Semantic Provenance and Policy-Aware Explanations for Agentic Coding Systems

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

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

TAAPAAI Workshop @ ESWC 2026
May 10, 2026, Dubrovnik, Croatia

Outline

Context and Background

- **Agentic coding** is an increasingly popular software development approach where autonomous AI agents plan, write, test, and modify code with minimal human intervention.
- **A coding agent** (in this context) is a software powered by a large language model (LLM) that performs software development tasks autonomously.
- These agents understand the goal, break it down into steps, and execute the necessary actions to complete the work.

Problem Definition

- **Agentic coding systems** execute multi-step software development workflows (planning, editing files, running tests, invoking tools) under policy constraints
- When a run fails or is blocked, developers need trace evidence connecting actions to tools, files, and policy decisions
- **Current logging approaches:**
  - Text logs or JSON traces record events
  - Rarely encode stable links for causal reconstruction
  - No policy explanation or impact inspection across runs
  - Guardrails expose only accept/reject, not refusal evidence

Research Questions

RQ1 – Fidelity How accurately do provenance queries reconstruct executed decision paths and policy refusals?

RQ2 – Runtime What solve-loop runtime and SPARQL explanation-query latency are observed? **(Main Focus)**

RQ3 – Developer Utility Do explanations improve debugging efficiency, correctness, and trust vs text logs?

RQ4 – Design Contribution Which components (policy context, BPMN alignment, logging granularity) contribute most to explanation quality?

An Example Scenario: "solve_baseline"

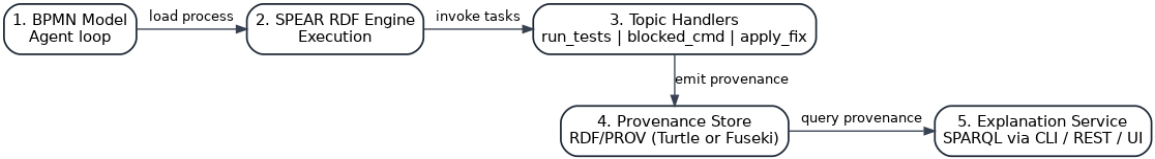
What is `solve_baseline`?

- A deterministic coding task: diagnose and fix failing unit tests in a small Python project
- **Not** an established benchmark – designed for controlled evaluation
- Each run executes the same core workflow:
  1. Run baseline tests (fail expected)
  2. Analyze error output
  3. Apply local fix
  4. Post-fix verification (pass expected)
- Three retry-policy profiles: standard, aggressive, auto (5 repetitions each)

Our Approach: Semantic Provenance

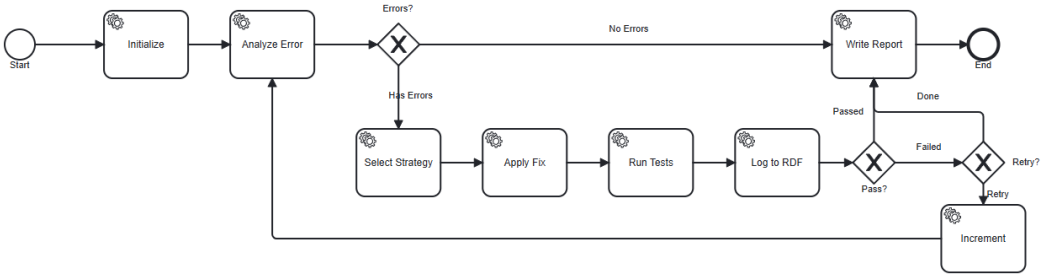
- Integrating **semantic provenance** to agentic coding systems
- Executions follow executable **BPMN** control flow
- Logged as **PROV-aligned RDF** linked to artifacts and policy outcomes
- Resulting graph supports:
  - **Why** – action chronology
  - **Why-not** – refusal diagnosis
  - **Impact** – artifact-level downstream analysis
  - **What-if** – replay target selection (scaffold)
- **PROV-O** supplies shared vocabulary for cross-tool trace comparison

