



LAYER 2 OF THE AI-FIRST SDLC: A PROPOSED AGENT-DRIVEN ARCHITECTURE FOR BUILD, VALIDATION, AND RELEASE READINESS

Ambar Nath Saha ¹ and Debashis Patra ^{2,*}

¹ Information Technology, Cognizant Technology Solutions Canada Inc., Nova Scotia, Canada.

² Computer Science, UT Austin, TX, USA.

* Corresponding Author

ORCID Details

Ambar Nath Saha: <https://orcid.org/0009-0006-4457-4753>

Debashis Patra: <https://orcid.org/0009-0004-7894-1852>

World Journal of Advanced Research and Reviews, 2026, 32(01), 374–381

Publication history: Received on 29 August 2026; revised on 04 October 2026; accepted on 06 October 2026

Article DOI: <https://doi.org/10.30574/wjarr.2026.32.1.2549>

Copyright © 2026 Author(s) retain the copyright of this article. This article is published under the terms of the Creative Commons Attribution License 4.0

ABSTRACT

This paper proposes Layer 2 of the AI-First Software Development Lifecycle (SDLC) introduced in our parent framework and extended through the Layer 1 Product and Designer Layer. Layer 2 is a conceptual Build, Validate, and Release architecture intended to transform a Product Requirement Bundle (PRB) and a Design Artifact Bundle (DAB) into a documented, validated, merge-ready pull request. It defines six proposed specialist agents—Architecture, Coding, Testing, Review, PR, and CI Gate—coordinated by a Layer 2 Sub-Orchestrator. The design includes a bounded Feedback-Driven Repair Loop, revalidation after code modifications, non-overridable CI and security gates, and a mandatory human merge decision. The proposed design draws on the authors' technical analysis, review of existing software-engineering and agentic-AI approaches, and exploratory proof-of-concept work. The integrated architecture has not been implemented or empirically evaluated; potential effects on delivery time, code quality, cost, and compliance remain hypotheses for future evaluation.

Keywords: Agentic AI; AI-First SDLC; Multi-Agent Orchestration; Software Architecture; Automated Testing; Feedback-Driven Repair

1 INTRODUCTION

Our parent paper proposed an AI-First Software Development Lifecycle (SDLC) spanning thirteen phases from epic creation through production monitoring [1]. A companion Layer 1 paper defined how a Product and Designer Layer could transform a natural-language epic into a Product Requirement Bundle (PRB) and Design Artifact Bundle (DAB) [2]. The next design problem is how those artifacts might guide architecture decisions, implementation, testing, code review, and a human-controlled merge.

In established software-delivery practice, build, test, review, and integration activities depend on coordination among people, tools, and quality controls [3]. Research prototypes such as ChatDev, SWE-agent, and MetaGPT have demonstrated different forms of agentic collaboration and repository interaction [4, 5, 6]. These developments motivate

a conceptual layer that organizes those capabilities around structured requirements, bounded retries, traceable artifacts, and explicit human authority, rather than assuming that code generation alone constitutes a complete delivery process.

Layer 2 corresponds to Phases 3 through 10 of the previously proposed AI-First SDLC. It specifies six specialist agents—Architecture, Coding, Testing, Review, PR, and CI Gate—coordinated by a Layer 2 Sub-Orchestrator and subject to the shared Knowledge Layer and Guardrails. Throughout this paper, agent behaviors are proposed design responsibilities, not claims about a deployed implementation. We illustrate the design with the same example used in the earlier papers: a user notification-preference feature supporting email, SMS, and push notifications.

Design basis and status: The conceptual framework was developed through extensive engineering-oriented analysis of existing SDLC processes and agentic-AI approaches, informed by literature and exploratory proof-of-concept work undertaken by the authors. These activities informed architectural choices but did not produce a working end-to-end Layer 2 prototype. No measured results, production deployments, or quantitative claims of effectiveness are presented in this article.

