Semantic Provenance and Policy-Aware Explanations for Agentic Coding Systems

Peb Ruswono Aryan[1], Fajar J. Ekaputra[2]

[1]TU Wien, Vienna, Austria
[2]Vienna University of Economics and Business, Vienna, Austria

Abstract

Agentic coding systems leave execution traces, yet developers still struggle to ask why a run failed, why an action was refused, or which artifacts were affected. This paper presents a Business Process Model and Notation (BPMN)-grounded semantic provenance approach that records runs as PROV-aligned RDF. A SPEAR (Semantic Process Engine for Agents over RDF) minimal coding agent emits action, artifact, policy, and governance events in one graph; SPARQL templates answer *why*, *why-not*, *impact*, and preliminary *what-if* questions. We evaluate deterministic reruns of the prototype and report solve-loop runtime and explanation-query latency. The measurements establish a baseline for later user, scalability, and ablation studies.

1. Introduction

Agentic coding systems are LLM-powered systems that execute multi-step software development workflows under policy constraints, including planning, file edits, tool calls, and validation. We use the term *coding agent* to refer to an LLM-powered system that autonomously executes multi-step software development workflows—editing files, running tests, invoking tools, and validating results—under policy constraints. When a run fails, is blocked, or changes artifacts unexpectedly, developers need trace evidence that connects actions to tools, files, and policy decisions.

Common logging approaches use text logs or JSON traces. They record events, but they rarely encode stable links for causal reconstruction, policy explanation, or impact inspection across runs. In practice, developers inspect these traces with grep, regular expressions, or tool-specific APIs. Cross-version tool changes can break implicit field links. Guardrail systems can block unsafe actions, but many expose only an accept/reject result rather than refusal evidence.

We address this gap with *semantic provenance for agentic coding*. Executions follow executable BPMN control flow and are logged as PROV-aligned RDF linked to artifacts and policy outcomes. The resulting graph carries the evidence needed for action chronology (*why*), refusal diagnosis (*why-not*), artifact-level downstream analysis (*impact*), and replay target selection (*what-if scaffold*). PROV-O supplies the shared vocabulary needed to compare traces across agent tools.

Research questions. We organize the evaluation around four questions: RQ1 (fidelity): how accurately do provenance queries reconstruct executed decision paths and policy refusals; RQ2 (runtime): what solve-loop runtime profile and SPARQL explanation-query latency are observed in the current implementation; RQ3 (developer utility): do explanations improve debugging efficiency, correctness, and trust versus text logs; RQ4 (design contribution): which components (policy context, BPMN alignment, logging granularity) contribute most to explanation quality.

Closest baseline distinction. Baselines are PROV-AGENT, AuditableLLM, and ToolGate [1, 2, 3]. They focus on unified provenance, audit integrity, and pre-execution enforcement, respectively. Our work targets post-hoc causal and policy explanation over executable process traces.

ESWC'26: Trust, Autonomy and Accountability in PKG-Based Agentic AI (TAAPAAI) Workshop on May 10, 2026 in Dubrovnik, Croatia

Contributions. The paper makes five contributions: a BPMN-based execution model for coding agents on the SPEAR RDF-native engine; a PROV-based trace model with action, artifact, and policy links; SPARQL templates for why/why-not/impact questions and a what-if scaffold; a reproducible SPEAR minimal coding agent[1] with policy controls and template knowledge; and measurements of runtime behavior and explanation-query latency, with a follow-up evaluation plan for fidelity, user utility, and component ablations.

2. Related Work

Provenance—the documentation of how data and processes evolve over time—supports reproducibility, debugging, and accountability in computational systems [4]. The W3C PROV standard provides a generic model for entities, activities, and agents, but applying it to agentic systems requires extensions for domain-specific concepts like policy compliance and action outcomes. Recent work on unified agent provenance, such as PROV-AGENT [1], captures interaction traces across tool calls and LLM reasoning steps. However, PROV-AGENT does not ground provenance in executable process models, which limits its support for causal reconstruction and policy-level explanation over agent runs. Agentic coding systems add further requirements: structured refusal evidence, artifact-level impact queries, and cross-run comparability [5, 6].