Five-Component Architecture



- **BPMN Process Model** – executable coding-agent loop
- **SPEAR Engine** (*ongoing work*) – RDF-native BPMN execution
- **Topic Handlers** – tool integration + policy checks
- **Provenance Store** – RDF traces
- **Explanation Service** – SPARQL templates

BPMN Coding-Agent Loop: "solve_baseline" Scenario



- Explicit error-analysis, fix-strategy, retry, and verification stages
- Stable task identifiers for provenance alignment
- Gateway decisions for retry vs finish control flow

Core Vocabulary

- `ag:Run` – one end-to-end agent execution
- `ag:Action` – one concrete operation (test, blocked command, file edit)
- **PROV links**:
  - `prov:wasAssociatedWith` – action to run
  - `prov:used` – action to artifact
  - `prov:startedAtTime` – wall-clock ordering
  - `prov:wasInformedBy` – **explicit causal chain** independent of timestamps

Key Insight

`prov:wasInformedBy` distinguishes sequential from concurrent actions, independent of timestamp resolution.

Trace Excerpt: Action Structure

Action 1 – baseline test execution

```
@prefix ag: <http://vibe.graph/agent#> .
@prefix prov: <http://www.w3.org/ns/prov#> .

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

Trace Excerpt: Policy Refusal

Action 2 – blocked command with causal link

```
<.../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 <.../run/1> ;
  prov:wasInformedBy <.../action/1> ;
  prov:startedAtTime "2026-02-11T10:02:15Z" .
```

- `prov:wasInformedBy` links to predecessor action
- Policy fields (`policyId`, `policyDecision`) explain refusal

Governance Events

Beyond blocked-action reasons, the agent shall capture complex **governance events** as first-class provenance, e.g.,

- Approval-gated risky tool actions
- External authorization check outcomes
- Redaction-profile selection for sensitive outputs
- Retry-policy profile selection
- Calibration updates for deterministic template knowledge

(In this paper) Lightweight pattern: events attached via identifiers and timestamps, not a heavy policy ontology.

SPARQL Template Families Overview for Event Explanation

Template	Question Answered
Why	What actions happened in this run, and in what order?
Why-not	Why was a specific action blocked or denied?
Impact	What other actions were affected by this artifact?
What-if	Which actions should be replayed if we change one step?

All templates are parameterized by run URI or action URI and executed over the same provenance graph.

Why Template: Causal Chain

Question: *What happened in run `run/1`, and in what order?*

Query

```
SELECT ?label ?tool ?status ?startedAt
WHERE {
  ?action a ag:Action ;
    prov:wasAssociatedWith <.../run/1> ;
    ag:status ?status ;
    prov:startedAtTime ?startedAt .
  ?action rdfs:label ?label ;
          ag:tool ?tool .
} ORDER BY ?startedAt
```

Expected output:

Label	Tool	Status	Time
run baseline tests	pytest	completed	10:01:00
blocked shell cmd	shell	blocked	10:02:15
apply local fix	sed	completed	10:03:30

Why-Not Template: Query

Question: *Why was action `action/2` blocked in run `run/1`?*

Query

```
SELECT ?label ?tool ?reason
       ?policyId ?policyDecision
WHERE {
  ?action a ag:Action ;
    prov:wasAssociatedWith <.../run/1> ;
    ag:status "blocked" ;
    ag:reason ?reason ;
    ag:policyId ?policyId ;
    ag:policyDecision ?policyDecision .
  ?action rdfs:label ?label ;
          ag:tool ?tool .
}
```

- Filters actions with `ag:status "blocked"`
- Returns structured refusal evidence: reason, policy ID, decision

Why-Not Template: Expected Output

Query result for run `run/1`:

Label	Tool	Reason	Policy ID	Decision
blocked shell cmd	shell	policy: no network commands	POLICY-NETWORK-001	deny

- Returns blocked/denied actions with full refusal context
- Empty result when no policy violation occurred
- Enables developers to understand *why* an action was refused, not just that it was blocked

Impact Template: Downstream Effects

Question: *What other actions used the same file as action `action/3`?*

Query