2 RELATED WORK

2.1 AI-Powered Code Generation

Large language models and agentic development systems can assist with code generation, repository navigation, editing, and test execution. ChatDev, SWE-agent, and MetaGPT provide distinct examples of role-based collaboration and tool-using agents [4, 5, 6]. Commercial development tools provide further practical context; their capabilities and integrations change over time. Our contribution is not a claim that backlog integration, code generation, or automated correction is individually new. It is the proposed arrangement of six responsibilities within a PRB/DAB-driven SDLC layer with defined artifacts and governance checkpoints.

2.2 Automated Testing with LLMs

LLM-supported software testing and bug reproduction have been investigated in prior research [7]. Rigorous evaluation of generated code also shows why passing a small or incomplete test set cannot establish general correctness [8]. Within Layer 2, the proposed Testing Agent would derive test cases from acceptance criteria and report execution evidence to a bounded repair controller. The paper does not assert that this method has been validated in an integrated prototype.

2.3 AI-Assisted Code Review

Static analysis and code-review platforms can surface quality and security concerns, while agents may be used to propose or apply changes. The proposed Review Agent would combine automated findings with organization-specific coding standards and return proposed corrections for revalidation. A review-generated code change would invalidate the earlier test status until relevant regression tests and required checks pass again. The design therefore treats review as an additional control, not a guarantee of correctness [8, 9].

2.4 CI/CD Intelligence

CI/CD systems already automate deterministic build, test, and release checks [10]. Layer 2 proposes a CI Gate Agent that interprets pipeline evidence and summarizes residual concerns for the human reviewer. Its reasoning is advisory: it cannot override mandatory build failures, required tests, blocking security findings, or compliance rules. This separation between hard policy gates and contextual analysis is a design constraint, not an observed performance result.

3 POSITIONING WITHIN THE AI-FIRST SDLC

Layer 2 follows Layer 1 and accepts its PRB and DAB. Its proposed responsibility is to prepare a reviewed, CI-validated, merge-ready pull request. It does not merge code or deploy it autonomously: the human merge checkpoint is mandatory, and Layer 3 (Runtime, Deployment, and Recovery) would begin only after that approval. In the parent framework this is the yellow Build, Validate, and Release layer, covering Phases 3 through 10. Figure 1 summarizes the proposed roles and controls.

