SPINE: STRUCTURED POLICY REFINEMENT WITH INTENT-TO-EXECUTION LINEAGE FOR LLM SOFTWARE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

As Large Language Models (LLMs) expand agentic workflows across complex systems, execution logs capture runtime actions but fail to answer key operational questions: which stakeholder intent prompted an action, which intermediate decision derived it, and who was responsible. Consequently, operators must reconstruct attribution post hoc rather than retrieve it directly from the execution record. We present SPINE, a policy refinement framework that represents stakeholder intents as declarative policies and progressively refines them into definitive and imperative policies with explicit refinement relationships. The resulting policy hierarchy establishes an end-to-end intent-to-execution lineage: every policy node records its parent, definer, and enforcer, while an orchestrator stamps the active policy identifier onto execution records at write time. Traceability is thus reduced to a parent-pointer traversal bounded by policy tree depth. Evaluated on SWE-bench Lite instances using two foundation models across four intent representations (68 evaluated runs, 62 resolved), SPINE preserves task resolution performance while achieving 100% (344/344) write-time provenance stamping compared to 0% for other representations. SPINE fundamentally replaces implicit trace reasoning with explicit, queryable semantic lineage by construction.

1 INTRODUCTION

Autonomous systems are increasingly expected to interpret stakeholder intents and transform them into executable behaviors. However, current intent-driven architectures provide limited support for preserving the semantic relationship between stakeholder intent and the executable policies ultimately enforced at runtime. While existing approaches progressively refine intents into lower-level plans or policies while maintaining execution traces or reasoning logs, they generally do not explicitly represent the semantic relationships introduced during refinement. Consequently, it becomes difficult to determine which stakeholder intent an enforced action serves, through which sequence of policy refinements it was derived, or whether the attributes of the original intent has been faithfully preserved throughout refinement. Recent work by Meta (Wang et al., 2025b) on intent-driven network management showed that semantic attributes of the original intent, such as temporal constraints, may be omitted during refinement, leading to intent drift between stakeholder objectives and executable policies.

Large language models (LLMs) are increasingly used to realize this refinement: a model can read an intent stated in natural language and act on it as a general-purpose software agent (Dzeparoska et al., 2023; Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025a). Many agent workflows include a planning step, referred to here as refinement, that translates the intent into the steps or rules the agent should follow. Beyond the omission of semantic intent attributes during plan generation, two other problems follow: the rules produced from the plan are not reliably applied during execution, and the execution record does not indicate which refinement step each action corresponds to, so intent-execution lineage is reconstructed after rather than retrieved from the recorded workflow (Deshpande et al., 2025). The absence of an explicit semantic refinement model also limits explainability and accountability. Explanations are typically reconstructed retrospectively from execution logs or language-model reasoning rather than obtained directly from an explicit representation. Likewise, although autonomous systems record execution histories, they generally do not preserve the

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

semantic relationships between stakeholder intent, intermediate steps, and runtime actions, making it difficult to identify how a decision was made or who was responsible for each refinement step.

This paper introduces SPINE, a structured policy refinement framework that explicitly represents the transformation of stakeholder intent into structured, enforceable policies across abstraction layers. SPINE organizes policies into a hierarchy of declarative, definitive, and imperative policies connected through explicit parent-child relationships. A unified policy model and semantic refinement rules preserve policy semantics throughout refinement, producing an explicit intent-to-execution lineage that enables traceability, accountability, explainability, and semantic intent preservation by design.

SPINE is independent of the underlying intent assurance engine, cleanly decoupling policy representation from policy generation. Rather than proposing a specific refinement algorithm, SPINE provides a unified semantic representation for organizing, validating, and traversing policies produced during refinement. Consequently, the framework integrates seamlessly with language-model-based, symbolic, rule-based, or human-authored assurance engines without altering the representation. In this paper, we instantiate SPINE using an LLM-based assurance engine. We assume an intent formalization component ingests raw stakeholder input and yields a well-specified declarative policy P_0 adhering to the SPINE policy model, placing intent clarification and formal policy compliance verification outside the scope of this work. Instead, we focus on demonstrating how SPINE's hierarchical representation enables deterministic bi-directional tracing—from stakeholder intent down to refinement steps and runtime execution, and vice versa. The contributions of this paper are summarized as follows:

- We introduce a structured policy refinement hierarchy in which declarative, definitive, and imperative policies share a unified policy model and are connected by explicit semantic lineage from stakeholder intent to executable policies.
- We define semantic refinement rules that preserve intent attributes throughout refinement.
- We demonstrate how the resulting representation provides intrinsic traceability, accountability, explainability, and semantic intent preservation.
- We evaluate SPINE on three SWE-bench Lite instances (Jimenez et al., 2024), categorized in SWE-bench Verified by intent specificity and fix size (OpenAI, 2024), using two frontier foundation models (`claude-sonnet-4-6` and `gpt-5.6-terra`), demonstrating 100% $(344/344)$ write-time provenance stamping and perfect traceability and accountability (1.00) with zero task resolution penalty.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 formulates the problem. Section 4 presents the SPINE framework, including its architecture and semantic policy refinement model. Section 5 describes the implementation. Section 6 describes the experimental setup and Section 7 presents the results and the limitations. Finally, Section 8 concludes the paper and outlines future work.

2 Related Work

Policy-Based Management and Intent Refinement: Transforming intent into behavior is well-studied in policy management (Bandara et al., 2004), self-adaptive systems (Kramer & Magee, 2007; Weyns, 2020; Khakpour et al., 2012), and intent-based networking (Leivadeas & Falkner, 2023a; Jacobs et al., 2018; Dzeparoska et al., 2021b; Leivadeas & Falkner, 2023b). These frameworks refine objectives into operational policies, but treat outputs as endpoints rather than preserving refinement process history. RFC 9315 decouples intent from policy via closed autonomic loops (Clemm et al., 2022), which recent work extends from declarative to imperative representations (Dzeparoska et al., 2021a; 2023; 2024).

LLM Agents and Trace Evaluation: Agentic LLM systems execute natural-language goals by interleaving reasoning and tool use (Yao et al., 2023; Wang et al., 2024a; Guo et al., 2024; Wu et al., 2023; Xi et al., 2025). However, their logs capture execution history rather than explicit refinement lineage (Staufer et al., 2026; Chan et al., 2024). Interpreting such traces post hoc is notoriously difficult: Deshpande et al. (2025) reports low error-localization accuracy in agent logs, while MAST taxonomy attributes many agent failures to architectural design (Cemri et al., 2025).

Planning Frameworks and Intent Systems: Existing plan-then-execute workflows—including ReAct structures in SWE-agent (Yang et al., 2024), CodeAct in OpenHands (Wang et al., 2024b; 2025a), Plan-and-Solve (Wang et al., 2023), Plan-and-Act (Erdogan et al., 2025), and control-flow blueprints (Del Rosario et al., 2025)—lack policy identifiers and lineage metadata, preventing write-time attribution. Large-scale systems like Confucius (Wang et al., 2025b) demonstrate graph-based plan decomposition, but rely on external human validation rather than retaining explicit, queryable semantic lineage.

Provenance and Runtime Assurance: General provenance models such as W3C PROV (Moreau & Missier, 2013) and PROV-AGENT (Souza et al., 2025) record agent activities but omit structural policy refinement steps. Similarly, runtime assurance mechanisms like AgentSpec (Wang et al., 2026) (reactive enforcement) and VERIGUARD (Miculicich et al., 2025) (proactive verification) check policy compliance but do not track policy origins. SPINE complements these approaches by providing an explicit, write-time policy hierarchy that links every runtime action back to the refinement steps and originating intent.

