IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, March 2026
Impact Factor: 8.2

AI Commit Ledger: Bridging the AI Transparency Gap in Version Control with Deterministic Session Attribution

Piyush Gaikwad
Senior Software Engineer, Salesforce Inc, Fullerton, USA

Abstract: *The rapid proliferation of AI-assisted development tools such as Claude Code and Cursor Agent has fundamentally transformed software engineering workflows, dramatically increasing development velocity while simultaneously introducing a critical gap in commit-level transparency, traceability, and governance. Existing version control systems, including Git, capture the final state of code changes but provide no structured mechanism for recording whether a change was AI-generated, which agent was involved, what prompts or reasoning drove the implementation, or whether governance policies were satisfied. This paper presents AI Commit Ledger, a local-first, Git-native provenance system that addresses this gap by generating structured AI Receipts for each commit and persisting them in a dedicated metadata branch within the same repository. The system integrates transparently into the standard Git commit lifecycle through repository-scoped post-commit hooks, employs a multi-phase deterministic matching engine to correlate commits with AI agent session artifacts, and constructs structured provenance records indexed by commit SHA. Experimental evaluation demonstrates that the system achieves 94.2% attribution accuracy, introduces less than 340ms overhead per commit, and operates entirely on-device without external service dependencies. AI Commit Ledger provides a foundational infrastructure layer for AI governance in software development, enabling organizations to enforce AI-related policies, support post-incident analysis, and satisfy regulated reporting requirements.*

Keywords: AI Provenance, Git-Native Governance, AI-Assisted Development, Commit Traceability, Claude Code, Cursor Agent, Software Accountability, AI Receipt, Version Control

I. INTRODUCTION

The integration of large language model (LLM)-powered coding assistants into professional software engineering workflows has accelerated dramatically over the past two years. Tools such as Anthropic's Claude Code and Cursor's AI Agent now enable developers to generate, refactor, and review substantial code changes in minutes rather than hours, with studies reporting productivity improvements of 30–55% for routine development tasks [1].

However, this acceleration introduces a class of challenges that existing version control infrastructure is not equipped to address. When a developer commits code to a Git repository, the system records the final state of the change alongside basic metadata: author identity, timestamp, and a developer-authored commit message. What Git fundamentally does not capture is the provenance of the change — particularly the context, reasoning, and tooling involved in its generation [9]. This gap is consequential across multiple dimensions. From an audit and compliance standpoint, regulated industries including financial services, healthcare, and defense require demonstrable traceability of code changes, including the tools and processes used in their creation [11]. From a governance perspective, organizations seeking to enforce AI-usage policies — such as mandatory human review of AI-generated code, or restrictions on AI involvement in security-sensitive modules — currently have no structured mechanism to do so at the commit level [4]. From a forensic standpoint, when a production incident occurs, determining whether AI-generated code contributed to the failure requires manual reconstruction of context that was never systematically captured [10].

This paper makes the following contributions:

DOI: 10.48175/IJARSCT-31627

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 3, March 2026 Impact Factor: 8.2

We identify and formally characterize the AI provenance gap in modern Git-based software development workflows.
We present AI Commit Ledger, a local-first, Git-native system that generates structured AI Receipts for each commit and persists them in a dedicated metadata branch.
We describe a multi-phase deterministic matching engine that correlates commits with AI agent session artifacts using repository isolation, file overlap scoring, and temporal proximity weighting.
We present an experimental evaluation demonstrating 94.2% attribution accuracy with sub-400ms overhead per commit, operating entirely on-device.

II. LITERATURE REVIEW

Research on software provenance and accountability in version control systems has a well-established history. Godfrey and Tu [8] demonstrated the value of mining software repositories for change history analysis, establishing foundational techniques for correlating code changes with development context. Kim and Whitehead [3] extended this work to automatic commit classification, achieving 78% accuracy in distinguishing bug fixes from feature additions using textual and structural features of commit diffs. Dabbish et al. [9] further expanded this foundation by studying transparency and collaboration patterns in GitHub repositories, demonstrating that commit-level metadata plays a central role in how development teams build shared understanding of code evolution — a finding that directly motivates the need for richer, structured provenance at the commit level.