```
SELECT ?action2 ?label2 ?time2
WHERE {
  <.../action/3> prov:used ?file .
  ?action2 prov:used ?file ;
    prov:startedAtTime ?time2 .
  FILTER (?action2 != <.../action/3>)
  ?action2 rdfs:label ?label2 .
} ORDER BY ?time2
```

Expected output:

Action	Label	Time
action/4	post-fix verify	10:04:00
action/5	run regression	10:05:10

What-If Template: Counterfactual Scaffold

Question: *If we replay from action `action/3`, which actions should be re-run?*

Query

```
SELECT ?replay ?label ?status
WHERE {
  <.../action/3> prov:used ?file .
  ?replay prov:used ?file ;
    ag:status ?status .
  FILTER (?replay != <.../action/3>)
  ?replay rdfs:label ?label .
}
```

Expected output:

Action	Label	Status
action/4	post-fix verify	completed
action/5	run regression	completed

Evaluation Setup

- **Scenario:** `solve_baseline` – deterministic coding task (diagnose and fix failing unit tests)
- **Retry profiles:** standard, aggressive, auto (5 repetitions each)
- **SPARQL latency:** 50 repetitions for why/why-not/impact templates
- **Hardware:** Linux 6.6, AMD Ryzen 7 PRO 4750U, 16 vCPUs, Python 3.11

Results

Profile	Success	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

Evaluation Roadmap

- **RQ1 (Fidelity)** – compare template outputs with labeled ground truth; report precision/recall
- **RQ2 (Runtime)** – (Done)
- **RQ3 (Developer Utility)** – cross-over study:
  - Condition A: text logs only
  - Condition B: semantic explanations
  - Measures: time-to-diagnosis, correctness, NASA-TLX workload
- **Scalability** – sweep run length and graph size; add medium-size open-source repos

Evaluation Roadmap - Ablation Setup (RQ4)

Three ablations with expected failure modes:

No-policy-context Remove policy fields and approval/authz events → lower why-not completeness

No-BPMN-alignment Remove stable process-node alignment → weaker causal reconstruction

Coarse logging Record only step-level outcomes → lower impact-query usefulness

Each ablation produces a specific explanation deficit, testing whether each component adds value beyond generic logging.

Discussion and Limitations

- Policy context is compact fields, not yet a first-class policy ontology
- Impact queries rely on artifact-link structures, not full semantic program dependence
- What-if is a replay-oriented scaffold, not a complete counterfactual engine
- Current BPMN model uses sequential service tasks; parallel gateway support planned

Provenance as Governance Tool

When process intent and policy controls diverge, the run records both the attempted action and the denial context as queryable evidence.

Conclusion

- BPMN-grounded semantic provenance for agentic coding systems
- Records execution and provenance in RDF
- SPARQL templates explain why, why-not, and impact
- What-if scaffold for replay-oriented analysis
- Initial runtime and latency baseline established

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

Thank You!

https://github.com/pebaryan/spear

Code: examples/minimal_coding_agent

Appendix: Input – Failing Test Project

Target: small Python project with intentionally failing unit tests

Example failing test

```
def test_addition():
    result = add(2, 3)
    assert result == 6  # Bug: should be 5

def test_subtraction():
    result = subtract(10, 4)
    assert result == 5  # Bug: should be 6
```

- Agent must read test output, identify the bug, and apply the correct fix
- Deterministic: same errors, same expected fixes across all runs

Appendix: Execution Command

How to run the evaluation:

Run solve_baseline with standard profile

