Language Server CLI Empowers Language Agents with Process Rewards

Yifan Zhang **Lanser Team**
Princeton University
yifzhang@princeton.edu

October 24, 2025

Abstract

Large language models routinely hallucinate APIs and mislocalize edits, while language servers compute verified, IDE-grade facts about real code. We present `Lanser-CLI`, a CLI-first orchestration layer that pins and mediates a Language Server Protocol (LSP) server for coding agents and CI, exposing deterministic, replayable workflows. Our position is that language servers provide not only *structural information* (definitions, references, types, diagnostics) but also an actionable *process reward*: machine-checked, step-wise signals that align an agent's planning loop with program reality.

`Lanser-CLI` contributes: (i) a robust addressing scheme beyond brittle `file:line:col` via a `Selector DSL` (symbolic, AST-path, and content-anchored selectors) with a principled relocation algorithm; (ii) deterministic `Analysis Bundles` that normalize Language Server responses and capture environment/capability metadata with stable content hashes; (iii) a safety envelope for mutating operations (rename, code actions) with preview, workspace jails, and Git-aware, transactional apply; and (iv) a process-reward functional derived from Language Server facts (diagnostic deltas, disambiguation confidence, and safe-apply checks) that is computable online and replayable offline. We formalize determinism under frozen snapshots and establish a monotonicity property for the process reward, making it suitable for process supervision and counterfactual analysis.

Project Page: https://github.com/yifanzhang-pro/lanser-cli

1 Introduction

Large language models (LLMs) have catalyzed a wave of coding agents, yet their textual guesses about static structure, side effects, and symbol identity routinely drift from reality. By contrast, Language Server Protocol (LSP) servers compute verifiable facts: definitions, references, types, diagnostics, and safe edits. We ask a concrete question:

How should Language Agents obtain structural information and process reward via Language Servers?

Our answer is twofold. First, a *CLI-first, agent-native* layer is the right binding for language servers in planner-act loops because CLIs compose with Unix tooling, serialize cleanly to artifacts,

and are easy to containerize and gate in CI. Second, the same layer can transform server facts into a *process reward* that supervises intermediate steps (plan, locate, verify, apply) rather than only terminal outcomes.

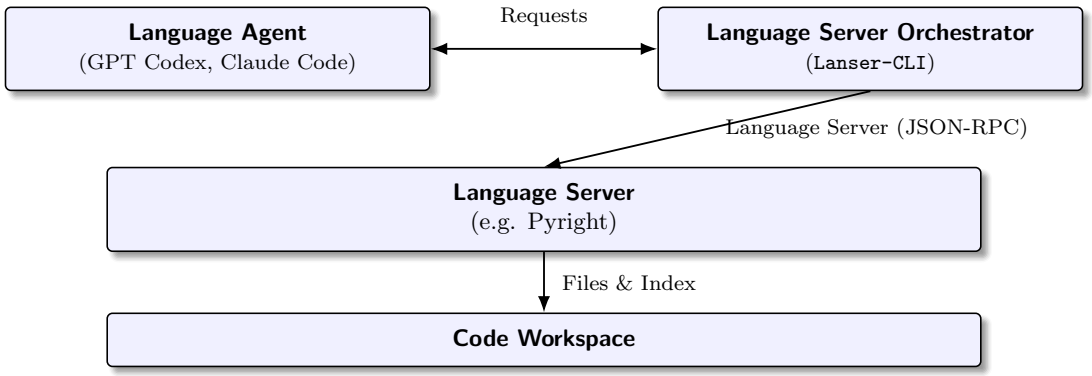

Figure 1 A language agent interacts with the `Lanser-CLI` orchestrator, which speaks JSON-RPC to a pinned LSP server (e.g., Pyright) over a concrete workspace. The orchestrator turns transient protocol sessions into stable artifacts.

`Lanser-CLI` is built around four goals that raw LSP usage does not guarantee: (i) determinism and replay; (ii) robust addressing that survives edits; (iii) safety for mutating operations; and (iv) process supervision via machine-checked signals correlated with task success. These goals shape the architecture and surface area of the CLI.