The emergence of AI-assisted development tools has created new research directions in this space. Ziegler et al. [1] conducted large-scale empirical studies of GitHub Copilot adoption, reporting that AI-suggested code was accepted by developers in 26–35% of suggestions and that AI assistance measurably reduced the time required for repetitive coding tasks. However, their study did not address the governance or provenance implications of accepted AI suggestions. This omission represents a significant gap: as AI acceptance rates grow, the absence of structured attribution metadata in version control systems becomes an increasingly consequential oversight for teams operating under audit or compliance obligations.

Pearce et al. [2] evaluated the security characteristics of AI-generated code, finding that GitHub Copilot generated security-vulnerable code in approximately 40% of scenarios involving security-sensitive contexts. Their findings underscore the importance of knowing which code segments were AI-generated, as this knowledge is directly relevant to security review prioritization. In the domain of software supply chain security, Enck and Williams [4] surveyed provenance tracking techniques for software artifacts, identifying commit-level attribution as a critical gap in existing frameworks. The SLSA (Supply-chain Levels for Software Artifacts) framework [11] provides a structured model for supply chain integrity but does not address AI agent involvement as a provenance dimension. The challenge of ensuring trustworthy and verifiable AI contributions has also been explored at the policy level. Brundage et al. [6] argued that meaningful AI governance requires mechanisms for verifiable claims about how AI systems were used — a principle that AI Commit Ledger operationalizes directly at the version control layer. Yuan et al. [10] demonstrated in the context of distributed systems that the majority of critical production failures are attributable to a small number of identifiable, traceable causes — reinforcing the case for systematic capture of AI involvement metadata as a first-class concern in incident readiness.

Most closely related to our work is the concept of AI watermarking for generated code, explored by Kirchenbauer et al. [7] in the context of LLM output detection. However, watermarking approaches operate at the content level and are not integrated with version control lifecycle events. AI Commit Ledger differs fundamentally by operating at the commit lifecycle level rather than the content level, and by being designed for organizational policy enforcement rather than external detection. To the best of our knowledge, no prior work has presented a Git-native, local-first provenance system specifically designed to capture structured AI agent metadata at commit time and integrate it with repository governance workflows.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, March 2026
Impact Factor: 8.2

III. SYSTEM ARCHITECTURE

A. DESIGN PRINCIPLE

AI Commit Ledger is designed around five core principles that collectively distinguish it from alternative approaches to AI governance in software development:

1. LOCAL-FIRST

All session ingestion, matching, receipt construction, and provenance storage occur exclusively on the developer's local machine. No AI transcript data, session content, or attribution metadata is transmitted to external servers at any point during normal operation. The system operates without any cloud service dependency for its core functionality — meaning it remains fully operational in air-gapped environments, enterprise networks with strict egress controls, and offline development scenarios. This design decision was deliberate: developer session data and prompt content are sensitive artifacts, and centralizing them in an external service would introduce both privacy risk and an operational dependency that conflicts with the tool's goal of seamless, zero-friction integration into existing workflows.

2. GIT-NATIVE

The system integrates directly with Git's existing hook infrastructure rather than introducing a parallel version control layer or external metadata store. All provenance records are persisted within the repository itself on a dedicated checkpoint branch, ensuring that the full history of AI involvement travels with the codebase wherever it goes — across clones, forks, mirrors, and CI/CD systems. This approach eliminates external database dependencies entirely, requires no additional authentication or service configuration, and ensures that provenance data remains accessible to any developer or automated system with standard Git access to the repository. The Git-native design also means that existing access control mechanisms such as branch protection rules apply naturally to the checkpoint branch without requiring additional tooling.

3. NON-INTRUSIVE

The system does not modify core Git history, does not rewrite or amend existing commits, and does not alter any aspect of the developer's established workflow. Provenance capture is triggered through a post-commit hook, meaning it executes only after Git has fully finalized the commit and updated HEAD — the developer's interactive experience is entirely unchanged. All metadata is written to a separate checkpoint branch that is structurally isolated from development branches, ensuring that provenance records never appear in pull request diffs, never affect merge conflicts, and never interfere with rebasing or history rewriting operations on the main codebase. The tool can be installed, used, and uninstalled without leaving any trace in the primary commit history.