3 PROBLEM STATEMENT

We consider the problem of semantic intent-to-policy refinement in autonomous systems. Given a stakeholder intent expressed as a declarative policy P_0 and a runtime assurance engine, the objective is to generate a hierarchy of increasingly operational policies $P_0 \rightsquigarrow P_1 \rightsquigarrow \cdots \rightsquigarrow P_n$ while explicitly preserving the semantic relationships introduced at each step. For every policy enforced at runtime, the system must mechanically answer four core questions:

1. Which stakeholder intent does this policy serve, and are intent's attributes correctly propagated through refinement?
2. Through which sequence of refinement steps was it derived?
3. Who defined and who enforced each refinement step?
4. How was this policy generated and enforced?

Existing systems record execution traces or workflow provenance but lack explicit semantic refinement relationships, requiring post hoc log reconstruction to answer these questions. Therefore, a complete solution must preserve semantic intent by design and provide an explicit refinement lineage that enables mechanical traceability, accountability, and explanation.

4 METHODOLOGY

SPINE is a semantic policy refinement framework for autonomous systems that provides an explicit representation of how stakeholder intent is progressively transformed into executable policies. Rather than treating policy refinement as a sequence of independent transformations, SPINE organizes the refinement process as a hierarchy of semantically related policies connected through explicit parent-child relationships. This representation enables semantic intent preservation, traceability, accountability, and intrinsic explainability as properties of the refinement model itself.

SPINE assumes the existence of two external components: (i) an intent formalization component that represents stakeholder intent as a declarative policy, and (ii) a runtime assurance engine responsible for generating and enforcing operational policies. The framework introduced in this paper governs the semantic structure of the refinement process between these components. It defines a unified policy representation, three levels of policy abstraction, formal refinement rules, and the policy refinement tree that records the semantic lineage from stakeholder intent to runtime enforcement.

The remainder of this section first presents an overview of the framework before introducing the policy model, abstraction hierarchy, refinement rules, and the properties that emerge from the resulting refinement structure.

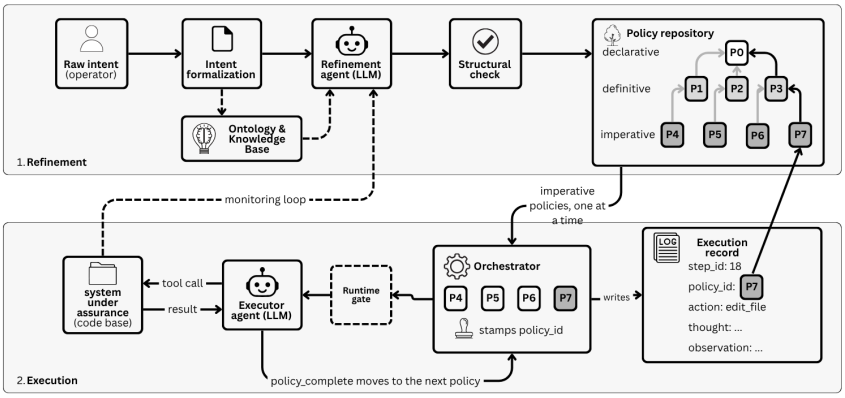

Figure 1: SPINE architecture. A raw intent is formalized and refined by the LLM into a policy tree that passes a structural check before entering the policy repository. The orchestrator loops through the imperative policies, hands each to the executor, receives the action and observation back, and writes the record, stamping the active policy identifier from its own state. From that identifier, the bold path walks up the parent pointers to the root intent.

4.1 SPINE Overview

Figure 1 presents the SPINE framework, instantiated here in LLM-based refinement and execution agents. A stakeholder's intent is first represented as a declarative policy by an intent formalization component. The runtime assurance engine then progressively refines this policy into increasingly operational policies according to the SPINE refinement model. The resulting hierarchy is stored in the policy repository together with the metadata required for verification and enforcement. During refinement, the runtime assurance engine consults domain ontologies and knowledge bases to derive policies consistent with stakeholder intent. The ontology defines the semantic concepts and relationships of the application domain, while the knowledge base provides the domain facts, operational constraints, and invariants required during refinement. Every generated policy is represented according to the SPINE policy model and incorporated into the refinement hierarchy by establishing an explicit parent-child relationship with the policy from which it was derived. Once an imperative policy is admitted to the refinement hierarchy, it becomes available for runtime enforcement. Every runtime action is subsequently associated with the imperative policy responsible for its execution, allowing the complete semantic lineage from stakeholder intent to enforced action to be retrieved through traversal of a refinement tree.

4.2 Policy Representation

SPINE defines policies using a uniform tuple $P = (D, E, A, C, CT)$, where D is the Definer, E is the Enforcer, A is the prescribed Action, $C = (R, S, T)$ specifies operational constraints (Resource R, Spatial boundary S, Temporal window T), and CT denotes environmental Context.

Each policy includes metadata $PMD = (ID, DM, EX, PR, AP)$ capturing its identifier (ID), enforcement domain (DM), expiration (EX), priority (PR), and an autonomic-permission flag (AP) specifying whether human oversight is required.

4.2.1 Policy Abstraction Hierarchy

SPINE organizes policies into a three-level hierarchy that progressively increases operational detail while retaining the unified tuple structure:

- **Declarative Policies**: High-level stakeholder intent specifying *what* to achieve without prescribing implementation (e.g., *"Protect sensitive files"*).
- **Definitive Policies**: Operational decisions decomposing abstract goals into functional responsibilities that explain *how* intent is met (e.g., *authenticate users before access"*, *au-*

thorize access by classification", *encrypt sensitive files"*, *monitor operations"*, and *record audit events"*).

- **Imperative Policies**: Concrete, directly executable rules with precise operational constraints (e.g., *deny read requests when clearance is lower than classification"*, *encrypt files using AES-256"*, *audit every write operation"*, and *lock accounts after three failed attempts"*).

Because all share the same policy representation, refinement specializes attributes monotonically rather than translating between heterogeneous policy languages, enabling uniform semantic verification across the hierarchy (see Appendix Table 5 for the complete tree of "Protect sensitive files").

4.3 Policy Representation Invariants

SPINE separates policy generation from policy representation. The assurance engine is responsible for determining the policies required to satisfy a stakeholder intent. The refinement strategy employed by the assurance engine is intentionally left unspecified and may consist of symbolic planning, rule-based reasoning, human-defined procedures, or language-model reasoning.

Let P denote a policy as defined in Section 4.2 and let $\mathcal{P}$ be the set of all policies generated by the assurance engine. SPINE requires every policy $P_i \in \mathcal{P}$ to satisfy the following representation invariants.

Invariant 1 (Uniform Representation). Every policy shall be represented using the unified policy model $P_i = (D_i, E_i, A_i, C_i, CT_i)$. Regardless of the refinement mechanism employed, all policies therefore possess the same semantic representation.

Invariant 2 (Unique Parent). Every non-root policy maintains exactly one parent policy: $\forall P_i \in \mathcal{P} \setminus \{P_0\},\ \exists!\, P_j \in \mathcal{P}$ such that $Parent(P_i) = P_j$.

Invariant 3 (Attribute Propagation). Let P_i be the parent of P_j. For every semantic attribute $x \in \{A, R, S, T, CT\}$, the child policy shall either preserve or specialize the corresponding attribute of its parent, $x_c \preceq x_p$, where $\preceq$ denotes semantic specialization. Consequently, semantic attributes are never silently discarded during refinement, preventing intent drift caused by omitted resource, spatial, temporal, or contextual constraints.

Invariant 4 (Abstraction Consistency). Every policy belongs to exactly one abstraction level, $Level(P) \in \{$*Declarative*, *Definitive*, *Imperative*$\}$, and refinement proceeds monotonically toward increasing operational specificity, *Declarative* $\rightarrow$ *Definitive* $\rightarrow$ *Imperative*. No refinement may increase the abstraction level of an existing policy.

Invariant 5 (Accountability). Every policy explicitly records both the entity responsible for defining the policy and the entity responsible for enforcing it: $D_i \neq \emptyset$ and $E_i \neq \emptyset$ for every $P_i \in \mathcal{P}$. These attributes establish accountability independently of the refinement algorithm that generated the policy.

