

ACGF-Lite: A Lightweight Agentic Code Governance Framework for Fintech Core Banking Development – An Industrial Case Study

Oscar Mendez*, Jorge Hernandez and Hector Florez

Universidad Distrital Francisco José de Caldas, Bogotá, Colombia

Abstract

Agents are increasingly used in software development. In artificial intelligence (AI), agents, in addition to attempting to reason, can have a degree of autonomy, thus giving rise to the term autonomous agents. Therefore, when using autonomous agents for software development, a framework is needed to regulate how the AI agent will interact and what responsibilities it has. In this paper, *ACGF-Lite* (*Agentic Code Governance Framework - Lite*) is presented to specify how to integrate AI agents into software development. Furthermore, a case study is presented for a company focused on creating financial services in Bogotá, Colombia. The use case is presented in different phases to show that using the framework can lead to substantial improvements in the governance of software development processes, optimizing the operational cost of code generation.

Keywords

Autonomous agents, Agentic software engineering, AI-assisted software development, Software governance, Human-agent collaboration, Fintech, ACGF-Lite

1. Introduction

Today, with the rise of Large Language Models (LLMs), there has been a significant increase in the apparent speed and productivity of software development. However, it is important to consider that writing code quickly does not always guarantee the expected quality or ensure that software quality criteria have been properly evaluated. Furthermore, the risk of violating regulations and compliance standards—which could jeopardize business operations or institutions implementing this type of software—is high when structured processes governing the artifacts generated by these models are not established [1].

Previously, developers typically sought solutions from colleagues on wikis or open-source projects to implement software improvements. Then came copilots, where developers would make a request and the copilot would respond. With the emergence of LLMs, the concept of agents regained prominence. Agents are hardware or software components capable of taking actions based on the context in which they are deployed [2]. Thus, in software engineering, an agent can plan, modify multiple files, execute commands, and attempt to correct its own errors. If autonomy is added, these agents must be capable of iterating until the product reaches stability in accordance with specified requirements. However, the challenge arises in determining the extent to which an agent should be permitted to act autonomously within code generation processes.

The rapid and unconstrained adoption of LLMs for code generation demands clearly defined and structured, reproducible, observable, and monitorable processes that delineate the responsibilities of both humans and AI agents. In the fintech industry, merely creating stable products is insufficient; it is essential to deliver secure systems that comply with regulatory controls, such as the Digital Operational Resilience Act (DORA), the Payment Card Industry Data Security Standard (PCI DSS), and the General Data Protection Regulation (GDPR). These regulatory controls will be detailed later. Consequently,

ICAIW 2026: Workshops at the 9th International Conference on Applied Informatics 2026, October 13–17, 2026, Orlando, United States

✉ omendez@udistrital.edu.co (O. Mendez); jehernandezr@udistrital.edu.co (J. Hernandez); haflorezf@udistrital.edu.co (H. Florez)

ORCID 0009-0002-2353-6185 (O. Mendez); 0000-0002-7129-7919 (J. Hernandez); 0000-0002-5339-4459 (H. Florez)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

this paper aims to ensure that when an AI agent is involved in software development, a mechanism is established to determine whether or not the agent should intervene. In addition, a real-world case study is presented. The study was conducted at a fintech company where the framework was tested to validate its functionality regarding code governance in software development processes within a live environment, allowing the comparison of various controlled scenarios from ideation to production deployment and traffic evaluation on the built artifacts.

1.1. Identified gap and contributions

While the studies presented later in this manuscript have examined AI-assisted software development, code quality, human–AI collaboration, and AI regulatory aspects, these dimensions are often considered independently. Therefore, straightforward governance mechanisms are required to integrate agent autonomy, human accountability, software quality, security, regulatory compliance, and economic cost within a practical software development workflow. Accordingly, the main contributions are presented:

- Generate a framework called ACGF-Lite to define when AI should interact within the software lifecycle—or, in other words, to ensure code governance for agent-generated code in a real-world fintech context. See section 4.2.
- Integrate autonomy levels with the mechanisms through which AI agents should govern software development phases. Refer to sections: 3.2 and 4.2.1.
- Present an industrial case study to evaluate the framework focused on a fintech company in Bogotá, Colombia. See section 4.3.

The core research question of this manuscript is: How does agent autonomy affect development process operational costs, software quality, and software build times in a *fintech* environment? The remainder of this manuscript is organized as follows. Section 2 introduces the background concepts. Section 3 presents related work. Section 4 describes the materials and methods, including the industrial case study and the experimental design. Section 5 presents the results obtained from evaluating ACGF-Lite. Section 6 discusses the findings, their implications, and the limitations of the study. Section 7 outlines future work. Finally, Section 8 summarizes the primary conclusions and outlines potential directions for future research.

2. Background

In this section, the main concepts for understanding the ACGF-Lite framework are discussed. First, how LLMs have evolved as software development assistants is outlined. Then, the responsibilities between humans and agents are explained, and finally, the necessary controls that AI-powered solutions must have are explained. These concepts are presented below.

2.1. Basic concepts in software development

Software development involves various activities, typically starting with planning and culminating in the final product reaching the end user. Several key concepts are essential for the proper design of AI agents within the software development process. The first is *continuous integration and continuous delivery (CI/CD)*, strategies designed to deliver the solution to a specific environment. Another key concept is the comparative evaluation of software quality through *quality assurance (QA)* testing. Defects identified during QA testing provide an observable indicator of potential problems that persist after implementation and before delivery. Also, in agent-based software development, it's the *correction loops*, which refer to an iterative process in which an AI agent generates or modifies code, runs a validation or test, analyzes the resulting feedback, and makes another modification. Finally, lines of code (LOC) represent the volume of source code produced during development.

2.2. Specification-driven development (SDD)

This approach involves software development based on explicit specifications of what needs to be built. In this case, it starts with explicit definitions of what the system should do and what its requirements are. When using agents, these specifications typically serve as structured inputs that guide the agents in planning, implementing, testing, and validating software changes.

2.3. Agentic software engineering

Today, thanks to LLMs, software engineering has moved from a simple interaction between developers' requests and responses to an autonomous approach to coding the evolution and adaptation of computer applications. Conventional AI assistants support developers by auto-completing code, finding errors, or answering questions about how the software project works. In software engineering, agents expand these capabilities by allowing AI agents to pursue objectives through multi-step actions [3].

Agents can be equipped with capabilities or skills that allow them to perform specific activities in software engineering such as code generation, testing, debugging, documentation, static code analysis, or, for example, interaction with development tools.