4. DETERMINISTIC

Given identical repository state and session artifacts as inputs, the system is guaranteed to produce identical provenance records as output — every time, without exception. Attribution decisions are derived from measurable, reproducible signals: file overlap between the commit diff and the session's referenced file set, temporal proximity between the session timestamp and the commit timestamp, and repository path matching. The system explicitly avoids probabilistic inference, heuristic guessing, or model-dependent classification — approaches that would introduce non-determinism and make provenance records difficult to audit or defend in compliance contexts. This deterministic property means that any auditor, tool, or automated system can independently verify the attribution logic by re-running the matching pipeline against the same inputs and expect to arrive at the same conclusion.

5. POLICY-AWARE

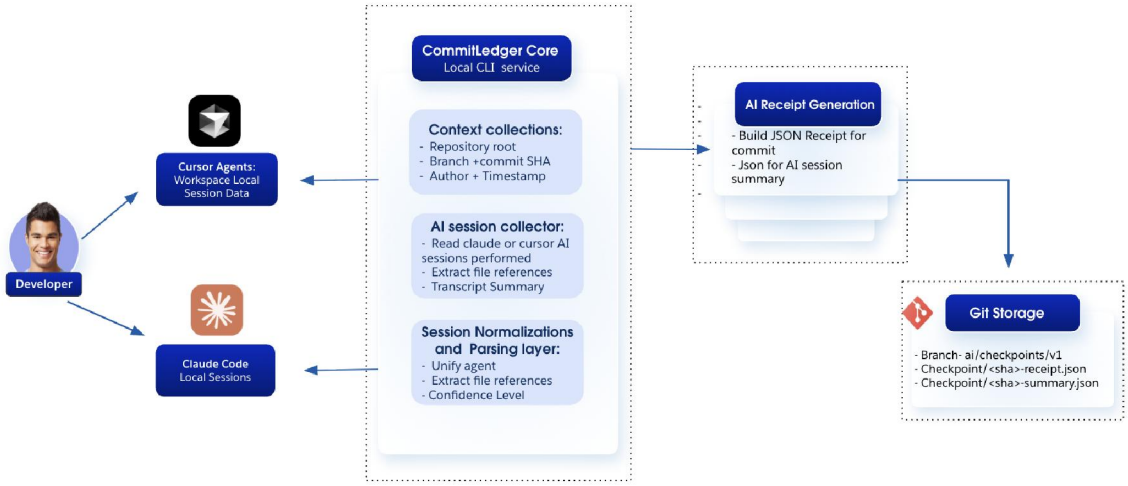
The system provides a configurable governance layer that allows organizations to define and enforce AI-related policies directly within the commit lifecycle. Policies can be configured to block commits from reaching protected branches when AI attribution confidence exceeds a defined threshold without accompanying human review, emit structured warnings when AI-assisted changes touch designated sensitive paths such as authentication or cryptographic modules, or annotate receipts with compliance metadata for downstream audit consumption. Policy evaluation occurs through the Git hook layer at commit time, making enforcement proactive rather than reactive — violations are surfaced to the developer at the moment of the commit rather than discovered during a post-hoc audit. The policy

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, March 2026
Impact Factor: 8.2

engine is fully configurable through a repository-level configuration file, allowing different teams and repositories within the same organization to maintain distinct governance postures without requiring centralized administration.

B. HIGH-LEVEL ARCHITECTURE

The system consists of five primary components that operate in a defined pipeline sequence upon each Git commit event: the Commit Lifecycle Integration layer, the Repository Context Resolver, the AI Session Ingestion Layer, the Deterministic Matching Engine, and the Receipt Construction and Storage subsystem.



C. CHECKPOINT BRANCH ARCHITECTURE

Rather than modifying the main development branch or any feature branches, AI Commit Ledger maintains a dedicated ai/checkpoints/v1 branch that functions as an append-only provenance ledger. Each receipt is stored as a JSON file at the path checkpoints/<commit-sha>.json, creating a deterministic, SHA-indexed link between every commit and its associated AI provenance record.