Personal and organizational knowledge graphs (PKGs) are used to ground agent behavior in graph-based memory [7, 8]. In the works considered here, the graph mainly represents agent knowledge or planning context, while the execution trace itself is not modeled as an executable process with policy-linked provenance. This distinction motivates our use of process models as the backbone for provenance capture.

Business Process Model and Notation (BPMN) [9] provides a visual language for specifying executable workflows, traditionally used in enterprise process automation. Recent work has explored executing BPMN models over RDF graphs, enabling processes to operate directly on semantic data [10, 11]. The SPEAR engine[2] (Semantic Process Engine for Agents over RDF) is one such implementation: it executes BPMN definitions as RDF-native processes and records each step as graph data. By modeling agent loops as BPMN processes, we use one representation for control flow and provenance. BPMN-based orchestration has also been applied to automated testing workflows in manufacturing contexts [12], showing a related use of process-grounded execution for multi-step technical tasks.

Explainable AI for tool-augmented LLMs has primarily focused on attribution, identifying which retrieved documents or tool calls influenced a generation, or on lightweight JSON/JSONL logging of execution traces [13, 14, 15]. Provenance-based approaches for LLM outputs have focused on fact-checking and attribution [5, 6]. These approaches improve visibility, but they do not directly answer queries such as *why-not* (why an action was blocked) or *impact analysis* (what changed after an action). Semantic graph representations can answer these questions when they encode links between actions, artifacts, and policies. Guardrail systems such as AuditableLLM [2] prioritize tamper resistance and compliance evidence, which is complementary to our explanation-query objective. Toolgate approaches [3] emphasize pre-execution tool-contract checks, while our contribution focuses on post-hoc causal and policy explanation over run traces. For our target setting, the unresolved step is to convert raw coding-agent run data into semantics that preserve causal, artifact, and policy links for post-hoc explanation.

Table 1 compares our work with the key related work selected as evaluation baselines. In this table, "Partial" denotes support that is present but not integrated as executable BPMN-grounded, query-template-oriented explanation.

[1] https://github.com/pebaryan/spear
[2] https://github.com/pebaryan/spear

Table 1
Comparator matrix for the closest baselines.

Approach	Exec. process grounding	Policy refusal evidence	Causal templates	Artifact impact query
PROV-AGENT	Partial	Partial	Partial	Partial
AuditableLLM	No	Yes	No	No
ToolGate	No	Yes (pre)	No	No
This work	Yes (BPMN)	Yes	Yes	Yes

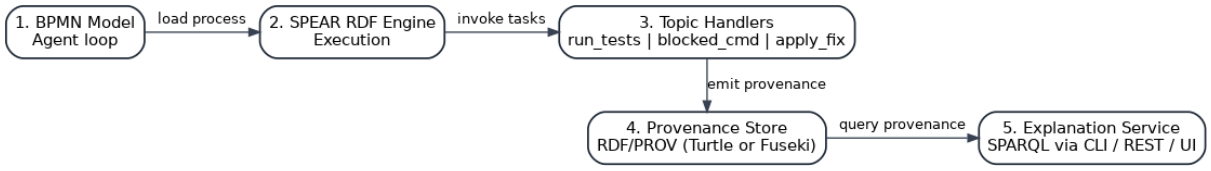

Figure 1: Five-component architecture: BPMN process model, SPEAR execution engine, topic handlers, provenance store, and SPARQL-based explanation service.