2.4. Human-agent collaboration and governance

Human-agent collaboration requires defining the degree of autonomy granted to an agent and the conditions under which human intervention is required. Low-risk activities, such as structuring, formatting, documentation, or generating initial project test definitions, may be suitable for automated execution, while activities involving critical business logic, architectural decisions, or the storage of sensitive data may require it. From this perspective, governance can be understood as the set of rules and controls that regulate the capabilities, permissions, oversight, and accountability of agents throughout the software development process. Without defined governance, agents could introduce a large number of errors, which are unintentional but can have this impact due to a lack of context.

2.5. Regulatory context in financial software

Software for financial institutions must operate with the premise of meeting security, resilience, data protection, and auditability requirements. The actions of AI agents can generate changes that lead to errors due to a lack of proper governance. The Digital Operational Resilience Act (DORA), for example, establishes requirements for ICT risk management, incident notification, resilience testing, and third-party ICT risk management for financial institutions in the European Union [4]. Similarly, the Payment Card Industry Data Security Standard (PCI DSS) provides technical and operational requirements designed to protect payment account data [5]. The General Data Protection Regulation (GDPR) establishes principles and requirements for the processing and protection of personal data, including data minimization, purpose limitation, and security [6].

3. Related Work

This section focuses on three thematic areas: 1. Agent-based software engineering, 2. AI governance in human-agent collaboration, and 3. Regulatory compliance in software development. The following details how these topics have been addressed.

3.1. AI-assisted and agentic software development

The evolution of task assistance in software engineering has moved from reactive autocomplete models to agentic systems [3, 7, 8]. Unlike traditional copilots, autonomous agents pursue complex goals through multi-stage project planning, loop reasoning, interaction with file system tools, and iterative self-correction [9]. However, the literature indicates that faster code generation does not directly imply

compliance with quality criteria or business requirements [10]. Empirical evaluations of code agent use highlight that the absence of explicit controls increases the risk of introducing technical debt, vulnerabilities, and a loss of code understanding by engineering and development teams, known as knowledge voids [11, 12].

3.2. AI governance and human-agent collaboration

Controlling and monitoring autonomous agents in the software development life cycle (SDLC) requires formal human-agent collaboration (Human-in-the-Loop) frameworks with bounded governance [13, 14]. Agent-based governance defines the rules, restrictions, and authorization levels that regulate the agent's ability to act throughout the software development process [15].

Recent research suggests structuring autonomy through levels of permission (*bounded autonomy*) [13]: while low-risk operational tasks (such as documentation, formatting or generating templates for testing) can be executed with a high level of autonomy, architectural decisions, modifications to core business logic, and the manipulation of sensitive data require mandatory and rigorous human approval and supervision (*approval gates*) [16].

3.3. AI governance in regulated software development with focus on fintechs

In fintech companies, the adoption of autonomous agents is contingent upon compliance with strict non-functional requirements for auditability, resilience, and information security [17, 18]. Agent systems must operate in accordance with the regulatory standards relevant to the local context in which the product is operating, for example:

- **DORA.** It establishes strict controls over operational continuity, ICT risk management, and the auditability of third-party automated components [19].
- **PCI DSS v4.0.** It requires rigorous technical protection and traceability in the handling of card data that enables financial transactions [20].
- **General Data Protection Regulation (GDPR).** It regulates data management, the right to be forgotten, data portability, and privacy by design in processing flows [19].
- **Habeas data.** Such as the right that every person has to know, update, and rectify the information that has been collected about them in databases of any type of public or private nature.

Therefore, despite isolated contributions in human-AI collaboration or code security, there is a significant gap in the literature on *lightweight frameworks* that operationally articulate and structure agent autonomy, human *accountability*, economic development metrics, and regulatory compliance on production financial platforms. The **ACGF-Lite framework** addresses this gap by providing a practical and reproducible structure for governing agent-based development in the fintech sector [21].

4. Materials and methods

This section presents the formulation of the proposed framework. It also outlines how it can be evaluated through a case study.

4.1. Overview architecture

Figure 1 summarizes the architecture of the ACGF-Lite framework as a projection of seven operational layers that cover the lifecycle of an agent-assisted task for the context in which the case study is developed: (1) system inputs, (2) code-generating AI agent, (3) the four governance matrices (task classification, approval gates, quality gates, and data classification), (4) the six-stage CI/CD pipeline, (5) the human roles that sign the approval gates, (6) regulatory compliance coverage, and (7) the auditable output artifacts. It is important to note that layers 1 through 4 describe the main task flow as follows:

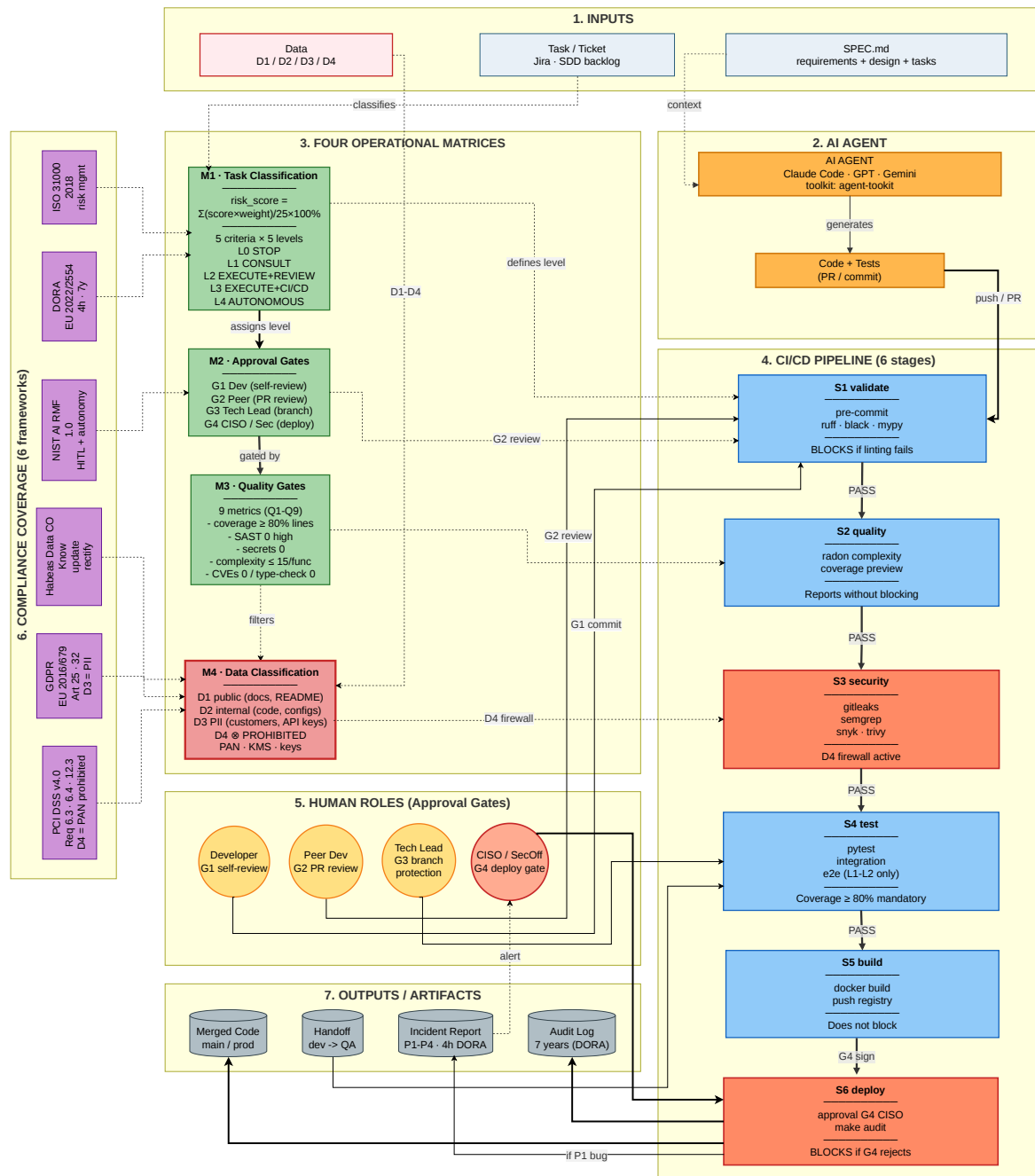


Figure 1: ACGF-Lite Overview Components

input, then agent, then classification, and finally execution. Layers 5 and 6 function as cross-cutting controls. In addition, human roles comprise specific pipeline gateways, as follows: G1 and G2 over S1, G3 over S4, and G4 over S6. Furthermore, the regulatory frameworks DORA, PCI DSS, and GDPR suggest restrictions on M1, M2, and M4. The M4 matrix establishes the D4 firewall rule, which restricts the agent's access to payment data or cryptographic keys; the S3 (security) and S6 (deploy) stages are blocking; and the G4 gateway—signed by the CISO—is mandatory for any production deployment task. For greater clarity, the relevant components of the framework are detailed in the following sections.

4.2. ACGF-Lite framework architecture and components

The **ACGF-Lite framework** is designed as a lightweight and agile architecture to standardize and regulate the integration of autonomous AI agents into the software development lifecycle. Its main objective is to resolve the operational tension between the speed of code generation by agents and the stringent requirements for code quality, auditability, cost control, and regulatory compliance demanded by the *Fintech industry*. The framework can be cloned and tested from the repository: <https://gitlab.com/oscfrayle/acgf-lite>.

4.2.1. Levels of agentic autonomy and risk scoring

ACGF-Lite defines the agent’s operational scope using a five-level autonomy scale ranging from 0 to 4, determined quantitatively by a weighted risk scoring equation:

$$\text{risk_score} = \frac{\sum(\text{criteria_score} \times \text{weight})}{25} \times 100\% \quad (1)$$

The weighted criteria assess data exposure (25%), the irreversibility of the action (25%), the impact on users/APIs (20%), technical complexity (15%), and critical dependencies (15%). Table 1 shows the levels of autonomy used by the presented framework. It also presents the rules based on the level of autonomy, criteria, and some application examples. It is important to note that the criteria values can be adjusted for each specific case study.

Table 1

Task classification and agentic autonomy levels in ACGF-Lite.

Level	Name	Autonomy	Criterion / Threshold	Application Examples
0	STOP	0%	$\text{risk_score} < 40\%$ (restrict by D4 data/irreversibility)	DROP DATABASE, direct production changes.
1	CONSULTATION	10–30%	$\text{risk_score} < 40\%$ (high impact or new dependencies)	Suggestions, <i>code review</i> , major refactoring, API changes.
2	EXECUTE + REVIEW	40–60%	$40\% \leq \text{risk_score} \leq 60\%$ (requires approval prior to merge)	New <i>features</i> , endpoints, data models, integrations.
3	EXECUTE + CI/CD	70–85%	$60\% < \text{risk_score} \leq 80\%$ (validated by test coverage and CI)	Unit tests, simple refactoring, <i>bug</i> resolution.
4	AUTONOMOUS	90–100%	$\text{risk_score} > 80\%$ (trivial and reversible tasks)	Documentation, <i>linting</i> , formatting (<i>Black/Ruff</i>), comments.

4.2.2. Repository structure and component matrix

Table 2 shows how a software project can govern agent autonomy using ACGF-Lite. In this way, the framework establishes governance by separating normative artifacts, automation packages or predefined templates, the automation suite, the test suite, and the operational function.

4.2.3. Quality control using gates and regulatory governance

ACGF-Lite adapts its controls to the previously outlined regulatory requirements through nine *quality gates* and four *approval gates*:

- **Quality gate:** This component is important for locking CI/CD. Thus, a change must pass the quality gates by meeting the following requirements: 0 critical issues in static analysis, 0 high/critical findings in SAST, test coverage of at least 80% of lines and 70% of branches, cyclomatic complexity of no more than 15 per function, and no exposed secrets or credentials.
- **Human approval gates:** Require explicit signatures for production deployments (G1: Tech Lead + Security), public API modifications (G2: Tech Lead + PO), access to restricted D4 data (G3: Security + Data Owner), and addition of new dependencies (G4: Security).

Table 2Directory structure and functional description of the `acgf-lite` repository.

Folder / File	Component / Type	Description and Operational Function
ACGF-LITE.md	Main Document	Regulatory framework that consolidates the 10 governance sections of the framework.
config/	YAML Configurations	Adjustable files for the risk matrix, <i>quality gates</i> , data classification, and approval gates.
automation-suite/configs/	CI/CD Artifacts	Templates ready for deployment of <code>.gitlab-ci.yml</code> , <code>.pre-commit-config.yml</code> , and <code>pyproject.toml</code> .
automation-suite/scripts/	Deterministic Scripts	Bash/Python modules for quality validation, security (SAST), <i>scaffolding</i> , <i>pipelines</i> , and deployment.
automation-suite/rules/semgrep/	Fintech SAST Rules	Custom static analysis rules in <code>fintech-security.yml</code> to detect vulnerabilities in financial services.
Makefile	Unified Entry Point	Defines standardized daily commands (<code>make check</code> , <code>make test-cov</code> , <code>make security</code> , <code>make secrets</code> , <code>make sdd-validate</code>).
templates/	Governance Templates	Formats for task classification (<code>task-classification.md</code>) and mandatory incident reporting (<code>incident-report.md</code>).
references/	Knowledge Base	Repository of more than 45 verifiable sources (DORA, PCI DSS v4.0, GDPR, NIST AI RMF, ISO 31000).