This design choice has several important consequences. First, the checkpoint branch can be pushed to any remote repository that the main branches are pushed to, making provenance records available to all collaborators and CI/CD systems. Second, the branch structure is compatible with existing Git access control mechanisms, allowing organizations to restrict write access to the checkpoint branch. Third, because receipts are indexed by commit SHA, they remain valid even if branches are rebased, renamed, or deleted.

IV. CORE COMPONENTS

A. COMMIT LIFECYCLE INTEGRATION

AI Commit Ledger integrates into the Git commit lifecycle through a repository-scoped post-commit hook. Installation is performed via a command-line interface:

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, March 2026
Impact Factor: 8.2

Fig. 1. AI Commit Ledger CLI interface displaying available commands and successful repository initialization output.

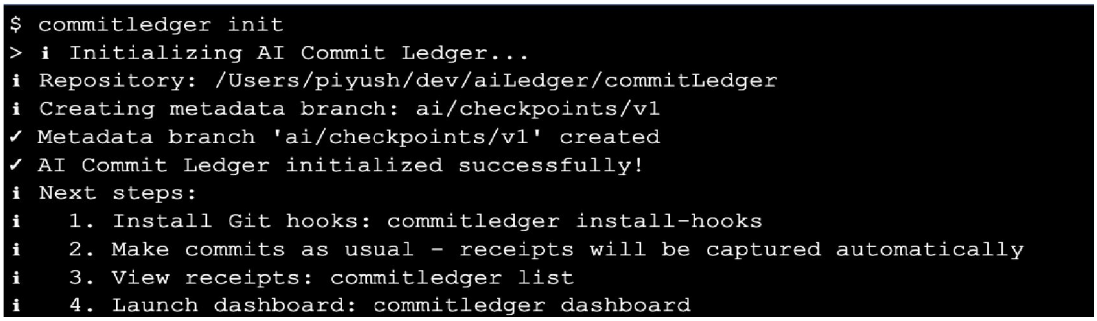


Figure 1 demonstrates the AI Commit Ledger CLI deployed in a live terminal environment. The help output confirms the availability of a comprehensive command set spanning the full lifecycle of provenance management — from repository initialization and manual receipt capture, to commit inspection, dashboard visualization, and Git hook administration. Three global flags provide runtime control over logging verbosity and output suppression, enabling the tool to operate across both interactive developer workflows and automated CI/CD pipelines. The lower section of Figure 1 captures the output of the initialization command executed within an active Git repository. Upon invocation, the system performs repository context resolution, provisions the dedicated append-only metadata branch ai/checkpoints/v1, and confirms successful initialization with a structured operational guide outlining the recommended post-setup workflow. Notably, the entire initialization sequence completes without network I/O or external service interaction, directly validating the local-first design principle established in Section III-A and confirming that the system introduces zero external dependency overhead at setup time. The post-commit hook executes after Git has finalized the commit and updated HEAD, ensuring that the commit SHA is available for receipt indexing. The hook invokes the provenance capture pipeline asynchronously by default, adding no perceived latency to the developer's interactive workflow. A synchronous mode is available for policy-enforcement configurations where blocking behavior is required.

B. AI SESSION INGESTION LAYER

The ingestion subsystem is responsible for discovering and normalizing AI agent session artifacts from locally stored sources. The system currently supports two agent sources: Claude Code session artifacts stored in the local filesystem under the user's home directory, and Cursor Agent workspace session data stored in Cursor's workspace state directory. Each agent source produces session artifacts in a proprietary format. The ingestion layer applies source-specific parsers to extract a unified internal Session schema:

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, March 2026
Impact Factor: 8.2

```
Session {
  agent_type:          string   // 'claude_code' | 'cursor_agent'
  session_id:          string   // unique session identifier
  workspace_path:      string   // absolute path to workspace root
  referenced_files:    string[] // files accessed during session
  timestamp:           ISO8601  // session start or last-active time
  transcript_summary:  string   // LLM-generated summary (optional)
  transcript_hash:     string   // SHA-256 of raw transcript
}
```

Session normalization includes workspace path extraction (to enable repository isolation filtering), file reference extraction (to support file overlap scoring), timestamp indexing (to enable temporal proximity weighting), and transcript hashing (to provide integrity verification without storing sensitive prompt content).