Rather than bespoke editor plug-ins or ad-hoc RPCs, `Lanser-CLI` turns language-server interactions into schema-validated JSON artifacts with explicit environment provenance and byte-stable hashing, yielding an auditable substrate for planner-act loops and CI. The artifacts support offline replay and enable counterfactual evaluation of agent decisions. Conceptually, `Lanser-CLI` separates three concerns: (i) *addressing* (what code element an agent intends), (ii) *analysis* (what the Language Server states about it), and (iii) *application* (how to enact a mutation safely). This separation allows plans to compose with clear contracts and measurable failure modes.

We instantiate against the Language Server Protocol (LSP) (Microsoft, 2025a) using Pyright for Python (Microsoft, 2025b).

Our `Lanser-CLI` offers: (i) a `Selector DSL` and `PositionSpec` union that addresses code semantically (symbol, AST path, content anchor) and survives edits (section 3); (ii) a deterministic repositioning algorithm with ambiguity surfacing and explicit evidence; (iii) `Analysis Bundles`with environment capture (server version, `positionEncoding`, interpreter, configuration digest) and a stable `bundleId` (section 5); (iv) safe mutating flows (previewed rename, transactional apply, workspace jail, dirty-worktree guardrails); (v) `Record/Replay` for byte-stable regeneration of past outputs under a frozen snapshot; (vi) an SDK and batch interface for high-throughput agent pipelines; (vii) stable symbol identifiers (`symbolId`) derived from structural fingerprints for cross-edit identity; and (viii) a *process-reward* functional that converts LSP-verified facts into per-step signals usable for process supervision and credit assignment (section 5).

Remark 1.1 (Bootstrapping)**.** `Lanser-CLI` **is utilized during its own development**: we run `Lanser-CLI` to prepare and preview refactors in this repository, validate schema changes against historical traces, and replay bundles in CI to detect nondeterminism.

2 System Design of Language Server CLI

2.1 Motivation: Bridging the Agent-Server Gap

While Language Servers provide a common vocabulary for code analysis, integrating them reliably into autonomous agent loops exposes fundamental gaps. To be effective at scale, agents require four key capabilities that raw Language Server interactions do not provide.

First, raw server outputs can be fragile; coordinates are brittle, and server defaults (e.g., `utf-16` indexing) often mismatch agent I/O, leading to off-by-one and encoding errors. This necessitates: (i) *determinism*, including stable response ordering, content hashing, and version pinning; and (ii) *robust addressing*, using selectors that can survive code edits and resist positional drift.

Second, automated edits are inherently risky. Agents therefore require (iii) *safety*, implemented as a set of guardrails for mutating operations.

Finally, a fourth piece is often overlooked: (iv) *process supervision*. Agents benefit from intermediate, verifiable feedback that is correlated with final task success (e.g., "diagnostics decreased," "rename is safe," "ambiguity resolved").

`Lanser-CLI` is designed to close these gaps by transforming inherently interactive language-server sessions into verifiable, replayable artifacts. This achieves *protocol grounding* for LLM agents—preferring machine-checked facts over model speculation—and exposes a shaped process reward computable directly from those artifacts.

2.2 Architecture Overview

The design of `Lanser-CLI` is centered on an *orchestrator* that mediates all agent-server communication. This component manages the language-server lifecycle (start/stop, capability negotiation, cancellation, restarts with backoff), synchronizes document state, and normalizes server responses, ultimately emitting `Analysis Bundles`.

Beyond session management, the orchestrator implements a lightweight cache to coalesce identical in-flight queries (a single-flight pattern) and serves subsequent callers from memoized bundles. A comprehensive tracing system captures all JSON-RPC frames and workspace digests, enabling `Record/Replay` to regenerate byte-stable outputs offline for auditing and testing.