- **Alignment with DORA, PCI DSS v4.0, and GDPR:** Ensure incident notification within less than 4 hours and record retention for 7 years (DORA), strict prohibition of handling PAN card numbers (PCI-01 to PCI-03 / D4 Data), and PII minimization (D3 Data) under human supervision.

4.2.4. Quality Gates en CI/CD

To reinforce the principle of “*don’t repeat what’s in CI/CD*,” the framework defines nine quality gates that are automatically executed in the pipeline. Blocking metrics (BLOCK) cancel the *merge* or deployment when they fail; warning metrics (WARNING) report unblocked technical debt, according to a three-phase approach recommended in the repository. Table 3 details the threshold, the type of block, and the tool that implements each control.

Table 3

Quality Gates required by ACGF-Lite in the CI/CD pipeline.

Gate	Metric	Threshold	Type	Tool	Description
QG-1	Static analysis (linting)	0 errors	BLOCK	ruff, flake8, eslint	Style and code errors in each PR.
QG-2	SAST (static security analysis)	0 high, 0 critical	BLOCK	semgrep, bandit	OWASP Top 10 vulnerabilities and fintech rules.
QG-3	Test coverage (lines)	$\geq 80\%$	BLOCK	pytest-cov, jest	Lines covered by unit/integration tests.
QG-4	Branch coverage	$\geq 70\%$	BLOCK	pytest-cov -branch	Logical paths covered in tests.
QG-5	Cyclomatic complexity	≤ 15 / function	WARNING	radon, complexity	Maintainability per function.
QG-6	Secrets detected	0	BLOCK	gitleaks, detect-secrets	Credentials, API keys, tokens in code.
QG-7	CVEs in dependencies	0 critical	BLOCK	snky, trivy	Vulnerabilities in libraries and Dockerfile.
QG-8	Strict type-checking	0 errors	WARNING	mypy	Type consistency in Python.
QG-9	Post-deploy audit log	generated	BLOCK	make audit	Traceability entry before production.

4.2.5. Gates for human approval

The four ACGF-Lite approval gates (G1–G4) define who must approve what type of change before the code reaches production. Unlike quality gates, approval gates are always blocking and depend on the autonomy level (L0–L4) and the data classification (D1–D4) involved in the task. Table 4 shows the four gates, their triggering action, the required approvers, and the mechanism in the pipeline.

- **Activation Rule:** A task at level L0 or L1 triggers all four gates mandatorily; a L2 task requires G1–G3 and, if it involves D3/D4 data, also G4; a L3 task requires G1–G2, escalating to G3 when

Table 4

Human Approval Gates in ACGF-Lite, indexed by the signing role. The Trigger column indicates the scenarios in which each gate must fire; a single action may require the simultaneous fulfillment of multiple gates. This table unifies the notation presented in the architecture diagram (Fig. 1) and in the reference repository <https://gitlab.com/oscfrayle/acgf-lite>.

Gate	Signing Role	Required Approver(s)	CI/CD Mapping	Trigger
G1	Developer (self-review)	The developer themselves	pre-commit hooks (linting, format, type-check)	Every commit or pull request.
G2	Peer Developer (PR review)	1 developer from the team	Pull Request review + branch checks	Any PR targeting main or prod.
G3	Tech Lead or Senior Developer	Tech Lead <i>and</i> Product Owner (for public APIs)	Branch protection rules + codeowners	Changes to public API, access to D3 data, new external dependencies.
G4	CISO or Security Officer	Security Lead <i>and</i> Data Owner (for D4)	Deployment gate + make audit (7-year audit log)	Production deployment, access to D4 data, or any compliance-relevant change.

accessing D3 data; an L4 task only requires G1, escalating to G2 when it involves D2+ data or public APIs.

4.2.6. Data classification and agent permissions

To determine the autonomous agent’s behavior, a fourth axis was generated. This axis is a matrix that allows classification into levels D1, D2, D3, and D4, so that greater sensitivity requires greater human supervision. The table 5 shows the details of the proposed levels. For each level, the proposed name, representative examples, the access granted to the agent, and the logging and alerting requirements are specified.

Table 5

Permissions and data classification.

Level	Name	Examples	Agent Access	Logging
D1	Public	Documentation, READMEs, <i>open source</i> , marketing	Full access	Not required
D2	Internal	Source code, non-sensitive configs, application logs, tests	Access with <i>audit log</i>	Mandatory <i>audit log</i>
D3	Confidential	Customer data, PII, production configs, internal API keys	Read-only + G3 approval	<i>Audit log</i> + automatic alert
D4	Secret	PAN (cards), SSN, cryptographic keys, KMS, production credentials	⊗ PROHIBITED without explicit human supervision + G4	Any access = immediate alert to CISO

- **Firewall rule:** Under no circumstances may autonomous agents access D4 data without human supervision and G4 approval. Any attempt to access data, whether by copying, reading, or inference, must generate an alert and immediately block the agent’s execution. This rule is important because it is a strict limit of the proposed framework. Furthermore, it should be noted that this rule takes precedence over any autonomy optimizations.

4.2.7. Alignment with regulatory and normative standards

ACGF-Lite articulates its controls with six regulatory frameworks and international standards relevant to fintech. The coverage is both **documentary and procedural**: the framework provides the necessary structure to support compliance during an audit, although the technical implementation of specific requirements (encryption at rest, *key management*, retention policies) remains the responsibility of the fintech company adopting the framework. Table 6 summarizes the mapping.

Table 6

Mapping between ACGF-Lite and regulatory frameworks applicable to the fintech sector.