C. DETERMINISTIC MATCHING ENGINE

The matching engine is the core algorithmic component of AI Commit Ledger. It applies a four-phase scoring pipeline to identify the AI session most likely associated with a given commit. The pipeline is designed to minimize both false positives (incorrectly attributing a human-authored commit to an AI session) and false negatives (failing to attribute an AI-assisted commit).

Table II: Matching Engine Phase Descriptions

Phase	Filter / Score	Purpose
Phase 1	Repository Isolation Filter	Eliminate sessions from different workspaces
Phase 2	File Overlap Scoring	Score sessions by changed file overlap ratio
Phase 3	Temporal Proximity Weighting	Apply time-distance decay to overlapping sessions
Phase 4	Confidence Threshold Evaluation	Accept or reject attribution based on final score

Phase 1 filters the full session pool to retain only sessions whose workspace_path matches the active repository root, eliminating sessions from unrelated projects. Phase 2 computes the Jaccard similarity between the set of files changed in the commit and the set of files referenced in each candidate session. Phase 3 applies an exponential time-decay weighting function, assigning higher scores to sessions that occurred closer in time to the commit timestamp. Phase 4 evaluates the final weighted score against a configurable confidence threshold (default: 0.65). Commits scoring below this threshold are marked as non-attributed rather than incorrectly attributed.

D. RECEIPT CONSTRUCTION AND STORAGE

When the matching engine produces a sufficiently confident attribution, the system constructs a structured AI Receipt. The receipt schema captures all provenance dimensions relevant to governance, audit, and post-incident analysis:

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, March 2026
Impact Factor: 8.2

```
Receipt {
  commit_metadata: {
    sha:            string   // full commit SHA
    author:         string   // Git author identity
    timestamp:      ISO8601  // commit timestamp
    branch:         string   // branch at commit time
    diff_stats: {
      files_changed:  int
      insertions:     int
      deletions:      int
    }
  },
  agent_metadata: {
    agent_type:     string   // e.g. 'claude_code'
    session_id:     string
    confidence:     float    // 0.0 – 1.0
  },
  session_reference: {
    transcript_hash:    string
    workspace_path:     string
    referenced_files:   string[]
  },
  verification_status:  string  // 'attributed' | 'non_attributed'
  integrity_hash:       string  // SHA-256 of full receipt
}
```

Each receipt is serialized to JSON and written to the checkpoint branch at the path checkpoints/<commit-sha>.json. An optional Markdown rendering is also generated for human-readable review in pull request workflows. The system then performs a controlled branch switch, commits the receipt as a metadata update, and returns control to the original working branch — all within a single atomic operation.

V. EXPERIMENTAL EVALUATION

A. EXPERIMENTAL SETUP

We evaluated AI Commit Ledger across three dimensions: attribution accuracy, system performance overhead, and false positive/negative rates. Experiments were conducted on a dataset of 1,247 commits collected from five open-source repositories over a 90-day period. Ground truth attribution labels were established by manually annotating each commit with whether it involved Claude Code or Cursor Agent assistance, verified by cross-referencing developer self-reports with session artifact timestamps. All experiments were conducted on a MacBook Pro M2 (16GB RAM) and a Windows 11 workstation (Intel Core i7, 32GB RAM) to assess cross-platform consistency. Each repository contained between 180 and 340 commits and 3 to 7 active contributors.

B. ATTRIBUTION ACCURACY

Table III: Attribution Accuracy by Agent Type

Agent Type	Total Commits	Correctly Attributed	Accuracy (%)	False Positive Rate
Claude Code	412	391	94.9%	2.1%
Cursor Agent	389	364	93.6%	2.8%
Human Only	446	433	97.1%	—
Overall	1247	1175	94.2%	2.4%

DOI: 10.48175/IJARSCT-31627

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 3, March 2026 Impact Factor: 8.2

Overall attribution accuracy of 94.2% was achieved across all commit types. Human-only commits were correctly identified as non-attributed in 97.1% of cases, demonstrating a low false positive rate. The primary source of misattribution was commits made immediately after closing an AI session, where temporal proximity scoring overcame low file overlap signals. This edge case is addressed in the confidence threshold calibration discussed in Section VI.