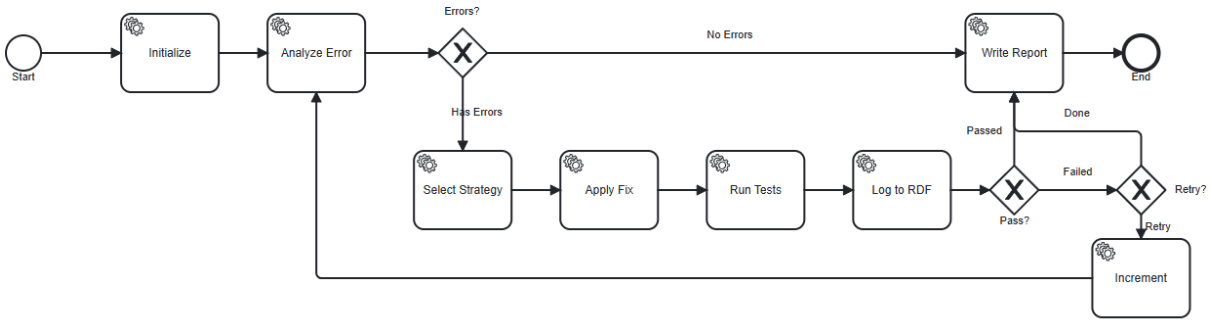

Figure 2: BPMN coding-agent loop used in the minimal coding agent implementation.

3. System Overview

Figure 1 shows the five-component architecture used in our system: (1) Business Process Model and Notation (BPMN) process model, (2) SPEAR (Semantic Process Engine for Agents over RDF) execution engine, (3) topic handlers, (4) provenance store, and (5) explanation service. The design separates process control, tool execution, logging, and query-time explanation, while keeping all components connected through RDF data. We will explain each of these components next.

BPMN Process Model. The workflow is modeled in BPMN 2.0 as an executable coding-agent loop with explicit error-analysis, fix-strategy, retry, and verification stages. Service tasks map to runtime handler topics (e.g., run tests, analyze errors, apply fix, log run report), and gateway decisions represent control-flow branches such as retry versus finish.

Compared with implicit loops in plain scripts, BPMN provides an explicit control-flow model that is auditable and easier to inspect. It also gives stable task identifiers that can be reused in provenance logs and later explanations. Figure 2 illustrates an example of the BPMN process model for coding-agent loop described in this paper.

SPEAR Execution Engine. SPEAR (Semantic Process Engine for Agents over RDF) executes BPMN directly over RDF graphs. Process state transitions and task events are represented as triples, so the same representation is used for both execution and analysis. This avoids a separate ETL step from runtime logs into an explanation store.

When SPEAR reaches a service task, it delegates work to the registered topic handler and continues

with returned results. Each task activation and completion is logged with BPMN node identifiers, which supports direct alignment between process diagrams and trace data.

Topic Handlers and Tool Integration. Topic handlers connect process tasks to concrete operations. In our implementation they cover baseline tests, blocked commands, local fixes, rollback, and post-fix verification. They execute tools (e.g., `pytest` or file edits), return structured results, and emit status data used by provenance and explanation queries.

This separation keeps tools independent from BPMN internals and makes extension straightforward: new capabilities are added as handlers without changing the execution core. Policies are defined as handler-level checks in the SPEAR topic handlers. Each handler evaluates risk conditions—such as command type, file path patterns, or network access requirements—against a policy catalog and emits structured refusal evidence (`ag:status`, `ag:reason`, `ag:policyId`, `ag:policyDecision`) into the provenance graph at runtime, rather than blocking execution silently.

Provenance Store. Execution traces are persisted as RDF (Turtle) in case files and an engine graph. The same structure can be loaded into a triplestore (e.g., Fuseki) for multi-run analysis.

Each run stores process metadata (timestamps, instance identifiers), action events, and policy-relevant status fields. This provides both an audit trail and a queryable basis for explanation templates.

Explanation Service. The explanation service is implemented as parameterized SPARQL templates over the provenance graph. In the current implementation, templates are executed from CLI scripts; the same templates can be exposed through REST or IDE integrations.

Instead of raw tuples, outputs are formatted for developer-facing questions: why an edit happened, why a command was blocked, and what was affected by a change.

4. Provenance Model

Our model combines W3C PROV-O with a lightweight agent namespace (`ag:`) and policy-related fields. The goal is to capture temporal order, action-artifact links, policy outcomes, and explicit causal ordering in one queryable knowledge graph.