Regulation / Standard	Articles / Requirements	ACGF-Lite Controls
DORA (EU 2022/2554)	Art. 8 (ICT Risk Mgmt), Art. 9 (Incident Reporting 4h), Art. 10 (Resilience Testing)	Risk Score (Mod. 1) + Incident Response (Mod. 7) + audit logs 7 years
PCI DSS v4.0	Req. 6.3 (Secure Dev), 6.4 (Code Reviews), 12.3 (Security Awareness)	Approval Gates G2–G4 + Quality Gates QG-2, QG-6 + D4 data prohibited
GDPR (EU 2016/679)	Art. 25 (Privacy by Design), Art. 32 (Security of Processing)	D3 data with restricted access + audited encryption in CI (QG-3)
Habeas Data (CO)	Right to know, update, and rectify personal data	<i>Audit log</i> on D3 + export/deletion templates
ISO 31000:2018	Risk management framework	Adjustable weights (default, <i>fintech_banking</i> , <i>startup_agile</i>) in <i>config/risk-matrix.yaml</i>
NIST AI RMF 1.0	Autonomy levels + Human-in-the-Loop	Levels L0–L4 + Approval Gates G1–G4

4.3. Case study design

Figure 2 presents the proposed case study to demonstrate that the framework can be used in a real-world setting. The study was conducted at a Colombian company specializing in creating software services for digital money lending. At the time of the study, the company had 24 technology employees. These employees held positions such as backend developers, frontend developers, infrastructure engineers, quality assurance engineers, UI/UX designers, and project managers. Five key projects were studied at the company: Contracts, Messaging, Connection Model, Contract Manager, and Bank Accounts.

The case study considered three phases to validate autonomy within the proposed framework. The phases consisted of the following:

- **Phase A.** This phase was when no autonomous agents were being used to develop source code in the projects. The data collected was the history of each project and was obtained using the Jira tool. The extracted data included development effort, quality outcomes, CI/CD execution times, and code volume.
- **Phase B.** This represents an end-to-end (E2E) software development approach, in which AI agents were allowed to participate in the entire software development process using an SDD approach.
- **Phase C.** Phase C represents the ACGF-Lite-governed approach, in which the human and the agent have specific responsibilities.

The evolution of the source code included business changes based on product needs. The changes studied were primarily backend-oriented, where changes in business logic were evident. This configuration allowed for a comparison of development effort between each phase.

4.4. Evaluation Metrics

The phases of the case study presented above were evaluated using metrics to measure development effort, software quality, operating expenses, and resource consumption when using the agents. Development effort was measured by the team hours required to complete the selected projects. Software quality was assessed based on the number of defects identified during QA activities. Operational overhead was characterized by the average execution time of CI/CD processes and the number of self-correction cycles observed during agent-assisted development. Additionally, the LOC values were recorded as a descriptive indicator of the volume of code generated. For the phases that used agents, the consumption of language model tokens was also recorded, considering both input and output tokens.

For each metric, the results of the agent-based phases were compared to the first phase. Relative changes were calculated using:

$$\Delta M = \frac{M_{\text{phase}} - M_{\text{baseline}}}{M_{\text{baseline}}} \times 100 \quad (2)$$

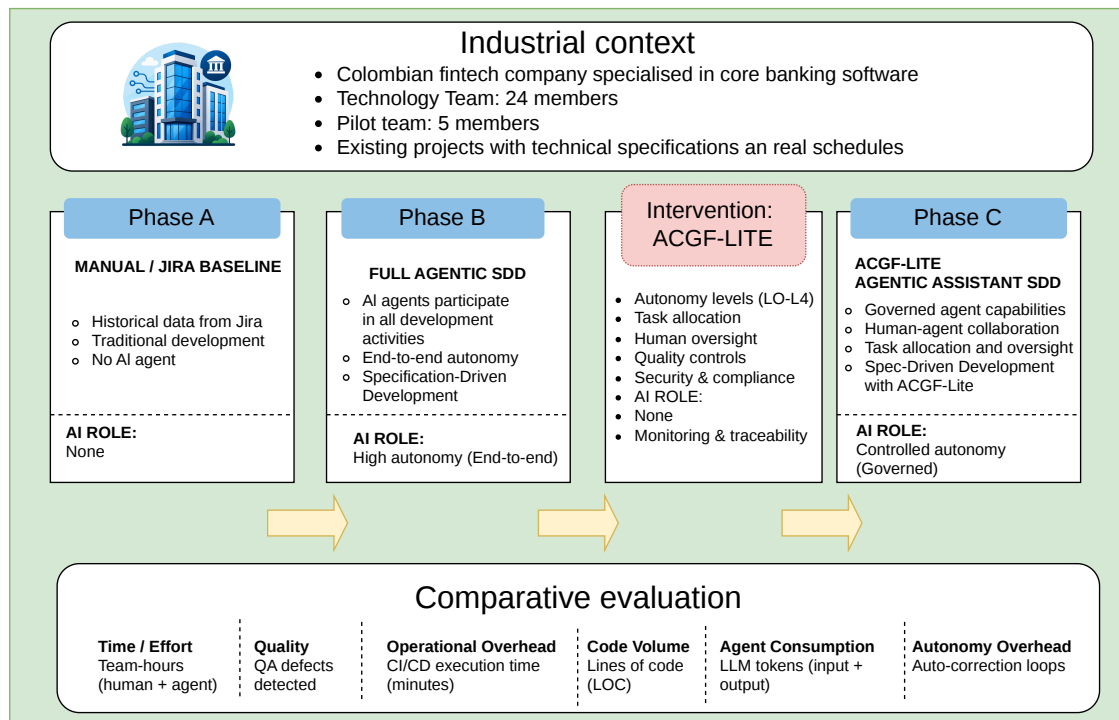


Figure 2: Case study for ACGF-Lite

Where M represents the value of the corresponding metric. A negative value represents a decrease compared to the first phase, while a positive value represents an increase. For metrics with a baseline value of zero, such as token consumption and self-correction cycles, absolute values were reported instead of calculating percentage changes. Additionally, a direct comparison was made between the two agent-assisted development phases to analyze the reduction in token consumption and self-correction cycles achieved using the ACGF-Lite-governed approach.

4.5. Ethical and confidentiality considerations

The case study was conducted with the consent of the participating organization. The company agreed to the use of the collected data for research and publication purposes. The analyzed data included anonymous information and no sensitive employee data. The data were aggregated before analysis to avoid including individual employee productivity information.

5. Results

In this section, we present the results of the framework compared to the proposed case study. Each of these results is shown below.

Figure 3 shows the evaluation results for the framework in the case study. Thus, Figure 3a shows the normalized development results relative to Phase A, where there was no AI intervention. The Full Agentic SDD phase substantially reduced team work hours, reaching approximately 13% of the baseline value. However, this reduction was accompanied by an increase in quality assurance defects, CI/CD execution time, and lines of code. In contrast, ACGF-Lite achieved a more moderate reduction in team work hours, reaching approximately 76.5% of the baseline, while keeping quality assurance defects below the baseline and CI/CD time close to the baseline. Thus, the results suggest that implementing agent autonomy control generates substantial reductions in development effort, as well as in the time it introduces operating costs.

Figure 3b compares agent-related resource consumption and correction overhead between the two agent development phases. The E2E Agents configuration consumed 328,600 tokens and generated 36 auto-correction loops, while ACGF-Lite consumed 144,900 tokens and generated only four correction loops. This represents a 55.9% reduction in token consumption and an 88.9% reduction in correction loops compared to the unconstrained agent configuration.