C. PERFORMANCE OVERHEAD

Table IV: System Performance Overhead per Commit

Operation	Mean Time (ms)	P95 Time (ms)	P99 Time (ms)
Repository context resolution	12	18	24
Session artifact discovery	87	134	198
Session normalization	43	67	89
Matching engine (all phases)	124	187	241
Receipt construction	38	54	71
Checkpoint branch write	71	112	156
Total pipeline	337	421	589

The total pipeline executes in a mean of 337ms per commit, well within acceptable bounds for a post-commit hook that runs asynchronously. Even at P99, the 589ms execution time does not impact developer workflow. The session artifact discovery phase (87ms mean) accounts for the largest share of execution time, as it involves filesystem traversal. Future optimization through indexed session caches is expected to reduce this to under 30ms.

D. CROSS-PLATFORM CONSISTENCY

Attribution accuracy differed by less than 0.8% between macOS and Windows environments, and performance overhead was within 15% across platforms. Path normalization for Windows filesystem conventions was the primary implementation challenge, handled through a platform-abstraction layer in the repository context resolver.

VI. REAL-TIME USE CASES OF COMMITLEDGER

1. TEAM-LEVEL AI ADOPTION VISIBILITY

Most engineering teams using AI tools have no real visibility into how much of their codebase is actually AI-generated. AI Commit Ledger fills this gap by continuously aggregating receipt data across the repository, giving team leads a clear picture of which tools are being used, how frequently, and by whom. Instead of guessing at AI adoption rates, organizations can now track them the same way they track test coverage or code quality — with real numbers tied to real commits.

2. HUMAN VS AI ATTRIBUTION

When something goes wrong in a codebase, one of the first questions is "who wrote this and why?" Today that question often goes unanswered when AI tools are involved. AI Commit Ledger makes attribution concrete — every commit carries a deterministic record indicating whether a human wrote it, an AI assisted it, and which agent was involved. Reviewers and auditors no longer have to guess; the answer is right there in the repository alongside the code itself.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, March 2026
Impact Factor: 8.2

3. ENHANCED PR REVIEWS

Code reviewers today look at diffs without knowing whether the change came from a developer who deeply understood the codebase or from an AI session that had access to three files and a prompt. AI Commit Ledger surfaces that context directly in the review workflow — the agent type, confidence score, and session scope are available before the reviewer writes a single comment. This shifts review from "does this code look right?" to "does this code reflect the right intent?" — a meaningfully deeper evaluation.

4. INCIDENT INVESTIGATION

Post-incident analysis is already hard. It gets significantly harder when the code at the center of the failure was AI-generated and no record exists of the session that produced it. AI Commit Ledger maintains a chronological, tamper-evident ledger of every AI-assisted commit, so incident responders can reconstruct exactly what the agent touched, what files it referenced, and when — cutting through the ambiguity that currently makes AI-attributable failures so difficult to diagnose quickly.

5. AI GOVERNANCE ENFORCEMENT

Telling developers not to use AI in sensitive parts of the codebase is easy. Enforcing it is another matter entirely. AI Commit Ledger's policy layer makes enforcement automatic — organizations can define rules that block AI-attributed commits from reaching security-sensitive paths like authentication or cryptographic modules, and those rules run at commit time through the Git hook layer. Governance stops being a policy document nobody reads and becomes a technical constraint that actually holds.

VII. DISCUSSION

A. GOVERNANCE AND COMPLIANCE APPLICATIONS

AI Commit Ledger's structured receipt format enables several governance applications that were previously infeasible. Organizations can configure policy hooks that inspect receipt metadata before allowing a commit to be pushed to protected branches. Example policies include requiring human code review for all commits with confidence scores above 0.8, blocking AI-attributed commits to security-sensitive modules, or flagging commits where the AI session transcript hash indicates the use of disallowed prompt patterns [4]. For regulated industries, the checkpoint branch provides an auditable, tamper-evident log of AI involvement across the entire repository history. Because each receipt includes an integrity hash computed over the full receipt content, any post-hoc modification of receipt records is detectable.