4.1. Core Vocabulary

The core classes of our ontologies are:

- **`ag:Run`**: one end-to-end agent execution.
- **`ag:Action`**: one concrete operation in a run (test call, blocked command, file edit).
- **PROV links**: `prov:wasAssociatedWith`, `prov:used`, `prov:startedAtTime`, and `prov:wasInformedBy` for explicit action ordering.

While timestamps (`prov:startedAtTime`) provide wall-clock ordering, they are insufficient for distinguishing concurrent from sequential actions. We therefore record `prov:wasInformedBy` links from each action to its immediate predecessor, providing an explicit causal chain independent of timestamp resolution. For sequential actions, both timestamps and `wasInformedBy` agree on ordering; for concurrent actions in parallel BPMN gateways, multiple actions may share a predecessor, and the absence of a `wasInformedBy` link between two actions sharing the same timestamp signals concurrent execution.

4.2. Action and Artifact Links

Actions are linked to artifacts (for example, edited files) through `prov:used` and to their predecessors through `prov:wasInformedBy`. We represent target files with file URIs and keep action status in `ag:status` (e.g., completed, failed, blocked). Optional reasons (e.g., policy refusal messages) are stored in `ag:reason`.

This structure supports explanation traces such as run → action → artifact and allows filtering by status for why-not queries. The `prov:wasInformedBy` links additionally support causal chain reconstruction independent of timestamp ordering.

4.3. BPMN Alignment and Policy Context

In the SPEAR-based run, each event also carries BPMN node identifiers. This makes it possible to trace results back to concrete process tasks in each run.

Policy context is represented by action status/reason plus lightweight policy fields (`ag:policyId`, `ag:policyDecision`, `ag:policyClause`) in the current system. This keeps the model compact while supporting structured blocked-action explanations.

4.4. Governance and Safety Event Modeling

Beyond blocked-action reasons, the minimal coding agent records governance events—provenance-captured records of policy-relevant system decisions—as first-class provenance context. Governance events include approval-gated risky tool actions, optional external authorization checks, redaction-profile selection for sensitive outputs, retry-policy profile selection, and calibration updates for deterministic template knowledge.

This extension is still lightweight and compatible with the core run/action pattern: governance events are attached to run reports and action nodes through identifiers and timestamps rather than a heavy policy ontology. The benefit is practical queryability for compliance and debugging questions such as: which risky actions required approval, which actor approved or denied, which authorization decision was returned, and which redaction/retry profile was active for a run. For redaction, only profile identifiers and field-level redaction markers are stored, not redacted payload values.

The same pattern also supports post-hoc quality analysis: template calibration metadata links evaluation outcomes to subsequent template-weight updates, enabling traceable explanations of why deterministic repair preference changed across iterations.

4.5. Trace Excerpt

Listing 4.5 shows a shortened trace pattern used in our implementation, including explicit causal ordering via `prov:wasInformedBy`.

Minimal provenance trace pattern with causal ordering

```
@prefix ag: <http://vibe.graph/agent#> .
@prefix prov: <http://www.w3.org/ns/prov#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://vibe.graph/run/1> a ag:Run ;
rdfs:label "Demo agent run" .

<http://vibe.graph/run/1/action/1> a ag:Action ;
rdfs:label "run baseline tests" ;
ag:tool "pytest" ;
ag:status "completed" ;
prov:wasAssociatedWith <http://vibe.graph/run/1> ;
prov:startedAtTime "2026-02-11T10:01:00Z"^^xsd:dateTime .

<http://vibe.graph/run/1/action/2> a ag:Action ;
rdfs:label "blocked shell command" ;
ag:tool "shell" ;
ag:status "blocked" ;
ag:reason "policy: no network commands" ;
ag:policyId "POLICY-NETWORK-001" ;
ag:policyDecision "deny" ;
prov:wasAssociatedWith <http://vibe.graph/run/1> ;
prov:wasInformedBy <http://vibe.graph/run/1/action/1> ;
prov:startedAtTime "2026-02-11T10:02:15Z"^^xsd:dateTime .
```

5. Explanation Templates

We implement three SPARQL template families (why/why-not/impact) and one what-if scaffold. Templates are parameterized by run URI or action URI and executed over the same RDF graph used for provenance capture.