```
$ python agent_run.py \
    --scenario solve_baseline \
    --profile standard \
    --repetitions 5 \
    --deterministic
```

Outputs per run:

- `session_history.ttl` – provenance trace (RDF)
- `run_reports.ttl` – run-level summary
- `engine_graph.ttl` – full execution graph
- Query outputs (why/why-not/impact)

Appendix: Retry Policy Profiles

Three profiles control the agent's repair behavior:

Profile	Behavior
Standard	Conservative: applies one fix per iteration, runs full test suite after each change. Prioritizes correctness over speed.
Aggressive	Applies multiple fixes in a single iteration, skips intermediate test runs. Faster but may introduce regressions.
Auto	Dynamically selects strategy based on error complexity: simple errors use standard, complex/multi-failure errors use aggressive.

- All profiles emit the same provenance structure for cross-profile comparison
- Results: Standard fastest (p50: 4.15s), Aggressive (p50: 5.19s), Auto (p50: 5.46s)

Appendix: Evaluation Methodology per RQ

How each research question is evaluated:

RQ	Methodology	Status
RQ1 Fidelity	Compare template outputs against labeled ground truth. Measure precision/recall for action reconstruction and policy refusal detection.	Planned
RQ2 Runtime	Deterministic reruns of `solve_baseline` under 3 profiles (5 reps each). Measure solve-loop duration (p50/p95). SPARQL latency over 50 repetitions per template.	Done
RQ3 Developer Utility	Cross-over user study: (A) text logs vs (B) semantic explanations. Measures: time-to-diagnosis, diagnosis correctness, NASA-TLX workload.	Planned
RQ4 Design Contribution	Ablation study: remove one component at a time (policy context, BPMN alignment, logging granularity).	Planned

Appendix: RQ1 – Fidelity Evaluation Details

Goal: How accurately do provenance queries reconstruct executed decision paths?

- **Ground truth:** manually label action sequences, policy decisions, and artifact links from known runs
- **Metrics:**
  - Precision: fraction of query results that match ground truth
  - Recall: fraction of ground truth events recovered by queries
  - F1-score per template (why/why-not/impact)
- **Inter-annotator agreement:** Cohen's κ between two independent labelers
- **Error breakdown:** classify false positives/negatives by template and error type

Status: Planned

Baseline traces collected; labeling protocol defined. Full fidelity evaluation pending.

Appendix: RQ3 – Developer Utility Study Design

Goal: Do semantic explanations improve debugging over text logs?

- **Design:** within-subjects cross-over study
  - Condition A: text logs only
  - Condition B: semantic explanations (SPARQL query results)
- **Tasks:** fixed debugging scenarios on policy- and repair-relevant agent runs
- **Primary measures:**
  - Time-to-diagnosis (seconds)
  - Final diagnosis correctness (binary)
- **Secondary measures:** NASA-TLX workload, confidence calibration, trust ratings
- **Controls:** counterbalanced condition order, blocked randomization by experience

Status: Planned

Study design defined; participant recruitment and task materials pending.

Appendix: RQ4 – Ablation Study Design

Goal: Which components contribute most to explanation quality?

Ablation	Expected Failure Mode
No-policy-context	Remove policy fields and approval/authz events → lower why-not completeness and refusal-diagnosis precision
No-BPMN-alignment	Remove stable process-node alignment → weaker causal reconstruction and explanation consistency
Coarse logging	Record only step-level outcomes, no artifact links → lower impact-query usefulness

- Each ablation produces a *specific* explanation deficit, not just aggregate degradation
- Tests whether each component adds value beyond generic logging

Status: Planned

Ablation conditions defined; execution and measurement pending.

Appendix: Provenance Trace (1/2) – Action Chain

Trace excerpt from a `solve_baseline` run:

session_history.ttl – actions 1–2

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

<.../run/1> a ag:Run ;
  rdfs:label "solve_baseline run" .

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

Appendix: Provenance Trace (2/2) – Fix + Verification

Continued – actions 3–4:

session_history.ttl – actions 3–4

```
<.../run/1/action/3> a ag:Action ;
  rdfs:label "apply fix" ;
  ag:tool "file_edit" ;
  ag:targetFile <file://src/calculator.py> ;
  ag:status "completed" ;
  prov:wasAssociatedWith <.../run/1> ;
  prov:wasInformedBy <.../action/2> ;
  prov:startedAtTime "2026-02-11T10:02:00Z" .

<.../run/1/action/4> a ag:Action ;
  rdfs:label "post-fix verification" ;
  ag:tool "pytest" ;
  ag:status "completed" ;
  ag:exitCode 0 ;
```

Appendix: Trace Statistics

Artifact sizes from a typical run:

Artifact	Triples	Size
session_history.ttl	300	18.5 KB
run_reports.ttl	–	51 KB
engine_graph.ttl	2,058	157.4 KB

- Traces remain compact enough for interactive inspection
- All artifacts are queryable via SPARQL templates
- Repository READMEs provide step-by-step reproduction commands