Environment capture. Each `Analysis Bundle` records {`serverVersion, positionEncoding, pythonExe, pythonVersion, venvPath, configDigest, platform`}, enabling reproducibility checks and differential debugging across machines.

Contracts and invariants. All location lists are ordered by the total order $(\texttt{uri}, sL, sC, eL, eC)$ with stable tie-breakers. `bundleId` is the SHA-256 of a JCS-canonicalized subset of fields that excludes volatile timestamps. Given an identical workspace snapshot, server/version/encoding, and request, `Lanser-CLI` yields a byte-identical bundle; see theorem 5.1. Replayability extends to any scalar computed solely from bundle contents (e.g., the process reward in section 5).

3 Selectors and Repositioning

3.1 The `Selector DSL` for Robust Addressing

Agents require references that can survive edits, a need unmet by brittle `file:line:col` coordinates. The `Lanser-CLI Selector DSL` is designed to capture intent rather than absolute byte offsets by unifying multiple addressing strategies. Selectors are represented programmatically as a `PositionSpec` (a tagged union) and textually as a canonical string, which are resolved to concrete ranges by a deterministic relocation procedure.

PositionSpec (Structured Union). The `PositionSpec` defines the internal, structured representation for all selector types, enabling rich, programmatic specification:

- `Cursor: {kind:"cursor", uri, line, col, indexing:"utf-16|utf-8|codepoint"}`
- `Range: {kind:"range", uri, start:[l,c], end:[l,c]}`
- `Symbolic: {kind:"symbol", qualname:"pkg.mod:Class.method", role:"def|sig|body|doc", overload:0}`
- `AST path: {kind:"ast", path:[["module","pkg.mod"],["class","C"],["def","m"]]}`
- `Content anchor: {kind:"anchor", uri, snippet:"def load_data(", ctx:24, hash:"sha1:..."}`

All forms optionally carry `docVersion` (a document snapshot identifier) to pin relocation to a specific, known version of a file.

Canonical String Form. For CLI usage, logging, and human readability, the `Selector DSL` provides a compact, canonical string-based syntax that maps directly to the `PositionSpec` structures:

```
# Cursor/range
src/app.py@L42:C7
src/app.py@R(42,7->44,1)

# Symbolic
py://pkg.mod#Class.method:body
py://pkg.mod#function_name:sig

# AST path (subset)
ast://[module=pkg.mod]/[class=Class]/[def=method]/name[1]

# Content anchor (snippet + context N chars)
anchor://src/app.py#"def load_data("?ctx=24
```

3.2 Indexing Semantics and Encoding

A critical detail for coordinate-based selectors (`Cursor` and `Range`) is the handling of position encoding, a common source of off-by-one errors. `Lanser-CLI` negotiates `positionEncoding` with the server at `initialize`, preferring `utf-16` (per LSP specification) but also supporting `utf-8`.

While the server operates on its negotiated encoding, CLI I/O can be declared separately via `--index-io=utf-8|utf-16|codepoint`. We adopt *codepoint* to mean Unicode scalar values. When

server and CLI indexings differ, `Lanser-CLI` emits both coordinate systems in verbose mode and records the negotiated server-side encoding in bundle metadata. This explicitly surfaces and resolves the ambiguity of LSP's default UTF-16 indexing, which often mismatches the UTF-8 context common in client tools (Microsoft, 2025a).

3.3 Repositioning and Ambiguity Resolution

Beyond managing encodings, the primary challenge is ensuring selectors remain valid as code evolves. This is the task of the RELOCATE algorithm (algorithm 1), which resolves a (potentially stale) selector against the current workspace state and surfaces ambiguity with ranked, deterministic evidence.

Strategy. Given a selector, `Lanser-CLI` resolves its position as follows: (1) attempt a direct map via `docVersion` if the snapshot is available; (2) reparse the workspace and resolve symbolic/AST paths against the current code structure; (3) for content anchors, perform a fuzzy match using winnowed k-grams within a context window; (4) score all candidates and disambiguate using a deterministic scoring model.