5.1. Why: Causal Chain Explanation

The **Why** template reconstructs action chronology for a run and links actions to tools, status, timestamps, and optional target artifacts. It traverses `prov:wasInformedBy` chains to ensure causal ordering independent of timestamp resolution. It supports explanations such as baseline test failure, policy-blocked command, local fix, and post-fix pass.

5.2. Why-Not: Blocked Action Explanation

The **Why-not** template extracts blocked actions and their recorded refusal reasons. Templates are parameterized by run/action identifiers at query time.

Why-not template: blocked actions with reasons

```
PREFIX prov: <http://www.w3.org/ns/prov#>
PREFIX ag: <http://vibe.graph/agent#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?action ?label ?tool ?blockedAt ?reason ?policyDecision ?policyClause
WHERE {
    BIND(IRI($runUri) AS ?run)
    ?action a ag:Action ;
    prov:wasAssociatedWith ?run ;
    ag:status "blocked" ;
    prov:startedAtTime ?blockedAt .
    OPTIONAL { ?action rdfs:label ?label }
    OPTIONAL { ?action ag:tool ?tool }
    OPTIONAL { ?action ag:reason ?reason }
    OPTIONAL { ?action ag:policyDecision ?policyDecision }
    OPTIONAL { ?action ag:policyClause ?policyClause }
  }
ORDER BY ?blockedAt
```

In our implementation, this query returns blocked or denied actions when policy controls are triggered and returns an empty result when no blocked action is present in the selected run.

5.3. Impact: Downstream Effect Analysis

The **Impact** template estimates downstream effects by finding later actions that use the same artifact as a seed action. This answers questions such as "what happened after this file was edited?" and "which tests were affected by this artifact?"

Impact pattern: downstream actions sharing an artifact

```
?action1 prov:used ?artifact .
?action2 prov:used ?artifact .
FILTER (?action2 != ?action1)
```

5.4. What-If: Counterfactual Scaffolding

The **What-if** component is currently a scaffold: it identifies actions coupled to a selected action via shared artifacts and marks replay candidates for counterfactual analysis. Full counterfactual evaluation requires controlled replay semantics and is outside the current implementation.

6. Evaluation

In this section, we report on our evaluation setup (cf. Section 6.1), initial results (cf. Section 6.2), our plan for further evaluation (cf. Section 6.3, and threats to validity (cf. Section 6.4).

6.1. Evaluation Setup

We implement the approach in a SPEAR minimal coding agent that combines BPMN-grounded execution with provenance-first explanation support. The implementation integrates policy controls, governance logging, and query templates in one RDF-native stack.

Table 2
Implementation mapping of required capabilities.

Capability	Runtime mechanism	Provenance/evaluation artifact
Policy-constrained tool use	Risk-classified tool calls with approval gates	Approval-event records with decision and rationale
Authorization compliance	External policy-decision check before risky execution	Authorization outcomes linked to run records
Sensitive-data protection	Configurable redaction profiles for reports and logs	Redaction profile plus redacted-field evidence
Adaptive repair behavior	Retry-policy profiles (standard/aggressive/auto in current reruns)	Run reports with selected profile and final outcome
Repair knowledge evolution	Template calibration from evaluation history	Template-knowledge updates with calibration metadata
Run-level explainability	Stable run identifiers with action timestamps	Correlated why/why-not/impact query outputs

Prototype Development of the Minimal Coding Agent. Table 2 maps required capabilities to concrete implementation mechanisms. Explanation support is implemented through runtime controls that emit structured provenance fields, not only through query templates.