4 REFERENCE ARCHITECTURE DIAGRAM

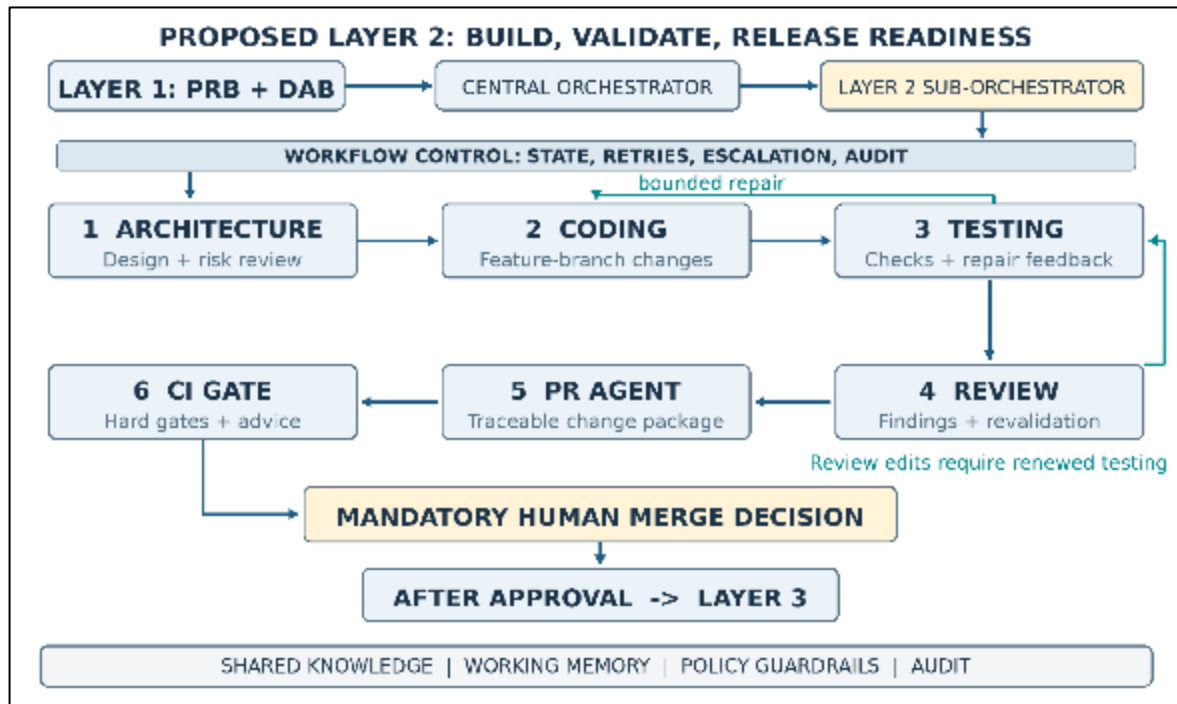


Figure 1: Proposed Layer 2 architecture and control points. All agent flows are conceptual; hard CI gates and the human merge decision remain mandatory.

5 THE LAYER 2 SUB-ORCHESTRATOR

The proposed Layer 2 Sub-Orchestrator coordinates the six specialist agents while the Central Orchestrator retains responsibility for the broader SDLC. Its conceptual interfaces are `beginLayer2(prbId, dabId)` and `mergeReady(prId)`. It would maintain workflow state, working memory, a configurable repair budget, and escalation rules. The normal sequence is Architecture, Coding, Testing, Review, PR creation, and CI Gate. A test failure, review modification, or CI failure would return affected work to the appropriate earlier stage. Stage-specific and overall rework budgets would bound these transitions; exhaustion of either budget would stop automatic retries and escalate to a human. An unresolved mandatory-policy failure would block progression until resolved. No runtime implementation of this state machine is reported here.

6 THE SIX AGENTS OF LAYER 2

6.1 Architecture Agent

The proposed Architecture Agent would read the PRB and DAB, consult the shared Knowledge Layer for approved system context, and draft technical designs for APIs, database changes, integration contracts, and system interactions [11]. An authorized connector based on the Model Context Protocol (MCP) could publish design artifacts to Confluence and update approved knowledge records [12]. In the notification-preference example, the agent would specify preference endpoints, a preference data model, and interfaces with existing notification services. These are illustrative design outputs, not artifacts generated by a deployed Layer 2 system.

Human-in-the-loop: The optional architecture checkpoint would be triggered by defined complexity or risk conditions. An authorized senior architect or tech lead could approve, amend, or replace a proposed technical design. Lower-risk designs could proceed under predefined organizational policy; the system would not be permitted to bypass any otherwise mandatory architecture or security approval.

6.2 Coding Agent

Following architecture approval, the proposed Coding Agent would operate in a restricted development sandbox and create changes on an isolated feature branch. It would use acceptance criteria, approved architecture, codebase context, and non-functional requirements to draft application, integration, configuration, logging, and error-handling changes. Agentic tools and containerized environments are illustrative integration choices, not evidence that such an

environment was deployed for this paper. For the notification example, candidate outputs would include preference REST endpoints, service logic, and channel integrations [5].

6.3 Testing Agent and the Feedback-Driven Repair Loop

The proposed Testing Agent would derive functional and integration tests from the PRB, DAB, acceptance criteria, and applicable organizational test patterns. Depending on feature risk, it could also propose performance, security, and mutation-testing checks. The agent would execute available tests in an authorized environment and preserve the outcomes as evidence. Research on LLM-supported bug reproduction provides relevant background, while findings about inadequate test suites caution against equating test passage with proof of correctness [7, 8].