4.4 Policy Refinement Tree

The representation invariants introduced above ensure that every generated policy has a uniform structure and an explicit parent relation. SPINE uses these parent relations to organize policies into a policy refinement tree. The tree is the primary artifact through which SPINE records semantic lineage from stakeholder intent to runtime enforcement.

A policy refinement tree is a rooted directed tree $T = (V, E)$ where each vertex $v \in V$ represents a policy and each directed edge $(P_i, P_j) \in E$ indicates that policy P_j was generated as a refinement of policy P_i. The root P_0 is the declarative policy representing the stakeholder intent. Internal nodes correspond to intermediate policies that operationalize the intent, while terminal nodes correspond to imperative policies that may be enforced by the runtime assurance engine. SPINE models the refinement hierarchy as a tree rather than a general directed acyclic graph because the purpose of the structure is to capture *realized semantic lineage*, not all possible policy dependencies. During

execution, the runtime assurance engine may consider multiple candidate refinements, but only the selected refinement path contributes to a particular enforced action. Consequently, every generated policy admitted to the refinement hierarchy has exactly one semantic parent: the policy whose refinement decision produced it. This unique-parent property gives every policy a unique path to the root intent. Formally, for every policy $P_i \neq P_0$, $\exists!\, P_j \in V$ such that $(P_j, P_i) \in E$. The unique path from the root to a policy P_i is denoted as $Path(P_i) = \langle P_0, P_1, \ldots, P_i \rangle$. This path records the sequence of refinement decisions that led from the stakeholder intent to P_i. If P_i is an imperative policy that produces or authorizes a runtime action (a), then $Path(P_i)$ also provides the semantic lineage for a.

An execution record is $r = (k, t, ID(P^*), A(P^*), m, a, o)$: the step k, the time t, the identifier and action of the policy P^* active when the record was written, the provenance mode m, and the agent's action a and observation o. The lineage of a record is $Path(P^*)$, so every record resolves to the intent by a walk of at most $\mathrm{depth}(T)$ hops.

4.5 Properties of SPINE

Given the structural invariants, SPINE satisfies four key properties:

1. **Semantic Intent Preservation**: Ensures stakeholder intent meaning is preserved across refinement. Follows from Invariants 1 (Uniform Representation) and 3 (Attribute Propagation), which enforce consistent semantic structure and attribute specialization.

2. **Traceability**: Guarantees recovery of the complete refinement lineage from any leaf policy back to the root intent. Follows from Invariant 2 (Unique Parent), ensuring a unique path to P_0.

3. **Accountability**: Identifies responsible entities for every policy in the tree. Follows from Invariant 5 (Accountability Metadata), requiring explicit recording of Definer D and Enforcer E.

4. **Explanation Fidelity**: This property follows from Invariant 2 (Unique Parent), which establishes a unique refinement path, and Invariant 1 (Uniform Representation), which preserves the complete semantic information at every refinement step. Consistent with structural theories of explanation, which view explanations as causal or derivational relationships rather than isolated outcomes (Pearl, 2009; Miller, 2019), we define the explanation of policy P as its unique refinement path:

$$\mathrm{Explain}(P) = \mathrm{Path}(P) = \langle P_0, P_1, \ldots, P \rangle$$

   Traversing this static parent chain yields an intrinsic, complete explanation by construction.

5 Implementation

SPINE defines a technology-agnostic architectural sequence for intent processing: intent formalization, hierarchical policy refinement, boundary enforcement, and runtime observability. In this study, we instantiate SPINE within an LLM-based software engineering agent, as illustrated in Figure 1.

5.1 Intent Formalization, Refinement, and Validation

The pipeline begins by formalizing raw operator input into a root declarative policy P_0 using few-shot prompts and structured output schemas. An LLM assurance component progressively refines P_0 into a policy tree spanning declarative, definitive, and imperative abstraction tiers. Each node retains its identifier, parent pointer, abstraction level, and derivation rationale. Prior to execution, a rule-based check validates structural integrity by enforcing three invariants: (i) the root node must be declarative, (ii) all parent identifiers must resolve to existing nodes, and (iii) abstraction levels must increase monotonically from root to leaf (see Appendix G for a complete validated path from *astropy-12907*)

5.2 Orchestration and Enforcement

An orchestrator situated between the policy tree and the executor manages runtime enforcement. Declarative and definitive policies are enforced prior to runtime during refinement and setup. For imperative leaf policies, the orchestrator passes instructions sequentially to the executor agent. To ensure reliable provenance, the orchestrator, rather than the LLM executor, stamps the active policy identifier ($ID(P^*)$) directly onto each execution record from its internal system state. Decoupling provenance stamping from the LLM prevents hallucinated, misattributed, or omitted policy identifiers during tool use. Downstream observability hooks (e.g., sandboxing or linting) can be integrated after action logging without altering the underlying policy schema. Each execution run links to its corresponding refinement tree and prompt request via content hashes, establishing a verifiable audit trail across artifacts. The orchestrator also makes the tree binding. When the 18 policy trees of this study were instead appended to the request and run without the orchestrator, none of their 322 records carried a policy identifier, apart from five that the model mentioned in its stated reasons. One of those trees stated that no change should be made: the executor given it as context edited the code and resolved the issue, whereas the orchestrator, given the same tree, made no edits.

6 Experimental Setup

6.1 Benchmark, Instance Selection, and Models

We evaluate SPINE using SWE-bench Lite (Jimenez et al., 2024), which uses containerized validation and regression testing to evaluate bug fixes in real GitHub repositories. Using SWE-bench Verified annotations (OpenAI, 2024), we sampled instances across specification and fix-size tiers (seed 20): the pilot *astropy-12907* (specified, medium fix; fixed before sampling), *pytest-11143* (underspecified, small fix), and *sympy-15011* (underspecified, medium fix). These instances address nested compound model separability matrices, assertion rewriting failures, and matrix symbol `SyntaxError` bugs, respectively.

Two foundation models, `claude-sonnet-4-6` and `gpt-5.6-terra`, serve as both refinement and execution engines under fixed decoding parameters. The execution environment is a 30-turn LangGraph scaffold (LangChain, Inc., 2024) modeled after SWE-agent (Yang et al., 2024), providing repo search, file navigation, and editing tools. All representation settings share the identical scaffold; SPINE additionally exposes a `policy_complete` signal to drive orchestration transitions. We compare four representations of the intent at execution time: Raw Prompt, the intent executed directly; Free-Form Steps, a plan in prose; Trajectory Format, a sequence of thought, action and expected-observation steps; and SPINE.

6.2 Requests, Ground Truth, Scoring, and Metrics

Requests: Requests pair the instance problem statement with setting-specific templates (Appendix A). Refinement requests apply to Free-Form Steps, Trajectory Format, and SPINE. For execution, whole-artifact injection appends the full refined intent after a plan header, whereas orchestrated execution delivers imperative policies sequentially. To mitigate prompt format sensitivity (Sclar et al., 2024), all prompts are frozen and SHA-256 hashed. Execution artifacts record their own request hash alongside the hash of the source refinement artifact.

Ground Truth and Scoring: Reference policy trees, answer keys, and rulings sheets were derived manually from problem statements. Answer keys map required refinement steps to the problem statement. Scoring is manual: a keyword search against the answer key drafts each score, and a human reads the artifact and confirms or overrides every draft. Refinement steps receive partial credit ($0, \frac{1}{2}, 1$) under an adjudicated rubric (Version 2; Appendix C).