The reduction in correction loops is particularly relevant because it indicates that ACGF-Lite not only reduced agent resource consumption but also limited repeated auto-correction loops during development.

Full-agent SDD achieved an 87.0% reduction in team work hours but was associated with a 58.8% increase in quality assurance defects. In contrast, ACGF-Lite was associated with a 23.5% reduction in team-hours and an 11.8% reduction in quality assurance defects. Thus, the two approaches occupy distinct regions of the speed-quality space: unconstrained agent development prioritizes development speed, while ACGF-Lite provides a more balanced outcome between development effort and observed quality.

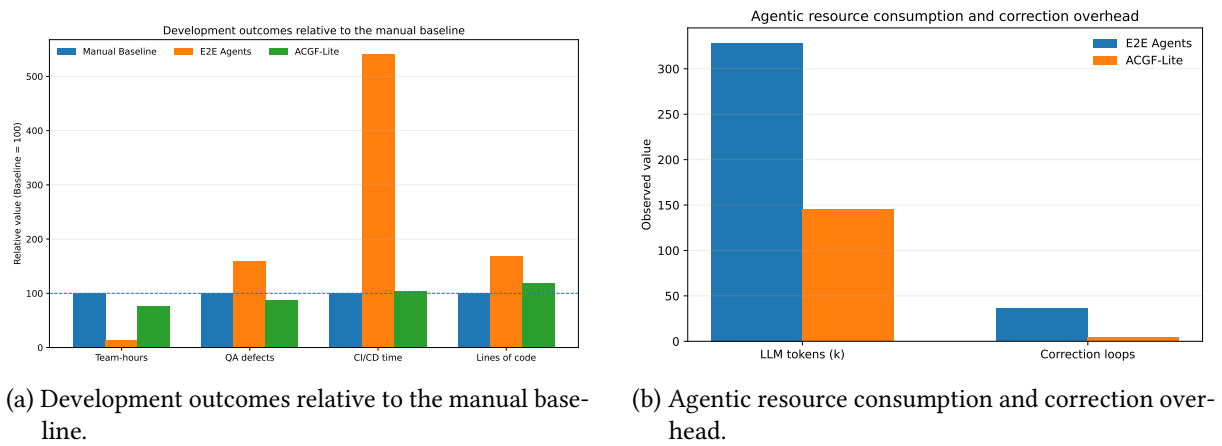


Figure 3: Evaluation results of ACGF-Lite. (a) Development outcomes relative to the manual baseline. (b) Agentic resource consumption and correction overhead.

6. Discussion

In this section, the ACGF-Lite framework is discussed along with the results obtained from the case study. Therefore, it is important to highlight that testing these types of frameworks in a production fintech environment represents a valuable and uncommon opportunity in the literature, given the high complexity and operational risks involved in intervening in **Core Banking**. However, precisely because it is a real-world operation where budgets must be managed and risks to end users mitigated, redundantly duplicating development or indefinitely extending production testing is not feasible. This is the basis for defining the validation structure, acknowledging its potential limitations, in order to have a first formal approach with productive results.

6.1. General discussion

A lower degree of agent autonomy does not necessarily imply lower efficiency; rather, it reflects an intentional redistribution of effort toward human-controlled activities, where unconstrained agent iteration would have generated additional overhead. Therefore, agent governance must consider not only what an agent is allowed to do, but also how much computational interaction is required to complete a task. The large increase in CI/CD execution time observed in the unconstrained agent phase suggests that the development time savings can be partially transferred to subsequent engineering activities. The increase in code volume observed in the unconstrained agent phase is consistent with

the overdesign identified during the pilot project, although the lines of code alone cannot establish an increase in technical debt or a reduction in maintainability. However, the following section details the most relevant scenarios according to Figure 4.

6.2. Software Quality: Defects and Self-Correction Loops

Phase B produced 27 defects, representing a 58.8% increase compared to the control baseline (17 defects) and an 80.0% increase compared to Phase C (15 defects). Ambiguous specifications were interpreted literally by the agents, leading to unintended functional behaviors. This increase is primarily attributed to model hallucinations and the loss of code traceability resulting from unreviewed agent generation. Furthermore, Phase B accumulated 36 self-correction loops, compared to 0 in Phase A and 4 in Phase C. High-complexity modules—such as Messaging (13 loops) and Contract Manager (14 loops)—were particularly susceptible to this behavior. Phase C achieved an 88.9% reduction in cost-impacting loops, demonstrating the vital role of human-in-the-loop oversight and reinforcing the paradigm of agents as assistive tools rather than fully autonomous entities prone to unbounded correction cycles.

6.3. Development Time: Velocity vs. Control

Phase B (Full Agentic) reduced total development time from 1,224 h to 159 h, achieving an 87.0% efficiency gain over the baseline control (traditional workflow). However, Phase C (Agentic Assistant) recorded 936 h, revealing that human review and validation of agent-generated code introduce significant overhead—specifically 777 additional hours relative to the autonomous setup. Nevertheless, Phase C remained 23.0% more efficient than conventional development. This confirms that while autonomous agents maximize raw delivery speed, they do so at the expense of traceability and artifact control, introducing substantial risk into auditable, compliance-driven environments.

6.4. Code Density and Maintainability

Phase B generated 63,571 lines of code (LoC), a 69.2% increase relative to the control baseline (37,571 LoC). This expansion suggests unrefactored code and architectural redundancy, highlighting the agents' lack of critical discernment; LLMs naturally optimize for statistical likelihood at the token level rather than abstract, holistic context evaluation as human engineers do. Phase C produced 44,750 LoC, representing a 29.7% reduction compared to Phase B (though still 19.1% above the control baseline). This outcome indicates that human intervention effectively prunes boilerplate, dead code, and unnecessary bloat.

6.5. Operational Overhead: CI/CD Pipelines and Token Consumption

Cumulative CI/CD pipeline execution time revealed a critical pattern. Phase B required 146 min, which is 5.4 times higher than Phase A (27 min) and Phase C (28 min). This surge occurred because autonomous agents repeatedly generated redundant unit tests and bloated code across self-correction iterations. This evidence validates that deterministic, human-guided pipelines substantially reduce CI/CD execution overhead without precluding the strategic use of AI agents. Regarding computational expenditure, Phase B consumed 328.6 K tokens, whereas Phase C reduced usage to 144.9 K tokens, achieving a 55.9% cost reduction. This decrease directly correlates with the reduction of self-correction loops and the execution of deterministic pipelines that avoid redundant LLM invocations with full context windows. In Phase C, targeted and precise prompts allowed developers to guide code generation, eliminating unconstrained agent iterations and superfluous code synthesis.