When a test fails, the proposed Feedback-Driven Repair Loop would send the failure details to the Coding Agent for a candidate correction. The Testing Agent would then rerun the affected tests and applicable regression checks. The Sub-Orchestrator would enforce a configurable stage retry budget (for example, three to five cycles), count the attempts against the overall rework budget, and escalate unresolved cases when either limit is reached. Repeated passing tests would be a required progression criterion, not a claim that generated code is defect-free. The loop has not been evaluated for convergence, timing, or reliability in an integrated system.

6.4 Review Agent

After the required tests pass, the proposed Review Agent would inspect code and reports for maintainability, style, performance concerns, and security findings, using approved static-analysis tools such as SonarQube and applicable coding standards [9]. It could recommend or perform bounded corrective changes where permissions permit. Crucially, any modification to executable code would return the affected branch for suitable testing and review checks; the previously recorded passing status could not simply be reused. Findings that exceed the agent's authority would be escalated to a human reviewer.

6.5 PR Agent

The proposed PR Agent would assemble a pull request with a change summary, technical design, relevant test and review evidence, pipeline status, and links to the originating Jira story or epic. The aim is to make the proposed work traceable and easier for a human to examine, not to eliminate the need for that examination. In the notification example, the PR would link the preference APIs to the corresponding requirements and validation artifacts.

6.6 CI Gate Agent

The proposed CI/CD pipeline would execute mandatory build, regression, policy, and security checks using organization-approved systems such as GitHub Actions, SonarQube, and Trivy [9, 13, 14]. The CI Gate Agent would then interpret the results, highlight unusual nonblocking warnings, and prepare a reasoned recommendation for the human reviewer. It could mark a PR as ready for human review only when every mandatory gate passes; model confidence would never override a required test, security, or compliance failure. If a gate fails, the Sub-Orchestrator would route the work for correction subject to the overall rework budget and require the relevant checks to run again. The CI Gate Agent's recommendation is advisory, and final merge authority remains with a human.

Illustrative scenario (hypothetical): For a notification-preference feature, the Architecture Agent would draft APIs and the Coding Agent would prepare changes. Failed acceptance tests would trigger bounded repair; review edits would require renewed testing. The PR Agent would package traceability evidence, while the CI Gate Agent would report mandatory checks and advisory concerns. Only an authorized human could approve the merge. No such end-to-end execution is reported here.

Figure 2 summarizes the proposed failure-handling decisions, including bounded repair, revalidation after review edits, mandatory CI gates, and human escalation. It depicts a conceptual workflow, not an executed prototype.