Refinement artifacts are evaluated on four metrics. Let $\mathcal{P}$ denote the policies in a SPINE artifact or the steps in a prose or trajectory artifact, and $\mathcal{P}^-$ the same set excluding the root. Traceability is $\mathrm{TC} = |\{P \in \mathcal{P}^- : Path(P) \text{ reaches } P_0\}|/|\mathcal{P}^-|$, the share of non-root policies or steps whose lineage reaches P_0; a step without a parent pointer receives 0. Accountability completeness is $\mathrm{AC} = |\{P \in \mathcal{P} : D(P), E(P) \text{ named}\}|/|\mathcal{P}|$, the share naming both a definer and an enforcer. Explanation fidelity is $\mathrm{EF} = |\{P \in \mathcal{P} : \text{rationale present}\}|/|\mathcal{P}|$, the share containing a rationale. Semantic intent fidelity is $\mathrm{SIF} = \frac{1}{5}\sum_{f \in \{A,R,S,T,CT\}} s_f(P_0)$, where $s_f \in \{0, \frac{1}{2}, 1\}$ indicates whether field f of the

root policy preserves the corresponding intent value. TC, AC, and EF are also computed from the structural check in Section 5.1; the human-scored values are reported and cross-checked against the computed ones. Execution runs are scored on outcome, cost, behavior, and provenance. Outcome is the task resolution returned by SWE-bench's Docker evaluation. Cost comprises vendor-reported tokens, execution turns, wall-clock time, and storage per record in bytes; tokens, turns, and time are compared only within the same model. Behavior captures the number and composition of records and run-level flags such as turn-budget exhaustion. For provenance, a record r is recorded when its stamped identifier names a policy whose lineage $Path(P^*)$ reaches the intent; runtime provenance is the share of records satisfying this condition.

Repeats and reporting. Each instance, setting, and model is run three times end to end. A repeat consists of one refinement followed by its corresponding execution, and artifacts are scored per repeat, following prior agentic studies (Cuadron et al., 2025). Resolution rates are reported with Wilson confidence intervals. We perform no hypothesis tests: with three instances, an exact paired permutation test has only $2^3 = 8$ sign assignments, making the smallest attainable two-sided p value $2/8 = 0.25$ by construction. The intervals therefore describe the uncertainty. Following deployment-aware evaluation practice (Manik & Wang, 2026), every run artifact also records instrumentation behavior, allowing anomalous cells to be traced to their cause; flags and counts are reported in Appendix D.

7 RESULTS

As Table 1 shows, 69 runs were executed across the four representations, with 68 reaching SWE-bench's Docker evaluation and 62 resolving (Jimenez et al., 2024). Raw Prompt resolved 15/18 runs, Free-Form Steps 17/18, Trajectory Format 14/15, and SPINE 16/17; one `gpt-5.6-terra` SPINE run on *astropy-12907* stopped before evaluation. Resolution rates across evaluated runs were 0.83, 0.94, 0.93, and 0.94, respectively, with 95% confidence intervals of [0.61, 0.94], [0.74, 0.99], [0.70, 0.99], and [0.73, 0.99]. The intervals overlap substantially, so SPINE neither improves nor degrades resolution, and the small differences are largely associated with the instance: *astropy-12907* resolved 22/23 evaluated runs, *pytest-11143* 23/23, and *sympy-15011* 17/22.

7.1 PROVENANCE

Provenance is measured at both refinement and execution. All 18 SPINE refinements contain parent pointers resolving to existing policies, giving full traceability, whereas Free-Form Steps and Trajectory Format contain no such pointers, yielding zero traceability. Trajectory Format steps instead carry stated reasons (Yao et al., 2023), which contribute to explanation fidelity. Table 2 reports the four scores; for Free-Form Steps and Trajectory Format, semantic intent fidelity is capped at 0.5 per field because the intent is expressed in prose rather than root-policy fields. SPINE produced 7–11 policies per refinement, compared with 4–42 steps for Trajectory Format. Of 18 Trajectory artifacts, 10 parsed directly, 5 under a predefined rule selecting the model's last complete value, and 3 contained no complete value (Tam et al., 2024). All 18 SPINE trees passed the structural check; malformed Trajectory artifacts were left unrepaired and therefore produced no execution.

At execution, all 344 SPINE records written under an active policy carry a system-generated policy identifier, compared with 0 of 757 records across the three free-execution representations. The 344 excludes the 8 closing `done` records required after the final policy, which bring the SPINE total to 352. Per-representation counts are given in Table 3, with the exit breakdown in Appendix D.

Observability. The provenance fields in Table 4 are the only additions SPINE makes to an ordinary agent trace and map directly to OpenTelemetry span attributes. We exported all 18 SPINE runs as OpenTelemetry traces in OTLP/JSON using the OpenInference conventions adopted by TRAIL (Deshpande et al., 2025); the mapping is given in Appendix J. The traces contain 549 spans: 18 runs, 179 policies, and 352 records. Each record written under an active policy reaches the root through three to seven policy spans, while the eight closing `done` calls attach to the run span. Provenance is not inferred during export: the trace ID derives from the execution-artifact hash, record parents use the stamped policy identifier, and policy parents use the tree's parent pointers. Because

Table 1: Runs resolved by the SWE-bench Docker harness out of runs executed per cell (three, except where a trajectory failed to parse). †One run stopped before evaluation and is counted as unresolved.

	`claude-sonnet-4-6`			`gpt-5.6-terra`		
Representation	astropy	pytest	sympy	astropy	pytest	sympy
Raw Prompt	3/3	3/3	0/3	3/3	3/3	3/3
Free-Form Steps	2/3	3/3	3/3	3/3	3/3	3/3
Trajectory Format	3/3	3/3	2/3	3/3	2/2	1/1
SPINE	3/3	3/3	2/3	2/3†	3/3	3/3

Table 2: Refinement scores averaged over the scored artifacts of each representation (18 Free-Form Steps, 15 Trajectory Format, 18 SPINE)

Metric	Free-Form Steps	Trajectory Format	SPINE
Traceability	0.00	0.00	1.00
Accountability completeness	0.00	0.00	1.00
Explanation fidelity	0.00	1.00	1.00
Semantic intent fidelity	0.30	0.30	0.83

the traces use the standard OTLP/JSON format, policy attribution becomes a parent-traversal query in any compatible tracing backend.

7.2 Cost and Overhead

Provenance adds little storage and some execution time. The three provenance fields occupy 76–79 bytes, or 2.7–5.0 percent of a record. Per turn, SPINE's latency is comparable to the other representations. Per run, SPINE adds one `policy_complete` turn per imperative policy: a SPINE run takes 81.5 seconds of execution with `claude-sonnet-4-6`, against 41.0 to 46.1 seconds for the other representations, and stays within the others' range with `gpt-5.6-terra`. Refinement tokens, storage, and latency are reported in full in Appendix H.

Limitations. Four limitations bound our findings: (a) we evaluate three single-pass Python SWE-bench Lite instances, omitting specified small-fix intents, non-software domains, and post-execution re-refinement; (b) lineage holds by construction, verifying traceability rather than semantic correctness of policy trees; (c) we considered one model per vendor confounds serving-stack effects, and three instances yield wide Wilson intervals; (d) our experiment results lack external raters and runtime policy compliance gates, which can be done using AgentSpec (Wang et al., 2026) or VeriGuard (Miculicich et al., 2025).

8 Conclusion

This paper presented SPINE, a semantic policy refinement framework that organizes intent into an explicit hierarchy of declarative, definitive, and imperative policies. By establishing parent-child lineage, SPINE renders traceability, accountability, explanation fidelity, and semantic intent preservation intrinsic properties of the representation rather than post hoc reconstructions. Evaluated on SWE-bench within an LLM agentic workflow, SPINE preserves task resolution performance while establishing write-time provenance stamping at a modest storage cost and one additional executor turn per imperative policy (Section 7.2). Because SPINE records serialize into standard agent-trace formats (Deshpande et al., 2025), attribution data can be directly consumed by existing observability tools. Future work will extend SPINE toward an end-to-end runtime assurance pipeline, incorporating ontology-driven refinement, proactive compliance verification, and multi-domain evaluation.