6.6. Limitations of case study

The empirical evaluation of governance frameworks in real-world industrial environments entails methodological compromises stemming from the operational and business constraints of the fintech sector. For this reason, evaluating five critical services across three phases over approximately three

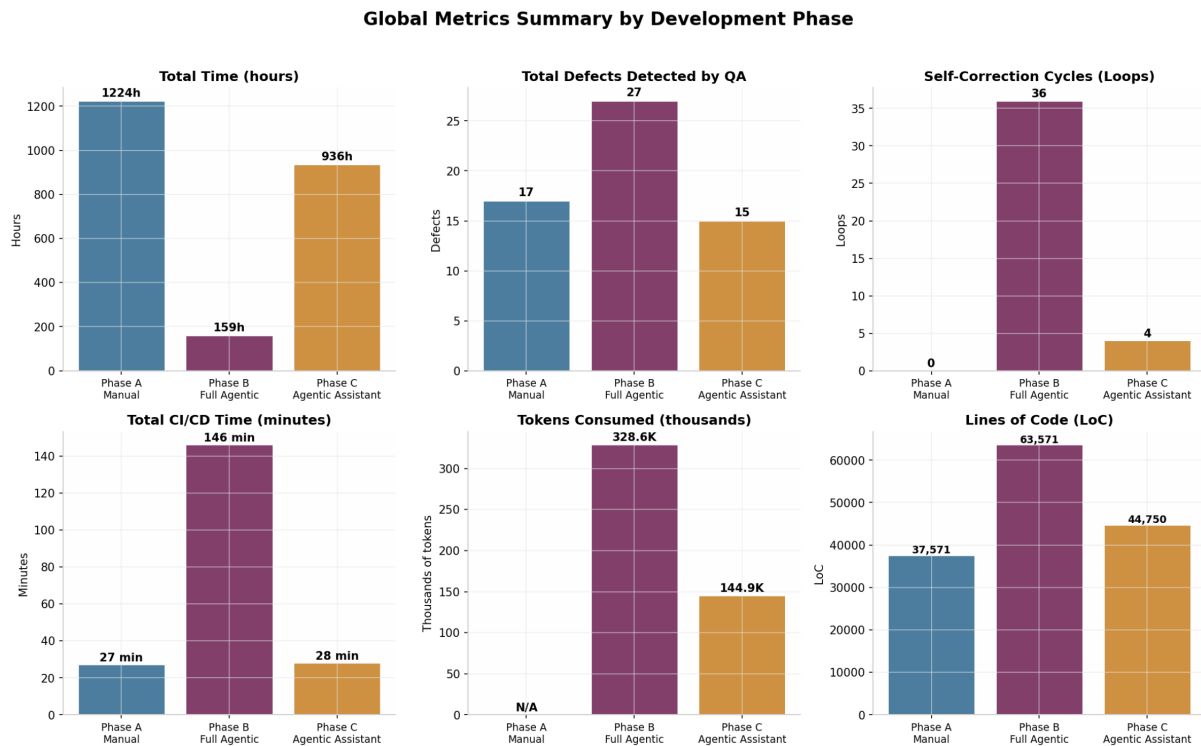


Figure 4: Summary overview of the three evaluated phases

months to obtain the necessary data for analysis constitutes a practical initial evaluation for this stage of the study. However, it is expected that using the framework will allow for the collection of more data across more projects in order to evaluate the framework and generalize its findings. It is also expected that technical debt metrics will be used without compromising the stability of the company’s projects. This latter point is a significant limitation for other industries or organizations requiring this type of implementation without disrupting their operations.

6.7. Principal limitations of ACGF-Lite

Governance definitions must be generated by a human. If they are constructed by an agent, the security and auditability criteria may not be met. Another limitation of ACGF-Lite is that its governance model focuses primarily on agent autonomy, task assignment, and human oversight, without fully specifying how these controls should be applied when agents interact with external development and operations systems. In particular, the current framework does not fully define system-specific permissions, tool invocation policies, cross-system authorization, or the governance of side effects on platforms such as source code repositories, issue tracking systems, CI/CD pipelines, and cloud infrastructure.

7. Future work

The following directions are proposed for future work.

- **Literature review.** A literature review article is expected to be drafted to further detail the topics in the related work section. This review will offer a perspective on the key findings regarding agent-based software development.
- **Governance for deploying the solution.** The framework doesn’t clarify how to generate definitions for deploying different applications. Therefore, we could expand the framework to define whether the agent can deploy the application or in what critical cases a human must perform the deployment and verify if the agent’s solution is appropriate.

- **Usage of capabilities or skills.** This framework will be enhanced with a skills governance model that defines how agent capabilities are assigned and restricted based on task risk and autonomy levels. This enhancement will explicitly establish which skills can be possessed, autonomy levels, data classifications, and access authorizations, allowing specific capabilities to be activated or deactivated depending on the context of each task.
- **Evaluate regulatory frameworks.** We want to explore other regulatory frameworks to strengthen the existing one. Perhaps we will explore mechanisms to ensure that regulatory requirements remain applicable independently of ACGF-Lite.

8. Conclusions

This manuscript presented a framework for establishing governance when interacting with agents during source code generation, particularly in fintech environments, to support software development lifecycle tasks. The results suggest that ACGF-Lite provides a more balanced trade-off between development effort, observed quality, and agent-related overhead. Thus, regarding the phases of the case study, we have the following conclusions for phase B: code traceability is lost; code redundancy is also present. Loops consume excessive tokens. Specs must be very well defined to avoid hallucinations and reprocessing. Agents operating autonomously accelerate development but add extra costs and increase the project's learning curve. Agents remove control from developers and create black boxes and injection risks. Furthermore, leaving agents to handle pipelines for CI/CD, code smell, linters, base structures, patterns, security reviews, and other deterministic processes increases both cost and time. All deterministic processes become conventional pipelines if the agents are executed as commands or can be run manually (they are .sh files). In phase C, the cost is reduced by using the framework and the pipelines/cookie cutter for deterministic processes in the project development. In this case, the artifact generation time increases, considering that the developer now has more control over the code (reviewing and adjusting what the agents generate). Additionally, the overhead that self-correcting loops can generate is controlled. Furthermore, CI/CD time is regulated since work is done on specific unit tests and deterministic pipelines are executed for code smell review, linters, base structures, patterns, and security review. On the other hand, in pre-commit processes, which is before uploading the source code to the Git server, there is more control over the generated code, which increases auditability. It doesn't make sense to have the agent audit what it itself generated, and having an auditing agent increases the cost by having to rescan the entire project, plus the risk of hallucinations, black box issues, reproducibility problems, etc.