Experimental Scenarios. The current deterministic evaluation uses the `solve_baseline` scenario—a reproducible coding task in which the agent must diagnose and fix failing unit tests in a small Python project—across retry-policy profiles (standard, aggressive, auto). This is not an established benchmark; it is a reproducible task designed for controlled evaluation. Each run executes the same core workflow: baseline test state, fix-loop execution, and post-fix verification with provenance emission. This setup supports controlled runtime and SPARQL-latency measurement while keeping run conditions comparable across profiles.

Generated Outputs and Reproducibility. Each run produces provenance trace files, engine graphs, and query outputs. The system also emits approval logs, template-KG snapshots, run-level reporting, and evaluation summaries. Repository READMEs provide step-by-step commands for reproducing the scenarios and measurements.

Evaluation Questions and Metrics. We evaluate along the four research questions introduced in Section 1. RQ1 (fidelity) concerns the precision and recall of provenance queries in reconstructing executed decision paths and policy refusals. RQ2 (runtime) measures the solve-loop duration profile and SPARQL explanation-query latency. RQ3 (developer utility) assesses whether semantic explanations improve debugging efficiency, correctness, and trust compared to text logs. RQ4 (design contribution) quantifies the ablation impact of individual architectural components—policy context, BPMN alignment, and logging granularity—on explanation quality.

6.2. Evaluation Results

We re-ran the measurements on the SPEAR minimal coding agent using deterministic evaluation mode for `solve_baseline` under three retry-policy profiles (standard, aggressive, auto), each with five repetitions. The runs were executed on a local workstation (Linux 6.6, AMD Ryzen 7 PRO 4750U, 16 vCPUs) using a Python 3.11 runtime for compatibility with the codebase. SPARQL explanation latency was measured over 50 repetitions for three query templates (`why`, `why-not`, `impact`) on the generated provenance graphs. This section reports feasibility-oriented baseline evidence rather than a full comparative study.

Table 3
Empirical results from re-runs on the minimal coding agent.

Profile	Success rate	Solve p50 (s)	Solve p95 (s)
Standard	100.0%	4.153	4.167
Aggressive	100.0%	5.186	6.476
Auto	100.0%	5.457	5.742
SPARQL explanation latency (ms, p50/p95): *Why 12.69/14.04; Why-not 30.56/33.85; Impact 12.85/13.97*			

The resulting provenance artifacts remain compact enough for interactive inspection. In the current workspace snapshot, `session_history.ttl` has 300 triples (18.5 KB), `run_reports.ttl` is 51 KB, and `engine_graph.ttl` has 2058 triples (157.4 KB). These measurements establish the current RQ2 baseline. The remaining evaluation questions require labeled traces, participant tasks, and ablations, which are defined below rather than claimed as completed results.

Ablation Setup and Expected Diagnostic Value. To answer RQ4 in a falsifiable way, we define three ablations and their expected failure modes.

- **No-policy-context**: remove policy-specific fields and approval/authz events. Expected impact is lower why-not completeness and refusal-diagnosis precision.
- **No-BPMN-alignment**: keep action logs but remove stable process-node alignment. Expected impact is weaker causal reconstruction and reduced explanation consistency.
- **Coarse logging**: record only step-level outcomes without artifact links. Expected impact is lower impact-query usefulness and weaker downstream traceability.

These ablations are designed for diagnostic separation as well as aggregate metrics: removing one component should produce a specific explanation deficit. This setup supports a direct test of whether each architectural component adds value beyond generic logging.

6.3. Evaluation Roadmap

The next evaluation stage covers three extensions beyond the deterministic harness. For explanation fidelity (RQ1), we will compare template outputs with labeled ground truth and report precision/recall and inter-annotator agreement. For developer utility (RQ3), we define a cross-over design with two conditions: (A) text logs only and (B) semantic explanations over the same underlying executions. Participants perform fixed debugging tasks on policy- and repair-relevant agent runs; the primary measures are time-to-diagnosis, final diagnosis correctness, confidence calibration, and NASA-TLX workload. For scalability, we will sweep run length and graph size, then add medium-size open-source repositories to characterize runtime, latency, and storage growth.