Scoring. Let s_{ast} be an AST-kind match indicator, s_{module} a module-equivalence score, J_{token} token Jaccard, and s_{prox} a proximity score. All features are normalized to $[0, 1]$. We define a convex combination for ranking candidates:

$$\text{score}(s, c) \;=\; 0.5\, s_{\text{ast}} \;+\; 0.2\, s_{\text{module}} \;+\; 0.2\, J_{\text{token}} \;+\; 0.1\, s_{\text{prox}}. \tag{3.1}$$

Ties are broken lexicographically on $(\texttt{uri}, \texttt{range})$, inducing a total order. Each bundle records the weights, features, and normalization steps used for auditability, and surfaces top-k alternatives when $\max \text{score} < \tau$ (indicating ambiguity).

Correctness sketch. Under a frozen snapshot, symbolic and AST selectors resolve to a unique target or return `E/AMBIGUOUS`. For anchors, if the snippet hash matches within the context window and no conflicting exact matches exist, RELOCATE returns the original range with score 1.0; otherwise, it ranks candidates by the convex combination above. Deterministic sort keys ensure identical outputs across runs.

Error taxonomy. Bundles carry structured error codes and, where applicable, disambiguation candidates with scores and explanations. Common errors include: `E/NOT_FOUND`, `E/AMBIGUOUS`, `E/VERSION_SKEW`, and `E/INDEXING_MISMATCH`.

4 Interfaces, Bundles, and Safety

`Lanser-CLI` exposes its orchestration capabilities through a CLI-first interface, providing a robust command structure designed for both interactive use and integration into automated agent loops or CI pipelines. The interface is functionally partitioned to handle code navigation, safe mutations, batch processing, and artifact validation.

Algorithm 1 `Lanser-CLI` Repositioning (RELOCATE)

Require: Selector s, workspace W, optional snapshot v
Ensure: Ranked candidates $\mathcal{C}$ with explanations

$\mathcal{C} \leftarrow \emptyset$
if v is present and W has exact $\text{map}(s, v)$ **then return** $\{(\text{map}(s, v), 1.0)\}$
end if
if $s.kind \in \{\text{symbol}, \text{ast}\}$ **then**
    $\mathcal{A} \leftarrow \text{resolve\_structural}(s, W)$ ▷ module import graph + parser
    $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{A}$
end if
if $s.kind = \text{anchor}$ **then**
    $\mathcal{H} \leftarrow \text{fuzzy\_within\_ctx}(s.snippet, s.ctx; k{=}7, w{=}4)$
    $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{H}$
end if
for all $c \in \mathcal{C}$ **do**
    $c.score \leftarrow f(s, c)$ ▷ Eq. (1): deterministic weights
end for
$\mathcal{C} \leftarrow \text{sort\_desc}(\mathcal{C}, \text{score}, \text{uri}, \text{range})$
if $\mathcal{C} = \emptyset$ **then**
    return ERROR(`E/NOT_FOUND`)
end if
if ties or low top-score **then**
    attach `disambiguation` evidence
end if
return $\mathcal{C}[1..k]$

Navigation Interface. Read-only operations for structural analysis are grouped under a set of navigation commands. Standard Language Server queries such as `lanser def`, `refs`, `hover`, `symbols`, and `diag` are supported, each accepting any POSITIONSPEC (as defined in Section 3) as its input target. A dedicated `lanser locate` command serves to resolve abstract selectors into concrete, verifiable ranges, offering an optional preview of the targeted code.

Safe Mutations and Guardrails. Mutating operations are designed with a "safety-first" principle. For instance, `lanser rename` is gated by a preceding `prepare-rename` check and defaults to a preview-only mode; explicit application of the changes requires the `--apply` flag. This entire process is protected by a multi-layered safety envelope, including a "workspace jail" (to prevent writes outside the project root), configurable allow/deny path filters, and a dirty-worktree refusal (which can be overridden with `--allow-dirty`), directly addressing the safety requirements for automated edits.

