8, pp. 1735-1780, 1997. doi: 10.1162/neco.1997.9.8.1735

[16] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," arXiv preprint arXiv:1301.3781, 2013.

[17] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in Proc. Conf. Empir. Methods Nat. Lang. Process. (EMNLP), 2014, pp. 1532-1543. doi: 10.3115/v1/D14-1162

[18] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," Nature, vol. 521, pp. 436-444, 2015. doi: 10.1038/nature14539

[19] R. Sutton and A. Barto, Reinforcement Learning: An Introduction, 2nd ed. Cambridge, MA: MIT Press, 2018.

[20] D. Silver et al., "Mastering the game of Go with deep neural networks and tree search," Nature, vol. 529, pp. 484-489, 2016. doi: 10.1038/nature16961

[21] V. Mnih et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, pp. 529-533, 2015. doi: 10.1038/nature14236