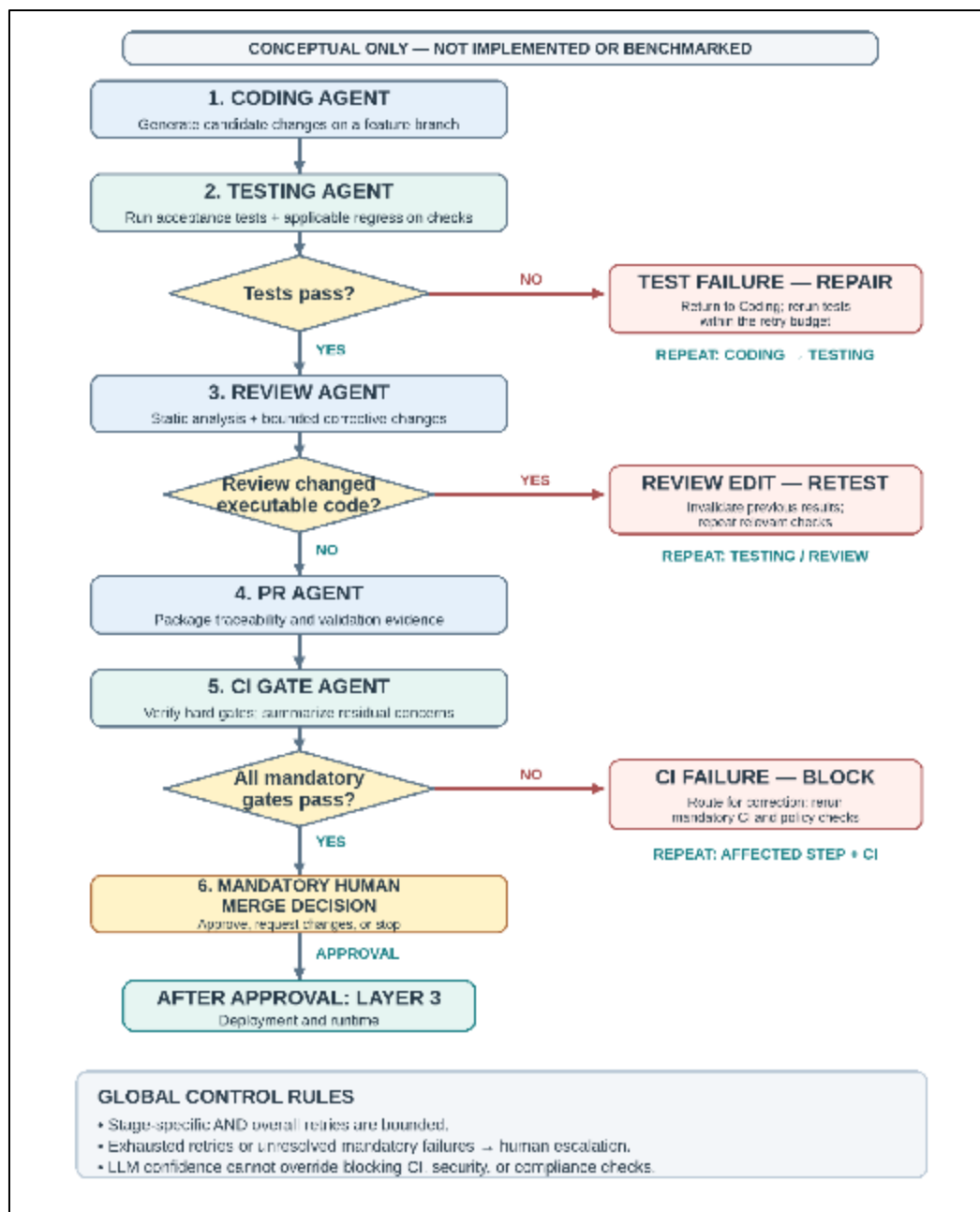


Figure 2: Proposed feedback-driven repair, revalidation, and CI-gate workflow. Mandatory human review remains the final merge checkpoint; failure paths are governed by stage-specific and overall retry budgets.

7 HUMAN-IN-THE-LOOP CHECKPOINTS

The proposed workflow contains one risk-dependent human architecture checkpoint and one mandatory human merge checkpoint. For high-risk or complex architecture, an authorized architect would review the technical design; any additional organizational approvals would remain in force. Before a final merge, a developer, tech lead, or engineering manager would review the documented PR, current validation results, security findings, residual risks, and design rationale. This mandatory decision remains with the human reviewer, as in the parent framework [1]. An agent's positive assessment is not a substitute for human authorization.

8 COMPONENT AND ARTIFACT SUMMARY

Table 1 summarizes the proposed agents, and Table 2 identifies the principal artifacts and their intended consumers. The listed capabilities and artifacts specify design responsibilities rather than implemented features.

Table 1: Layer 2 agent summary.

Agent	Proposed integration	Intended output
Architecture Agent	Reasoning LLM + approved knowledge / MCP tools	Technical design for review
Coding Agent	Restricted coding agent / sandbox	Candidate code on feature branch
Testing Agent	Test-generation agent + test runners	Test plans and execution evidence
Review Agent	LLM + static-analysis tools	Findings / proposed corrections
PR Agent	Repository API / MCP tools	Documented pull request
CI Gate Agent	Pipeline results + policy engine	Hard-gate status + advisory summary

Table 2: Primary artifacts flowing through Layer 2.