Batch Processing and Tracing. To support high-throughput agent pipelines, `lanser batch` executes command queues from JSONL-formatted input and produces structured JSONL responses suitable for planners. Any command can emit a complete JSONL trace of orchestrator metadata and underlying JSON-RPC traffic via `--trace-file`. This trace enables `lanser trace replay` to

regenerate byte-stable outputs offline, ensuring deterministic reproducibility.

Schema Contracts and Bundles. To guarantee reliable integration, `Lanser-CLI` provides explicit, machine-readable contracts. The `lanser schema` subcommands can export and validate the JSON Schemas for both the SELECTOR DSL inputs and the resulting output artifacts (`Analysis Bundles`). This contract-based approach allows an agent to validate its payloads before execution and permits CI systems to gate changes based on schema compatibility. Deterministic bundles, which encapsulate facts and metadata from an operation, are the core data artifact of the system and are detailed in section 5.

5 Process Rewards from Structural Signals

Planner-act loops benefit from verifiable intermediate signals. We expose a shaped *process reward* computed from LSP-derived facts that (i) is available online during planning, (ii) is deterministic and replayable from bundles, and (iii) correlates with final task success.

Our instantiation is compatible with potential-based reward shaping from reinforcement learning (Ng et al., 1999), while grounding the potential in machine-checked program facts rather than latent model states.

Definition. Let D_t be the count of server diagnostics relevant to the current target at step t, $S_t \in \{0, 1\}$ indicate that all safety checks passed for a prospective mutation (prepare-rename accepted, workspace jail holds, no conflicts), and $\alpha_t \in [0, 1]$ denote the top disambiguation confidence for a selector resolution. Define

$$r_t \;=\; \alpha\,(D_{t-1} - D_t) \;+\; \beta\,S_t \;-\; \gamma\,(1 - \alpha_t), \tag{5.1}$$

with fixed $\alpha, \beta, \gamma \geq 0$ recorded in bundle metadata. The first term rewards diagnostic reduction, the second rewards safe readiness to apply, and the third penalizes residual ambiguity.

5.1 Deterministic Analysis Bundles

`Analysis Bundles` normalize Language Server payloads and pin environment metadata. Lists are deterministically ordered by $(\texttt{uri}, sL, sC, eL, eC)$ with explicit tiebreakers. Each bundle has a stable `bundleId` computed as a hash over a canonicalized subset of fields (excluding volatile timestamps).

Response Envelope.

```
{
  "version": "1.3",
  "bundleId": "sha256:...",
  "status": "ok",
  "request": {"cmd": "definition", "selector": {...}},
  "resolution": {"original": "...", "resolved": {...}, "disambiguation": [...]},
  "facts": {"definitions": [...], "hover": {...}, "provenance": "lsp"},
  "edits": {"workspaceEdit": null, "diff": null},
  "processReward": {
```

```
  "version": "pr-v1",
  "r": 0.73,
  "components": {"diag_delta": 1, "safety": 1, "ambiguity_penalty": 0.27},
  "weights": {"alpha":0.5,"beta":0.4,"gamma":0.1},
    "explanation": "Eq. (\\ref{eq:proc-reward}) over frozen snapshot"
  },
  "environment": {"server": {"name":"pyright","version":"1.1.406"},
  "positionEncoding":"utf-16","python":{"version":"3.11.6"}, ...},
  "capabilities": {"partialResult": false, "cancellable": true},
  "meta": {"exit_code": 0,
    "sorting_keys": ["uri","range[0]","range[1]","range[2]","range[3]"]
  }
}
```

Proposition 5.1 (Determinism under frozen snapshot)**.** Fix a workspace snapshot S, Language Server server binary and configuration (V, Π), negotiated `positionEncoding`, and request Q. Then `Lanser-CLI` produces identical bundles B across runs, i.e., `bundleId`(B) is constant.