Declaration on Generative AI

During the preparation of this work, the authors used GPT-5.6 Luna and Grammarly to check grammar and spelling. Further, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

- [1] N. Forsgren, M.-A. Storey, C. Maddila, T. Zimmermann, B. Houck, J. Butler, The space of developer productivity: There's more to it than you think., *Queue* 19 (2021) 20–48.
- [2] J. Hernandez, D. H. Peluffo-Ordoñez, H. Florez, How do autonomous software agents learn? a systematic literature review, *PeerJ Computer Science* 12 (2026) e3756.
- [3] A. E. Hassan, H. Li, D. Lin, B. Adams, T.-H. Chen, Y. Kashiwa, D. Qiu, Agentic software engineering: Foundational pillars and a research roadmap, *arXiv preprint arXiv:2509.06216* (2025).
- [4] European Parliament and Council of the European Union, Regulation (eu) 2022/2554 on digital operational resilience for the financial sector, *Official Journal of the European Union*, 2022. URL: <https://eur-lex.europa.eu/eli/reg/2022/2554/oj>.

- [5] PCI Security Standards Council, Payment card industry data security standard: Requirements and testing procedures, version 4.0, 2022.
- [6] European Parliament and Council of the European Union, Regulation (eu) 2016/679: General data protection regulation, Official Journal of the European Union, 2016. URL: <https://eur-lex.europa.eu/eli/reg/2016/679/oj>.
- [7] B. Gülmez, Code generation with large language models: a survey from neural program synthesis to autonomous software development, *Applied Intelligence* 56 (2026). doi:10.1007/s10489-026-07230-0, scopus EID: 2-s2.0-105035919185.
- [8] J. Hernandez, M. F. Pollo-Cattaneo, D. H. Peluffo-Ordóñez, H. Florez, Cognitive architecture for learning in autonomous agents: A study in discrete navigation, in: *Communications in Computer and Information Science*, Springer, 2025, pp. 3–19. URL: https://link.springer.com/10.1007/978-3-032-07175-0_1. doi:10.1007/978-3-032-07175-0_1.
- [9] S. Liu, Q. Guo, X. Liu, Y. Liu, Mitigating cognitive vulnerabilities in code generation via multi-agent adversarial debate, in: *WWW 2026 - Proceedings of the ACM Web Conference 2026*, 2026, pp. 7412–7420. doi:10.1145/3774904.3792557, scopus EID: 2-s2.0-105038565523.
- [10] Z. Chen, L. Lin, X. Lin, Y. Chen, L. Wang, Levels of automated code generation (lacg): An operational taxonomy for ai-augmented software construction, *Applied Sciences (Switzerland)* 16 (2026). doi:10.3390/app16104788, scopus EID: 2-s2.0-105040125701.
- [11] T. Kehkashan, M. Abdullah, A. S. Al-Shamayleh, N. Ivković, N. A. Ismail, S. S. S. Ahmad, A. Rehman, A. Akhunzada, From benchmarks to deployment: a comprehensive review of agentic ai evaluation, *Artificial Intelligence Review* 59 (2026). doi:10.1007/s10462-026-11571-0, scopus EID: 2-s2.0-105041907209.
- [12] L. A. Costa, A. Fontão, R. P. d. Santos, A. Serebrenik, Applying generative artificial intelligence for vulnerability fixing in a proprietary software ecosystem, *Journal of Systems and Software* 234 (2026). doi:10.1016/j.jss.2025.112723, scopus EID: 2-s2.0-105024758998.
- [13] S. Karuppuchamy, Governed agentic ai for software platforms: A reference architecture for safe autonomy at scale, in: *Conference Proceedings - IEEE SOUTHEASTCON*, 2026. doi:10.1109/SoutheastCon63549.2026.11475963, scopus EID: 2-s2.0-105037800300.
- [14] M. Padubrin, A. C. Kulzer, E. Guerocak, Multi-agent llm architecture for systems engineering automation, in: *SAE Technical Papers*, 2026. doi:10.4271/2026-01-0777, scopus EID: 2-s2.0-105044709980.
- [15] M. Tuape, Y. Gabrielmichael, J. Kasurinen, Architecting trust: Designing human oversight and accountability for ai-driven software engineering under the eu ai act, in: *Proceedings - 2025 13th International Conference in Software Engineering Research and Innovation, CONISOFT 2025*, 2025, pp. 325–331. doi:10.1109/CONISOFT66928.2025.00048, scopus EID: 2-s2.0-105034779891.
- [16] A. Doshi, Y. Hong, C. Xu, E. Kang, A. Kapravelos, C. Kästner, Towards verifiably safe tool use for llm agents, in: *Proceedings - 2026 IEEE/ACM 48th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER 2026*, 2026, pp. 201–205. doi:10.1145/3786582.3786839, scopus EID: 2-s2.0-105044883200.
- [17] Y.-S. Hsia, F. Yu, J.-H. R. Jiang, Neuro-symbolic compliance: Integrating llms and smt solvers for automated financial legal analysis, in: *Proceedings - 2025 2nd IEEE/ACM International Conference on AI-powered Software, AIware 2025*, 2025, pp. 19–28. doi:10.1109/AIware69974.2025.00010, scopus EID: 2-s2.0-105035194797.
- [18] R. Osuagwu, T. Cook, M. Masoud, K. Ghosal, R. Mattivi, Scalecall agentic tool calling at scale for fintech: Challenges, methods, and deployment insights, in: *2025 3rd International Conference on Foundation and Large Language Models, FLLM 2025*, 2025, pp. 210–217. doi:10.1109/FLLM67465.2025.11390878, scopus EID: 2-s2.0-105035860228.
- [19] M. Trifiletti, Strategies to govern the imperative of artificial intelligence in payment systems: Integrating trust, security and european regulatory compliance, *Journal of Payments Strategy and Systems* 19 (2025) 328–344. doi:10.69554/IVAD7481, scopus EID: 2-s2.0-105026577732.
- [20] P. K. Joshi, Achieving pci-dss compliance in payment gateways: A comprehensive approach, *Journal of Technology and Systems* 6 (2024) 13–31. doi:10.47941/jts.2299.

- [21] P. S. Merchant, S. Gaba, Verifiable collaboration among agentic code assistants using blockchain for software engineering workflows, in: Proceedings - 2025 International Conference on AI x Software Engineering, AIXSE 2025, 2025, pp. 87–90. doi:10.1109/AIXSE64906.2025.00019, scopus EID: 2-s2.0-105038347948.