Artifact	Proposed producer	Intended consumer
Technical design (API / DB / integration)	Architecture Agent	Coding Agent; knowledge layer
Candidate code (feature branch)	Coding Agent	Testing Agent; Review Agent
Test cases + execution reports	Testing Agent	Coding Agent (repair); PR Agent
Review findings + changed code	Review Agent	Testing Agent (revalidation); PR Agent
Pull request with documentation	PR Agent	CI/CD pipeline; CI Gate Agent
Gate status + risk analysis	CI Gate Agent	Mandatory human merge reviewer

9 PROPOSED BENEFITS, CHALLENGES, AND DISCUSSION

9.1 Potential Build-to-Merge Efficiency

The proposed orchestration could reduce handoff delays by moving test feedback, review findings, and CI results directly between specialized agents. Related agentic software-engineering systems illustrate the feasibility of selected component tasks [4, 5, 6]. However, this paper does not report an integrated Layer 2 deployment, measured build-to-merge times, cost savings, or a comparison with human teams. Any efficiency benefit would depend on task complexity, integration quality, required approvals, and the number of failed repair attempts, and must be tested empirically.

9.2 Code Quality Through Layered Controls

In the proposed flow, generated code would undergo acceptance-based testing, review and static analysis, and independent CI checks before a human merge decision. Review-generated changes would require renewed validation. The CI Gate Agent would synthesize evidence while deterministic controls retain blocking authority. These multiple controls are intended to reduce avoidable defects but cannot establish defect-free software; empirical studies of code generation show that limited test coverage can miss substantial errors [8].

9.3 Bounded Feedback-Driven Repair

The proposed Feedback-Driven Repair Loop would allow the Coding Agent to respond to reproducible test failures and resubmit corrected code for validation without waiting for a new manual handoff. Its bounded retry policy also defines a clear failure path to human support. The design may reduce coordination time for suitable tasks, but repeated repair cycles, incorrect fixes, flaky tests, or incomplete acceptance criteria may offset those gains. Its reliability and time cost remain open evaluation questions.

9.4 Security, Compliance, and Human Governance

The design combines static-analysis findings, pipeline security checks, and organization-specific policies [9, 14, 15]. For notification preferences, personal information such as email addresses and telephone numbers would need controls appropriate to applicable privacy obligations, including GDPR where relevant [16]. Retrieved instructions, repository content, and connector outputs may also create prompt-injection risks, so tool permissions, input handling, and audit records would require independent controls [17]. These are proposed safeguards: the paper does not demonstrate actual regulatory compliance or resistance to attacks.

9.5 Limitations and Future Evaluation

This manuscript presents a conceptual architecture informed by the authors' technical analysis and exploratory proof-of-concept activities; the six-agent workflow has not been integrated or evaluated end to end. Important uncertainties include the quality and currency of PRB/DAB inputs, reliability of agent-generated code and tests, failure or oscillation across repair, review, and CI retry cycles, changes introduced during review, dependency on undocumented legacy systems, connector and credential security, model and CI operating costs, and the ability of a human reviewer to assess complex generated changes. A future evaluation should report a reproducible prototype, representative tasks, comparison baselines, build-to-merge time, defect escape rate, mandatory-gate adherence, human intervention frequency, and resource cost. No such results are claimed here.

10 CONCLUSION

This paper develops a conceptual specification for Layer 2 of the AI-First SDLC: six specialist agents coordinated by a Sub-Orchestrator to prepare a traceable, CI-validated, merge-ready pull request from Layer 1 requirements and design artifacts. The distinctive design choices are bounded repair, revalidation after code modifications, hard CI and security gates, and mandatory human merge authority. The framework was informed by research, technical analysis, and exploratory proof-of-concept work, but no integrated implementation or performance study is presented. Future work must determine whether the architecture achieves its intended benefits in real software-delivery settings.