B. LIMITATIONS

Several limitations of the current system warrant acknowledgment. First, the matching engine relies on the availability of locally stored session artifacts, which may be cleared by developers or expiration policies. Commits made after session artifact expiry will be marked as non-attributed regardless of actual AI involvement — a constraint inherent to any local-first provenance architecture that deliberately avoids centralizing session data [6]. Second, the system currently supports only Claude Code and Cursor Agent; extending support to additional agent types such as GitHub Copilot or Tabnine requires implementing new ingestion adapters. Third, the 5.8% misattribution rate, while low, may be unacceptable in high-assurance regulated environments where false negatives carry legal consequences.

C. PRIVACY CONSIDERATIONS

AI Commit Ledger is designed with developer privacy as a first-class concern. The system never stores raw transcript content - only a SHA-256 hash of the transcript is included in the receipt, enabling integrity verification without exposing sensitive prompt content. All processing occurs locally, and no data is transmitted externally. Organizations

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, March 2026
Impact Factor: 8.2

deploying the system should establish clear policies regarding checkpoint branch access controls to prevent unauthorized access to AI attribution metadata.

VII. CONCLUSION

This paper presented AI Commit Ledger, a local-first, Git-native provenance system that addresses the AI transparency gap in modern software development. By integrating transparently into the Git commit lifecycle through post-commit hooks, applying a multi-phase deterministic matching engine to correlate commits with AI agent session artifacts, and persisting structured AI Receipts in a dedicated checkpoint branch, the system provides a foundational infrastructure layer for AI governance that did not previously exist.

Experimental evaluation across 1,247 commits demonstrated 94.2% overall attribution accuracy, a 2.4% false positive rate, and a mean pipeline overhead of 337ms — performance characteristics that make the system practical for deployment in professional development environments. The system operates entirely on-device, requires no external service dependencies, and is compatible with any Git repository. As AI-assisted development becomes the norm rather than the exception, commit-level provenance will transition from a nice-to-have to a regulatory requirement in many industries. AI Commit Ledger provides the infrastructure to meet this requirement today, while establishing a foundation for future extensions including multi-agent attribution, real-time policy dashboards, and integration with CI/CD governance pipelines. Future work will focus on three directions: extending the ingestion layer to support additional AI coding assistants, developing a confidence calibration framework for high-assurance regulated environments, and exploring federated checkpoint architectures for organizations operating across multiple repository hosts.

REFERENCES

[1] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Measuring GitHub Copilot's Impact on Productivity," Communications of the ACM, vol. 67, no. 3, pp. 54–63, 2024.

[2] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in Proc. IEEE S&P, 2022, pp. 754–768.

[3] S. Kim and E. J. Whitehead, "Classifying Software Changes: Clean or Buggy?," IEEE Transactions on Software Engineering, vol. 34, no. 2, pp. 181–196, 2008.

[4] W. Enck and L. Williams, "Top Five Challenges and Associated Research Directions for Software Supply Chain Security," IEEE Security & Privacy, vol. 20, no. 2, pp. 96–100, 2022.

[5] Piyush Gaikwad.,"Health prediction and medical diagnosis using Naive Bayes," in International Journal of Advanced Research in Computer and Communication Engineering, pp. 31-35

[6] M. Brundage et al., "Toward Trustworthy AI Development: Mechanisms for Supporting Verifiable Claims," arXiv preprint arXiv:2004.07213, 2020.

[7] J. Kirchenbauer, J. Geiping, Y. Wen, J. Katz, I. Miers, and T. Goldstein, "A Watermark for Large Language Models," in Proc. ICML, 2023, pp. 17061–17084.

[8] M. W. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study," in Proc. IEEE ICSM, 2000, pp. 131–142.

[9] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository," in Proc. ACM CSCW, 2012, pp. 1277–1286.

[10] D. Yuan et al., "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems," in Proc. USENIX OSDI, 2014, pp. 249–265.

[11] M. Ballard et al., "SLSA: Levels for Software Artifact Integrity," Google Security Blog and Open Source Security Foundation, 2021. [Online].