REPRODUCIBILITY STATEMENT

Every request is frozen and verified by SHA-256 hash before the call that uses it, one artifact of record exists per cell and repeat, and each artifact stores the hash of the request that produced it and, for execution, of the refinement artifact it ran from. Every number in this paper is computed from those artifacts by three scripts, `build_tables.py`, `storage_cost.py` and `latency_estimate.py`, which write the result tables in Section 7 and the appendix, and a headline file that the text was checked against; the trace export of Section 7.1 is produced by a fourth script from the same artifacts. The supplementary material contains the request templates of Appendix A with the hash of every instantiated request, all refinement and execution artifacts, the Docker evaluation summaries, the ground-truth policy tree, answer key and rulings sheet for each instance, Version 2 of the scoring rubric, the adjudicated scorecards with a locator for every score, the four scripts, and the exported traces. Model versions, decoding settings, the turn budget, and the SWE-bench harness version are recorded in the artifacts and named in Appendix D.

ETHICS STATEMENT

This work evaluates LLM agents on public software engineering benchmark data, SWE-bench Lite and the SWE-bench Verified annotations, used under their published licenses, and involves no human subjects. The mechanism it studies records which policy an agent's action served; it does not judge whether the action was correct, and it should not be read as certifying agent behavior. Recorded attribution can support accountability, and it can equally be used to monitor people who operate agents; deployments should state who may read the records and for what purpose.

AI USE STATEMENT

Large language models are the object of study in this work: the two models named in Section 6 produced every refinement and execution artifact that the paper measures. The authors also used an LLM assistant to help draft and edit the paper, find related work, give feedback on the experimental design, assist in writing the Python code for the experiments and analysis, and interpret results. We did not use generative AI tools to generate data sets or to formulate or prove mathematical claims. No LLM produced a reported score: every refinement score was adjudicated by a human, and execution measures are computed by scripts from the stored artifacts. We reviewed all AI-assisted text and code, checked every reported number by regenerating the tables from the artifacts, and take full responsibility for the content.

REFERENCES

Arosha K. Bandara, Emil C. Lupu, Jonathan Moffett, and Alessandra Russo. A goal-based approach to policy refinement. In *Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004)*, pp. 229–239, 2004.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

Alan Chan, Carson Ezell, Max Kaufmann, Kevin Wei, Lewis Hammond, Herbie Bradley, Emma Bluemke, Nitarshan Rajkumar, David Krueger, Noam Kolt, Lennart Heim, and Markus Anderljung. Visibility into AI agents. In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency (FAccT)*, pp. 958–973, Rio de Janeiro, Brazil, 2024. ACM. doi: 10.1145/3630106.3658948.

Alexander Clemm, Laurent Ciavaglia, Lisandro Zambenedetti Granville, and Jeff Tantsura. Intent-Based Networking - Concepts and Definitions. RFC 9315, Internet Research Task Force (IRTF), October 2022.

Alejandro Cuadron et al. The danger of overthinking: Examining the reasoning-action dilemma in agentic tasks. *arXiv preprint arXiv:2502.08235*, 2025.

Ron F. Del Rosario, Klaudia Krawiecka, and Christian Schroeder de Witt. Architecting resilient LLM agents: A guide to secure plan-then-execute implementations. *arXiv preprint arXiv:2509.08646*, 2025.

Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. TRAIL: Trace reasoning and agentic issue localization. *arXiv preprint arXiv:2505.08638*, 2025.

Kristina Dzeparoska, Nasim Beigi-Mohammadi, Ali Tizghadam, and Alberto Leon-Garcia. Towards a self-driving management system for the automated realization of intents. *IEEE Access*, 9:159882–159907, 2021a.

Kristina Dzeparoska, Nasim Beigi-Mohammadi, Ali Tizghadam, and Alberto Leon-Garcia. Towards a self-driving management system for the automated realization of intents. *IEEE Access*, 9:159882–159907, 2021b. doi: 10.1109/ACCESS.2021.3129990.

Kristina Dzeparoska, Jieyu Lin, Ali Tizghadam, and Alberto Leon-Garcia. LLM-based policy generation for intent-based management of applications. In *2023 19th International Conference on Network and Service Management (CNSM)*, pp. 1–7, 2023.

Kristina Dzeparoska, Ali Tizghadam, and Alberto Leon-Garcia. Intent assurance using LLMs guided by intent drift. In *2024 IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–7, 2024.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-act: Improving planning of agents for long-horizon tasks. In *International Conference on Machine Learning (ICML)*, 2025.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. In *Proceedings of the 33rd International Joint Conference on Artificial Intelligence (IJCAI-24)*, pp. 8048–8057, 2024. doi: 10.24963/ijcai.2024/890.

Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. Refining network intents for self-driving networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks (SelfDN '18)*, pp. 15–21, 2018. doi: 10.1145/3229584.3229590.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Narges Khakpour, Saeed Jalili, Carolyn Talcott, Marjan Sirjani, and Mohammad Reza Mousavi. Formal modeling of evolving self-adaptive systems. *Science of Computer Programming*, 78(1): 3–26, 2012.

Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering (FOSE '07)*, pp. 259–268. IEEE Computer Society, 2007. doi: 10.1109/FOSE.2007.19.

LangChain, Inc. LangGraph. `https://github.com/langchain-ai/langgraph`, 2024.

Aris Leivadeas and Matthias Falkner. A survey on intent-based networking. *IEEE Communications Surveys & Tutorials*, 25(1):625–655, 2023a. doi: 10.1109/COMST.2022.3215919.

Aris Leivadeas and Matthias Falkner. A survey on intent-based networking. *IEEE Communications Surveys and Tutorials*, 25(1):625–655, 2023b. doi: 10.1109/COMST.2022.3215919.

Md Motaleb Hossen Manik and Ge Wang. Unified deployment-aware evaluation of open reasoning language models. *arXiv preprint arXiv:2604.07035*, 2026.

Lesly Miculicich, Mihir Parmar, Hamid Palangi, Krishnamurthy Dvijotham, Mirko Montanari, Tomas Pfister, and Long T. Le. VeriGuard: Enhancing LLM agent safety via verified code generation, 2025. URL `https://arxiv.org/abs/2510.05156`.

Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence*, 267:1–38, 2019. ISSN 0004-3702.

Luc Moreau and Paolo Missier. PROV-DM: The PROV data model. W3c recommendation, World Wide Web Consortium (W3C), 2013. https://www.w3.org/TR/prov-dm/.

OpenAI. Introducing SWE-bench verified. `https://openai.com/index/introducing-swe-bench-verified/`, August 2024.

Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2nd edition, 2009.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design or: How I learned to start worrying about prompt formatting. In *International Conference on Learning Representations (ICLR)*, 2024.

Renan Souza, Amal Gueroudji, Stephen DeWitt, Daniel Rosendo, Tirthankar Ghosal, Robert B. Ross, Prasanna Balaprakash, and Rafael Ferreira da Silva. PROV-AGENT: Unified provenance for tracking AI agent interactions in agentic workflows. In *2025 IEEE 21st International Conference on e-Science (e-Science)*, pp. 467–473, 2025. doi: 10.1109/eScience65000.2025.00093.

Leon Staufer, Kevin Feng, Kevin Wei, Luke Bailey, Yawen Duan, Mick Yang, A. Pinar Ozisik, Stephen Casper, and Noam Kolt. The 2025 AI agent index: Documenting technical and safety features of deployed agentic AI systems. In *Proceedings of the 2026 ACM Conference on Fairness, Accountability, and Transparency (FAccT)*, Montreal, QC, Canada, 2026. arXiv:2602.17753.

Zhi Rui Tam, Cheng-Kuang Wu, Yi-Lin Tsai, Chieh-Yen Lin, Hung-yi Lee, and Yun-Nung Chen. Let me speak freely? A study on the impact of format restrictions on performance of large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, 2024.