Proof sketch. The orchestrator (i) enforces deterministic sorting; (ii) canonicalizes JSON via the JSON Canonicalization Scheme (JCS) Rundgren et al. (2020); (iii) records environment invariants in the envelope; and (iv) excludes non-deterministic fields from the hash domain. Given identical inputs, Language Server responses are a function of (S, V, Π); thus, the resulting canonical JSON, and hence `bundleId`, is invariant. □

Proposition 5.2 (Monotonicity of process reward under invariants)**.** Under a frozen snapshot and fixed toolchain (S, V, Π), suppose an agent step does not increase ambiguity ($\alpha_t \geq \alpha_{t-1}$) and does not violate safety ($S_t \geq S_{t-1}$), while weakly decreasing diagnostics ($D_t \leq D_{t-1}$). Then for any non-negative weights in (5.1), $r_t \geq 0$.

Proof sketch. Each term is non-negative under the stated conditions: $(D_{t-1} - D_t) \geq 0$, $S_t \geq 0$, and $(1 - \alpha_t) \leq (1 - \alpha_{t-1})$. With $\alpha, \beta, \gamma \geq 0$, the sum is non-negative. Because bundles are deterministic under Theorem 5.1, r_t is replayable. □

5.2 Editing and Guardrails

Transactional Edit Application. `Lanser-CLI` ensures that workspace edits are applied transactionally to maintain file system integrity. The process involves writing changes to temporary files, synchronizing them to disk via `fsync`, and then atomically replacing the original files using the `rename(2)` syscall. This procedure ensures that file metadata, including permissions, line endings, and character encoding, is preserved. As an alternative, `Lanser-CLI` can leverage `git apply --3way` for patch application, which enables robust conflict detection. If a merge conflict occurs, the system reports a structured `E/APPLY_CONFLICT` error, including machine-readable conflict hunks. Additional file system integrity checks are enforced; for example, attempting a case-only rename (e.g., `file.py` to `File.py`) on a case-insensitive file system is prohibited, resulting in an `E/FS_PERMISSIONS` error.

Algorithm 2 Guarded Rename (PreviewThenApply)

Require: selector s, new name n, mode $\in$ {`dry-run`, `apply`}

1: assert clean git worktree or `--allow-dirty`
2: **if** !`prepareRename`(s) **then return** Error
3: $E \leftarrow$ `textDocument/rename`(s, n) ▷ WorkspaceEdit preview
4: $D \leftarrow$ diff(E); emit preview; **if** mode=`dry-run` **then return** D
5: apply atomically with jail + filters; **if** conflict **then return** `E/APPLY_CONFLICT`
6: notify server via `didChange`; **return** success bundle with D

Threat Model and Safety Envelope. The primary threats during automated editing include: (i) incorrect selector resolution leading to unintended edits; (ii) partial application of changes due to system failure; (iii) "workspace escapes" where edits affect files outside the intended project root; (iv) use of stale configuration; and (v) mismatches in file encoding or position indexing.

`Lanser-CLI` implements a multi-layered safety envelope. Operations are preview-by-default (`--dry-run`). A "workspace jail" confines all file modifications to the project root, supplemented by explicit allow/deny path filters. Mutating operations require a clean Git working tree unless explicitly overridden (`--allow-dirty`). The system performs encoding detection and, in verbose mode, reports dual coordinates (e.g., `UTF-16` and `UTF-8`) to prevent indexing errors. Ambiguity in selectors is explicitly surfaced with confidence scores, and atomic application ensures that partial failures can be rolled back.

Safety trade-offs are exposed as explicit policy hooks for CI systems and planning agents. Key controls include `--deny-apply-on-ambiguous`, the `--workspace-jail` flag (enforced by default), and `--allow-dirty`. These controls allow operators to configure the desired balance between automation and safety.

Our guardrails complement established program-transformation and differencing tools such as GumTree (Falleri et al., 2014) and refactoring detectors like RefactoringMiner (Tsantalis et al., 2018), but focus on determinism, auditability, and CI-grade safety envelopes.