To reduce learning effects in the user study, task order is randomized and condition order is counterbalanced. We collect structured post-task judgments about explanation usefulness, perceived auditability, and trust boundaries (for example, whether participants can distinguish policy refusals from functional failures). For methodology transparency, we plan blocked randomization by prior experience, pre-registered primary endpoints (diagnosis time and correctness), minimum sample-size targets from pilot variance, and effect-size reporting with confidence intervals. The ablations defined above instantiate RQ4 by testing whether policy context, BPMN alignment, and logging granularity each add value beyond generic logging.

6.4. Threats to Validity

- **Construct validity.** Because measurements currently focus on deterministic `solve_baseline` runs, explanation quality may be overstated; we address this with adversarial run variants, multi-granularity labels, and per-template error breakdowns.
- **Internal validity.** Runtime and latency can be biased by workstation load, warm caches, and startup effects; we mitigate this with interleaved run order, explicit warm-up separation, environment metadata, and p50/p95 reporting.
- **External validity.** Results on a small Python target may not transfer to larger, mixed-language, and non-deterministic repositories; we therefore plan staged transfer experiments with scaling curves for latency, trace size, and refusal diagnosis.
- **Conclusion validity.** The planned cross-over user study can be underpowered and affected by learning effects; mitigation includes power-aware recruitment, counterbalancing, blocked randomization by experience, and effect sizes with confidence intervals.
- **Operational validity.** Inconsistent population of approval/authz/redaction/retry fields can distort queries; we enforce schema checks, integrity queries, and explicit "unknown" states in CI.

7. Discussion and Limitations

The current system prioritizes transparent end-to-end behavior over feature completeness. Three limitations are relevant.

First, policy context is represented as compact fields and event metadata (status/reason, policy IDs, approval/authz events, redaction profile, retry profile); it is not yet a first-class policy ontology. Second, current impact queries rely on artifact-link structures available in the traces and do not model full semantic program dependence. Third, what-if remains a replay-oriented design for state restoration and not a complete counterfactual engine. Fourth, the current BPMN model uses sequential service tasks, so all actions in the evaluated scenario are strictly ordered. For concurrent actions in more complex BPMN models (e.g., parallel gateways), `prov:wasInformedBy` provides explicit ordering independent of timestamps, and the absence of such links between actions sharing a timestamp signals concurrent execution. Full support for parallel gateway execution is planned future work.

Provenance as a governance tool. The same trace structure also supports governance diagnostics. For example, when process intent and policy controls diverge (e.g., a BPMN service task attempts an operation that policy blocks), the run records both the attempted action and the denial context (`ag:status`, `ag:reason`, policy decision fields, and linked approval/authz events). This representation preserves the conflict as queryable evidence instead of collapsing it into a generic failure log, including structured evidence about actions that were intentionally blocked.

Within this scope, the evaluated runs produced executable provenance and low-latency query responses.

8. Conclusion

This paper presented a BPMN-grounded semantic provenance approach for agentic coding systems. The prototype records execution and provenance in RDF, then uses SPARQL templates to explain why actions occurred, why actions were blocked, and what changed as a consequence. It also includes a what-if scaffold for replay-oriented analysis.

The evaluated runs combine provenance capture, policy-linked refusal logging, and explanation queries over generated traces. Current measurements give an initial runtime and latency baseline for follow-up scaling and user studies.

Future work. Planned extensions include broader user studies, IDE-oriented explanation views, ontology-level policy modeling, and multi-agent replay-based counterfactual analysis.

Acknowledgments

We thank the TAAPAAI reviewers for their constructive feedback.

Declaration on Generative AI

During the preparation of this work, the author(s) used AI-assisted writing tools in order to: Grammar and spelling check, Paraphrase and reword. The author(s) also used an AI-assisted coding system to automate paper typesetting, template conversion, and LaTeX formatting. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] R. Souza, A. Gueroudji, S. DeWitt, D. Rosendo, T. Ghosal, R. Ross, P. Balaprakash, R. F. da Silva, Prov-agent: Unified provenance for tracking ai agent interactions in agentic workflows, in: Proc. Workshop on Agentic Workflows, 2025. URL: https://arxiv.org/abs/2508.02866.