Haoyu Wang, Christopher M. Poskitt, and Jun Sun. AgentSpec: Customizable runtime enforcement for safe and reliable LLM agents. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*, Rio de Janeiro, Brazil, 2026. arXiv:2503.18666.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2609–2634, 2023.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024a. doi: 10.1007/s11704-024-40231-1.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *International Conference on Machine Learning (ICML)*, 2024b.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025a.

Zhaodong Wang, Samuel Lin, Guanqing Yan, Soudeh Ghorbani, Minlan Yu, Jiawei Zhou, Nathan Hu, Lopa Baruah, Sam Peters, Srikanth Kamath, Jerry Yang, and Ying Zhang. Intent-driven network management with multi-agent LLMs: The Confucius framework. In *Proceedings of the ACM SIGCOMM Conference*, pp. 347–362, Coimbra, Portugal, 2025b. ACM. doi: 10.1145/3718958.3750537.

Danny Weyns. *An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective*. Wiley-IEEE Computer Society Press, 2020. ISBN 9781119574941.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025. doi: 10.1007/s11432-024-4222-0.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

A Request Templates by Setting

Each template below is the fixed text of a request with the variable parts shown in angle brackets. `<problem statement>` is the instance's problem statement, copied verbatim from the benchmark. The fixed text is identical for every instance and both models; the SHA-256 of each instantiated request is stored in the run artifact it belongs to and is verified before each live call. Tools are supplied to the model as tool definitions by the scaffold, not as prompt text, and the turn budget (30) is a scaffold parameter.

Composition. A refinement request exists only for three settings. Under Free-Form Steps, the problem statement is followed by one instruction: make no changes to the repository, and show a plan for resolving the issue with the steps to take and the reasoning. Under Trajectory Format, the same instruction is followed by an output format that asks for a JSON array of steps, each with a thought, an action naming one of the scaffold's tools with its arguments, and an expected observation. Under SPINE, the same instruction is followed by the policy model of Section 4.2, the refinement rules, one worked example on an unrelated intent, and the JSON schema the output must follow. The Raw Prompt setting has no refinement request.

Execution requests exist for every setting. Each is one instruction paragraph followed by the problem statement: fix the issue using the tools and call `done` with a one-line summary when the fix is applied. In the Raw Prompt setting this is the whole request, so the agent acts on the issue alone. Under whole-artifact injection, for Free-Form Steps and Trajectory Format, and for the runs of the tree without the orchestrator, the refined intent produced in the same repeat is appended after the line "PLAN (follow it; deviate only with stated reason)". Under orchestrated execution, the tree is not appended. A protocol paragraph explains that the plan arrives one policy at a time and that `policy_complete` requests the next one, and the current imperative policy follows as JSON. The tools are supplied to the model as tool definitions by the scaffold, and the turn budget of 30 is a scaffold parameter. Refinement requests and the Raw Prompt execution request are identical for both models, but execution requests that carry a refined intent differ between models in the artifact they embed.

A.1 Refinement requests

Free-Form Steps.

```
You are a software engineering agent working in a repository. The repository has the
    following reported issue.

<problem_statement>
<problem statement>
</problem_statement>
```

```
Do not make any changes to the repository in this session. Show me your plan for resolving
    this issue: the steps you would take and your reasoning.
```

Trajectory Format.

```
You are a software engineering agent working in a repository. The repository has the
    following reported issue.

<problem_statement>
<problem statement>
</problem_statement>

Do not make any changes to the repository in this session. Show me your plan for resolving
    this issue: the steps you would take and your reasoning.

OUTPUT FORMAT
Emit each step of your plan as one step of an agent trajectory. Return a JSON array
of step objects and nothing else. Each step object has exactly these
fields:

  thought               your reasoning for taking this step
  action                an object with:
                          name  one of: search_repo, open_file,
                                edit_file, done
                          args  an object of arguments for that tool
  expected_observation  what you expect this step to return

Order the steps by execution order.
```

SPINE. The policy model, refinement rules, worked example, and output schema of Section 4.2.

```
You are a software engineering agent working in a repository. The repository has the
    following reported issue.

<problem_statement>
<problem statement>
</problem_statement>

Do not make any changes to the repository in this session. Show me your plan for resolving
    this issue: the steps you would take and your reasoning.

Refine the reported issue above, a single high-level natural-language intent,
into a tree of structured policies.

POLICY MODEL
Every policy is a tuple P = (D, E, A, C(R, S, T), CT):

<tuple definition: definer, enforcer, action, constraints (resource, spatial, temporal),
    context, with the rules each field must obey; supplementary file lines 55-99>

State is NOT a tuple field. The monitored runtime condition (which actor, which
file, the current weather, a threshold crossing) is observability data captured
at run time, not part of any policy.

<abstraction level definitions: declarative, definitive, imperative, and the enforcer of
    each; supplementary file lines 105-110>

<refinement rules: root restates the intent; every non-root policy names its parent_id;
    one-sentence rationale per policy; constraint values may narrow, widen, or change down a
    branch; supplementary file lines 112-125>

============== WORKED EXAMPLE (a different, neutral intent for format only) ==============
<worked example: a cloud CPU-cap intent refined into five policies in the same format;
    supplementary file lines 128-157>
================================================================================

NOW DO THE SAME FOR THE REPORTED ISSUE ABOVE.

Decompose the intent into its sub-goals first, then refine each toward imperative,
executable policies. Use the policy model and levels exactly as defined above.
Assign the definer by level: human operator for the root, assurance engine for
every refined policy.

OUTPUT FORMAT
Return ONLY valid JSON, no prose, no markdown fences. Schema:
<output schema: a JSON object with the intent and a policies array; each policy has id,
    parent_id, level, definer, enforcer, action, constraints, context, rationale;
    supplementary file lines 169-184>
```

```
Each of resource, spatial, and temporal may be a single value or an array if the
policy has several. Use "always" for temporal when there is no time restriction.
Use null for any constraint that does not apply. Every non-root policy must have a
parent_id that exists in the list, and definer "assurance engine".
```

A.2 EXECUTION REQUESTS

Raw Prompt.

```
You are a software engineering agent working in a repository. Use the tools to inspect and
    modify the code; paths are relative to the repository root. Fix the following issue.
    When the fix is applied, call done with a one-line summary.

ISSUE:
<problem statement>
```

Whole-artifact injection (Free-Form Steps, Trajectory Format, and the tree without the orchestrator). The refined intent is the artifact produced in the same repeat.

```
You are a software engineering agent working in a repository. Use the tools to inspect and
    modify the code; paths are relative to the repository root. Fix the following issue.
    When the fix is applied, call done with a one-line summary.

ISSUE:
<problem statement>

PLAN (follow it; deviate only with stated reason):
<refined intent produced in the same repeat: Free-Form Steps text, Trajectory Format JSON
    array, or SPINE Structured policy tree JSON>
```

Orchestrated execution (SPINE). The scaffold adds the `policy_complete` tool; after each call to it, the next imperative policy replaces the current one.

```
You are a software engineering agent working in a repository. Use the tools to inspect and
    modify the code; paths are relative to the repository root. Fix the following issue.
    When the fix is applied, call done with a one-line summary.

ISSUE:
<problem statement>

PLAN PROTOCOL:
The plan for this issue is provided one policy at a time. Work on the current policy with the
    tools. When the current policy's action is complete, or cannot be performed with the
    available tools, call policy_complete with a one-line summary and the next policy will
    be provided.

CURRENT POLICY (<n> of <number of imperative policies>):
<current imperative policy as JSON: id, parent_id, level, definer, enforcer, action,
    constraints, context, rationale>
```

B SUPPLEMENTARY MATERIAL

The supplementary archive holds, per instance and cell: the instantiated requests with their SHA-256 hashes, the refinement and execution artifacts of record, the Docker evaluation summaries, and the exported traces. Per instance it holds the ground-truth policy tree, answer key and rulings sheet. It also holds the scorecards, one workbook per instance, with the adjudicated value and a locator for every score, and the scripts that regenerate every table and number in this paper from the artifacts.