STATEMENTS ON COMPLIANCE WITH ETHICAL STANDARDS

Acknowledgments

The authors thank the software-engineering and DevOps communities whose published work on agentic development, testing, review, and CI/CD informed the conceptual analysis presented here.

Disclosure of conflict of interest

The authors declare no competing interests.

Author contribution

A. N. Saha and D. Patra jointly developed the conceptual Layer 2 architecture, defined the agent responsibilities and proposed control flow, and drafted and revised the manuscript. Both authors approved the final version.

Data and materials availability

This article presents a conceptual architectural proposal and does not report a deployed system, empirical study, implementation benchmark, or experimental dataset. The authors' exploratory proof-of-concept activities are not offered as an end-to-end validation of the six-agent architecture. Publicly cited literature and tool documentation are identified in the references.

REFERENCES

- [1] Saha AN, Patra D. AI-First Software Development Lifecycle: An Agent-Driven Framework for Autonomous Planning, Coding, Testing, and Deployment. ESP J Eng Technol Adv. 2026;6(1):131-139. Available from: <https://espjeta.org/jeta-v6i1p115>
- [2] Saha AN, Patra D. Layer 1 of the AI-First SDLC: An agent-driven product and designer layer for autonomous discovery, requirement refinement, experience design and backlog generation. World J Adv Res Rev. 2026;30(2):1355-1364. doi:10.30574/wjarr.2026.30.2.1381.
- [3] Kim G, Humble J, Debois P, Willis J, Forsgren N. The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations. 2nd ed. IT Revolution; 2021.

- [4] Qian C, et al. ChatDev: Communicative Agents for Software Development. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers); 2024. p. 15174-15186. doi:10.18653/v1/2024.acl-long.810.
- [5] Yang J, et al. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *Adv Neural Inf Process Syst.* 2024;37. doi:10.52202/079017-1601.
- [6] Hong S, et al. MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework. In: International Conference on Learning Representations; 2024. Available from: <https://mlanthology.org/iclr/2024/hong2024iclr-metagpt/>
- [7] Kang S, Yoon J, Yoo S. Large Language Models Are Few-Shot Testers: Exploring LLM-Based General Bug Reproduction. In: Proceedings of the 45th International Conference on Software Engineering; 2023. p. 2312-2323. doi:10.1109/ICSE48619.2023.00194.
- [8] Liu J, Xia CS, Wang Y, Zhang L. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. *Adv Neural Inf Process Syst.* 2023;36:21558–21572. doi:10.52202/075280-0943.
- [9] SonarSource. SonarQube documentation [Internet]. Available from: <https://docs.sonarsource.com/>
- [10] Humble J, Farley D. Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley; 2010.
- [11] Bass L, Clements P, Kazman R. Software Architecture in Practice. 4th ed. Addison-Wesley; 2021.
- [12] Model Context Protocol. Specification [Internet]. 2026-07-28. Available from: <https://modelcontextprotocol.io/specification/2026-07-28>
- [13] GitHub. GitHub Actions documentation [Internet]. Available from: <https://docs.github.com/en/actions>
- [14] Aqua Security. Trivy documentation [Internet]. Available from: <https://trivy.dev/docs/latest/>
- [15] OWASP Foundation. OWASP Top 10:2025 [Internet]. Available from: <https://top10.owasp.org/2025/en/>
- [16] European Parliament and Council. Regulation (EU) 2016/679 (General Data Protection Regulation). Official Journal of the European Union; 2016. Available from: <https://eur-lex.europa.eu/eli/reg/2016/679/oj>.
- [17] Greshake K, Abdelnabi S, Mishra S, Endres C, Holz T, Fritz M. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. In: Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security; 2023. p. 79–90. doi:10.1145/3605764.3623985.