[2] D. Li, G. Yu, X. Wang, B. Liang, Auditablellm: A hash-chain-backed, compliance-aware auditable framework for large language models, Electronics 15 (2026) 56. doi:10.3390/electronics15010056, published online 23 December 2025.

[3] Y. Liu, X. Peng, J. Cao, X. Wang, S. Deng, J. Chen, J. Yin, X. Zhang, Toolgate: Contract-grounded and verified tool execution for llms, arXiv preprint (2026). URL: https://arxiv.org/abs/2601.04688.

[4] T. Lebo, S. Sahoo, D. McGuinness, et al., Prov-o: The prov ontology, in: W3C Recommendation, 2013.

[5] H. Sankararaman, M. N. Yasin, T. Sorensen, A. D. Bari, A. Stolcke, Provenance: A light-weight fact-checker for retrieval augmented llm generation output, in: Proc. EMNLP 2024 (Industry Track), 2024, pp. 1305–1313. doi:10.18653/v1/2024.emnlp-industry.97.

[6] B. D. Earp, H. Yuan, J. Koplin, S. Porsdam Mann, Llm use in scholarly writing poses a provenance problem, Nature Machine Intelligence 7 (2025) 1889–1890. doi:10.1038/s42256-025-01159-8.

[7] N. Cao, et al., Personal knowledge graphs and personal agents: A cross-fertilization opportunity, in: Proc. Knowledge Graph Conference, 2024.

[8] J. Wu, Q. Zhao, Z. Chen, K. Qin, Y. Zhao, X. Wang, Y. Yao, GAP: Graph-based agent planning with parallel tool use and reinforcement learning, 2025. URL: https://arxiv.org/abs/2510.25320. doi:10.48550/arXiv.2510.25320. arXiv:2510.25320.

[9] Object Management Group, Business Process Model and Notation (BPMN), Version 2.0.2, OMG Specification formal/2014-02-01, 2014. URL: https://www.omg.org/spec/BPMN/2.0.2/.

[10] D. N. Dolha, R. A. Buchmann, Experiments with natural language queries on RDF vs. XML-serialized BPMN diagrams, Procedia Computer Science 246 (2024) 3246–3255. doi:10.1016/j.procs.2024.09.315, 28th International Conference on Knowledge Based and Intelligent information and Engineering Systems (KES 2024).

[11] F. Hahn, S. Todorovikj, Ontology driven transformation of camunda bpmn instances into rdf knowledge graphs, Poster, AIKD-SD 2025, 2025. URL: https://doi.org/10.5281/zenodo.17702330. doi:10.5281/zenodo.17702330.

[12] S. König, B. Vogel-Heuser, A. Hradecky, et al., Executing automotive test workflows with bpmn and web service orchestration in software-defined car manufacturing, Production Engineering 19 (2025) 195–209. doi:10.1007/s11740-024-01302-1.

[13] M. Horovicz, AgentSHAP: Interpreting LLM agent tool importance with monte carlo shapley value estimation, 2025. URL: https://arxiv.org/abs/2512.12597. doi:10.48550/arXiv.2512.12597. arXiv:2512.12597.

[14] M. Lyu, S. Wang, R. Zhang, et al., Tart: Tool-augmented reasoning for tabular tasks with explanations, in: Findings of NAACL 2025, 2025, pp. 3450–3464. URL: https://aclanthology.org/2025.findings-naacl.244.

[15] X. Wang, M. Yin, E. Koh, M. D. Dogan, XAgen: An explainability tool for identifying and correcting failures in multi-agent workflows, 2025. URL: https://arxiv.org/abs/2512.17896. doi:10.48550/arXiv.2512.17896. arXiv:2512.17896.