C SCORING RUBRIC

Version 2 of the partial-credit rubric, the rules under which every score was adjudicated, is included in the supplementary archive as a single document. It states, for each of the six coverage obligations and the five root fields of semantic intent fidelity, what earns 1, 0.5 and 0, and the rules on before and after the fix, on tests framed as reproduction or verification, and on the order in which a tree is read. Across the three instances, 80.6, 82.4 and 83.3 percent of coverage drafts and 100, 96.5 and 98.8 percent of semantic-intent-fidelity drafts were accepted unchanged.

Table 3: Execution records per representation, one per tool call, and how many carry a system-written policy identifier; under SPINE the closing `done` after the last policy belongs to no policy and is not counted under an active policy.

Representation	Records	Under an active policy	Stamped with a policy id
Raw Prompt	286	–	0
Free-Form Steps	277	–	0
Trajectory Format	194	–	0
SPINE	352	344	344

D Instrument Log

The flags of Section 6.2, counted over the 69 execution runs of the four representations and the 54 refinement calls. Exit status: 61 runs ended with the executor's `done`, 7 exhausted the 30-turn budget (5 with `claude-sonnet-4-6`, 2 with `gpt-5.6-terra`; 3 under SPINE, 2 under Raw Prompt, 1 under Free-Form Steps, 1 under Trajectory Format), and 1 stopped early (`gpt-5.6-terra` under SPINE on astropy-12907, never evaluated). Stream errors: none. Evaluation environment: SWE-bench harness 5.0.2 with Docker 29.7.2 and Python 3.12.3. Freeze check: every request was hash-verified before the call, and the hash is stored in the artifact. Configuration readback: `claude-sonnet-4-6` applied temperature 0.0 in all 36 of its runs; for `gpt-5.6-terra` the artifacts record that the Responses API applied no temperature. Refinement: all 54 calls completed without truncation; 18 of 18 trees passed the structural checks; 15 of 18 trajectories parsed, 10 directly, 5 by the last-complete-value rule, and 3 not at all. Malformed or retried tool calls are not counted in this experiment. Under SPINE, 8 runs ended with a clean closing `done` after the last policy, 6 issued `done` while a policy was still open, 3 stopped on budget exhaustion, and 1 stopped with no `done`.

Table 3 gives the execution records of each representation.

E Record Schema

Table 4 gives the fields of one execution record, as written by the orchestrator for every tool call (Section 5.2).

Table 4: Fields are taken from the pilot run for astropy-12907 (`claude-sonnet-4-6` as planner and executor, run 1) under SPINE (schema `exec_record/2`). The writer column states which component supplies the value.

Field	Writer	Content
`step_id`	orchestrator	position of the record in the run
`timestamp`	orchestrator	wall-clock time at write
`policy_id`	orchestrator	identifier of the policy active when the action was taken
`policy_action`	orchestrator	the action verb of that policy, copied from the tree
`provenance_mode`	orchestrator	how the policy identifier was obtained (`recorded`)
`gate_flag`	orchestrator	whether a runtime gate intervened on this step
`action.name`	model	the tool called (`open_file`, `edit_file`, `search_repo`, `policy_complete`, `done`)
`action.args`	model	the tool arguments
`action.thought`	model	the model's stated reason for the call
`observation`	environment	the tool result returned to the model
`usage`, `wall_s`	orchestrator	tokens and seconds consumed by the step

F Illustrative Example: Protecting Sensitive Files

As a running example, consider a simple intent *protect files at /srv/app/config*. Intent formalization produces the root declarative policy representing the stakeholder objective of protecting files within

Table 5: Refined policy tree for the `protect_files(/srv/app/config)` intent. Confidentiality, integrity, and availability appear as the (C), (I), and (A) branches.

Policy	Definer	Enforcer	Action	Constraint (R; S; T)
P0 dec / root	Human Operator	Assurance Engine	protect	R: file; S: /srv/app/config; T: always
P1 def / P0	Assurance Engine	Analyze	classify	R: file; S: /srv/app/config; T: always
P1.1 imp / P1	Assurance Engine	Execute	tag	R: file; S: /srv/app/config; T: always
P2 def / P0	Assurance Engine	Monitor	monitor	R: file operation; S: /srv/app/config; T: always
P2.1 def / P2	Assurance Engine	Analyze	authorize	R: operation, actor; S: /srv/app/config; T: always
P3 def / P0	Assurance Engine	Plan	decide (C)	R: plan; S: /srv/app/config; T: always
P3.1 imp / P3	Assurance Engine	Execute	block	R: read operation; S: /srv/app/config; T: always
P3.2 imp / P3	Assurance Engine	Execute	block	R: copy operation; S: /srv/app/config $\rightarrow$ external; T: always
P3.3 imp / P3	Assurance Engine	Execute	block	R: move operation; S: /srv/app/config $\rightarrow$ external; T: always
P4 def / P0	Assurance Engine	Plan	decide (I)	R: plan; S: /srv/app/config; T: always
P4.1 imp / P4	Assurance Engine	Execute	block	R: write operation; S: /srv/app/config; T: always
P4.2 imp / P4	Assurance Engine	Execute	block	R: permission change; S: /srv/app/config; T: always
P5 def / P0	Assurance Engine	Plan	decide (A)	R: plan; S: /srv/app/config; T: always
P5.1 imp / P5	Assurance Engine	Execute	block	R: delete operation; S: /srv/app/config; T: always
P6 def / P0	Assurance Engine	Analyze	detect	R: resolved target; S: /srv/app/config & resolved targets; T: always
P6.1 imp / P6	Assurance Engine	Execute	reject	R: resolved target; S: outside /srv/app/config; T: always

the specified path. The assurance engine then refines this policy into definitive policies corresponding to the security objectives of confidentiality, integrity, and availability according to Table 5. Each definitive policy is further refined into imperative policies that govern concrete runtime actions. For example, the confidentiality policy produces an imperative policy that blocks unauthorized read operations on sensitive files.

Every policy records a pointer to its parent, allowing the complete refinement chain, from the imperative policy blocking a read, through the confidentiality policy, to the root declarative policy and the originating stakeholder intent, to be recovered by simple traversal of the refinement tree.

G One Lineage from Intent to Imperative

The pilot intent, astropy-12907, reports that `separability_matrix` returns the wrong matrix for a nested compound model: two models that the function reports as separable when combined at the top level are reported as not separable once the same combination is nested inside another model. Table 6 shows one root-to-leaf path of the tree that refinement produced for it, with every field value taken from the stored artifact. The full tree holds eleven policies: one declarative root, seven definitive policies enforced by Monitor, Analyze, or Plan, and three imperative policies enforced by Execute. The enforcer names are the phases of the autonomic control loop that the architecture inherits, and they tell the reader which part of the system was obliged to carry each policy out. P0 restates the reported issue as a single declarative policy: defined by the human operator, enforced by the Orchestrator, with the action "fix" scoped to the module that contains the failing function. It says what must be achieved and nothing about how. P3.1, three levels down through P3, is defined by

Table 6: An example of a root-to-leaf path from the refined policy tree of astropy-12907 using SPINE and claude-sonnet-4-6 during the first run. The field values are taken directly from the stored artifact; with a total of 11 policies in the whole tree.

Field	P0	P3.1	P4
Level	declarative	definitive	imperative
Parent	none	P3	P3.1
Definer	human operator	assurance engine	assurance engine
Enforcer	Orchestrator	Plan	Execute
Action	fix	design	patch
Resource	separability_matrix function in astropy.modeling.separable	corrected block-diagonal combination rule for '&' operator in separability_matrix	_separable function (or equivalent combiner) in astropy/modeling/separable.py
Spatial	astropy.modeling.separable module	astropy.modeling.separable module	astropy/modeling/separable.py
Temporal	always	always	always
Context	production	production	production

the assurance engine and enforced by Plan. Its resource has narrowed from the function itself to the specific combination rule that the fix must correct, yet it still touches no code. P4 is the first policy the executor can act on: enforced by Execute, with the action "patch", the resource narrowed to a helper function, and the spatial constraint narrowed from a module to a file path. Read downward, the rows illustrate progressively tighter constraints. Read upward, they explain why the patch was made. This is because P4 names P3.1 as its parent, P3.1 names P3, and P3 names P0. That walk is three hops long, and the rationale stored with each policy is the explanation at each hop. Temporal and context stay constant along this path, which is what a repository task with no time window would be expected to produce. Whether P4 names the right function is a scoring question, answered in Section 6.

H Cost and Overhead

Refinement cost. Refinement calls average 1,400–9,300 total tokens per instance and model—a fraction of execution-phase context totals. As shown in Appendix Figure 2, representations differ primarily in token distribution rather than overall volume. SPINE incurs higher input costs (2,200–6,400 tokens) due to its schema definitions, rules, and worked examples, but maintains stable output generation (1,900–2,600 tokens). Conversely, Trajectory Format exhibits the highest output token variance (up to 8,800 tokens with `claude-sonnet-4-6`), with 5 of 9 trajectory refinements requiring fallback parsing. SPINE's peak input cost on *pytest-11143* stems from prompt duplication of the problem statement within the schema's `intent` field—a formatting artifact that does not impact scoring.

Storage cost. Provenance is carried in three record fields, `policy_id`, `policy_action`, and `provenance_mode`, which occupy 76–79 bytes of a record averaging 1,500–2,800 bytes, or 2.7–5.0 percent. Including the activation queue and transition log, SPINE's provenance overhead accounts for 4.1–8.8 percent of the run's stored record bytes. In free-execution records, an auditor reconstructs the reason for an action from the thought and observation, which average 1,183–1,852 characters per record; the 78-byte stamp is 15 to 24 times smaller and can be queried directly.

Latency. Per turn, SPINE is comparable to the other representations (Table 7): 3.0 to 4.4 seconds with `claude-sonnet-4-6` against 3.1 to 3.9, and 1.9 to 2.3 seconds with `gpt-5.6-terra` against 1.7 to 2.9. A SPINE run takes 81.5 seconds of execution with `claude-sonnet-4-6`, against 41.0 to 46.1 seconds for the other representations, and 37.9 seconds with `gpt-5.6-terra`, within the others' range of 28.0 to 41.8. The difference comes from turns: SPINE adds one `policy_complete` turn per imperative policy, and its `claude-sonnet-4-6` runs also took more working turns. A linear fit of wall time on tokens (R squared 0.88 and 0.77) attributes 5.5 of those seconds with `claude-sonnet-4-6`, and none with `gpt-5.6-terra`, to the execu-

Table 7: Latency per representation: mean seconds per execution turn, with the refinement call and the execution time of the median run in parentheses (refinement+execution), in seconds. Wall time includes the executor re-sending the accumulated context on every turn and the provider's serving time, and SPINE turn counts include one `policy_complete` turn per imperative policy.

	`claude-sonnet-4-6`			`gpt-5.6-terra`		
Representation	astropy	pytest	sympy	astropy	pytest	sympy
Raw Prompt	3.4 (0+37)	3.3 (0+15)	3.2 (0+77)	1.7 (0+23)	2.2 (0+26)	2.1 (0+61)
Free-Form Steps	3.7 (28+21)	3.1 (17+23)	3.9 (19+95)	2.0 (14+33)	2.6 (14+39)	2.4 (32+57)
Trajectory Format	3.3 (96+43)	3.1 (33+20)	3.4 (112+33)	1.8 (12+23)	2.9 (10+35)	1.9 (10+40)
SPINE	4.4 (31+73)	3.0 (41+60)	4.1 (31+119)	1.9 (24+29)	2.3 (16+38)	2.2 (19+51)

tor re-sending its history. Figures are compared within a model only, since wall time includes the provider's serving time.

I Refinement Tokens

Figure 2 gives the mean refinement tokens per call discussed in Appendix H.

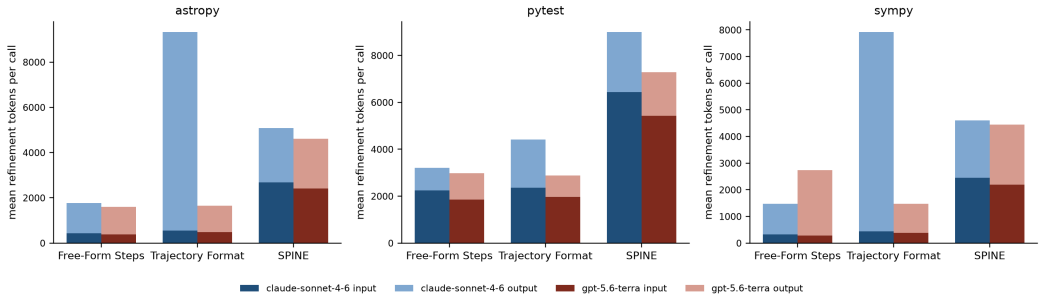

Figure 2: Mean refinement tokens per call, split into input and output, by representation and instance. Raw Prompt has no refinement call, so it does not appear. Token counts are compared within a model only.

J Trace Export

Section 7.1 reports the 18 SPINE runs exported as OpenTelemetry traces. Table 8 gives the mapping, and Listing 1 shows one root-to-leaf path from the pilot trace (astropy-12907, `claude-sonnet-4-6`, run 1): the run span, the policy span for P4, and the record span for the `edit_file` call written under P4. Long attribute values are cut at the ellipsis; nothing else is changed. Span identifiers are derived from the artifact hash and the policy or record key, so the same artifact always exports to the same trace.

Table 8: How SPINE objects map onto OpenTelemetry spans in the export. Span kinds follow the OpenInference conventions.

SPINE object	Span	Parent span	Attributes
Execution run	agent span, one per trace	none	artifact and plan hashes, model, instance (on the resource)
Policy	chain span, one per policy in the tree	its parent policy	id, parent id, level, definer, enforcer, action, resource, spatial, temporal, context, rationale
Execution record	tool span, one per tool call	the policy active at write time	tool name, arguments, step id, policy id, policy action

Listing 1: One root-to-leaf path from the pilot trace, as exported.

```
resource: service.name=spine, spine.instance=astropy__astropy-12907,
          spine.executor_model=claude-sonnet-4-6, spine.run=1,
          spine.artifact.sha256=7f8d2c0c..., spine.plan.sha256=8a5593c1...

span "run"       id=28c9f026207cc29f  parent=none
  openinference.span.kind = AGENT

span "P4 patch"  id=cb501eabd9b249a7  parent=d4e6a7856873bda2 (P3.1)
  openinference.span.kind = CHAIN
  spine.policy.id        = P4
  spine.policy.parent_id = P3.1
  spine.policy.level     = imperative
  spine.policy.definer   = assurance engine
  spine.policy.enforcer  = Execute
  spine.policy.action    = patch
  spine.policy.resource  = _separable function (or equivalent combiner)
                           in astropy/modeling/separable.py
  spine.policy.spatial   = astropy/modeling/separable.py
  spine.policy.temporal  = always
  spine.policy.context   = production
  spine.policy.rationale = Directly modify the code path that handles
                           the '&' (join/concatenate) operator so that...

span "edit_file" id=e73e79c380a08d06  parent=cb501eabd9b249a7 (P4)
  openinference.span.kind = TOOL
  tool.name              = edit_file
  input.value            = {"path": "astropy/modeling/separable.py",
                            "old": "cright[-right.shape[0]:, ...", ...}
  input.mime_type        = application/json
  spine.record.step_id   = 4
  spine.policy.id        = P4
  spine.policy.action    = patch
```

K Statistical Details

Resolution rates in Section 7 carry Wilson score intervals at 95 percent, computed from the resolved and evaluated counts per representation. No hypothesis tests are run, for the reason given in Section 6.2.