6 Related Work

Language servers and static analysis. The Language Server Protocol provides a transport-agnostic interface for definitions, references, diagnostics, and edits across IDEs and tools (Microsoft, 2025a). We build on Python's Pyright server for concrete instantiation (Microsoft, 2025b). Our selector design intersects with AST-aware differencing and refactoring ecosystems; while systems like GumTree (Falleri et al., 2014) and RefactoringMiner Tsantalis et al. (2018) focus on change extraction and refactoring detection, `Lanser-CLI` emphasizes deterministic resolution and replayable artifacts for agent loops.

Anchoring and robust localization. Content-anchored relocation in `Lanser-CLI` draws on local fingerprinting via winnowing Schleimer et al. (2003) and classical text-index structures such as suffix arrays Manber and Myers (1993), adapted to code-aware contexts and combined with structural signals.

Agents that use tools. Language-model agents that plan and call external tools include ReAct (Yao et al., 2022), PAL (Gao et al., 2023), and Toolformer (Schick et al., 2023). Unlike these, `Lanser-CLI` converts language-server outputs into deterministic `Analysis Bundles` and a process reward usable

for supervision and credit assignment inside the agent loop.

Process supervision and reward shaping. Step-level guidance for LMs via self-feedback or verbal reinforcement appears in Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023). Our process reward connects this line to potential-based reward shaping from RL (Ng et al., 1999), with the potential grounded in static-analysis facts rather than purely textual heuristics.

7 Conclusion

`Lanser-CLI` reframes how agents interact with language servers: determinism and replayability as first-class properties, robust addressing that resists drift, and safety rails that make automated edits auditable. Beyond structural facts, `Lanser-CLI` extracts a *process reward* that supervises intermediate steps with machine-checked evidence. By turning Language Server facts into stable `Analysis Bundles` and per-step signals, `Lanser-CLI` enables trustworthy planning, safer refactors, reproducible CI, and process-supervised learning signals for agent training.

References

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324, 2014.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, volume 202, pages 10764–10799. PMLR, 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

Microsoft. Language server protocol specification, version 3.17. `https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification`, 2025a. Accessed October 2025.

Microsoft. Pyright: Static type checker for Python. `https://github.com/microsoft/pyright`, 2025b. Accessed October 2025.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287. Citeseer, 1999.

Anders Rundgren, Bret Jordan, and Samuel Erdtman. Rfc 8785: Json canonicalization scheme (jcs), 2020.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, 2003.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th international conference on software engineering*, pages 483–494, 2018.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Appendix

A Selector Grammar and Escaping (EBNF)

```
selector := cursor | range | symbolic | astpath | anchor
cursor := path "@" "L" INT ":" "C" INT
range := path "@" "R(" INT "," INT "->" INT "," INT ")"
symbolic := "py://" moduleref "#" qualname ( ":" role )?
moduleref:= IDENT ( "." IDENT )*
qualname := IDENT ( "." IDENT | ":" IDENT )*
role := "def" | "sig" | "body" | "doc"
path := RELPATH | "file://" URI_PATH
anchor := "anchor://" path "#" quoted_snippet ( "?" "ctx=" INT )?
quoted_snippet := '"' { char | '\"' | '\/' } '"'
```

Escaping: percent-encode `# ? % " <space>` in anchor snippets and paths. Windows paths canonicalize to `file:///C:/...` (uppercase drive letter).

Overloads, properties, and descriptors. Overloaded functions can be targeted via `overload=`i. Properties use role `:sig` to target the getter signature; use `:def` to select the backing function object.

B Bundle Stability Rules

- Deterministic list ordering: (`uri`, sL, sC, eL, eC).
- `bundleId` := `sha256` over a JCS-canonicalized JSON of `(request, resolution, facts, edits, environment, capabilities, meta*)`, excluding volatile fields.
- Range encoding: flat `[sL,sC,eL,eC]` integer array.
- Size limits: cap references to 10^5 entries; mark truncation and expose a pagination cursor.
- Canonicalization: JSON Canonicalization Scheme (JCS) with UTF-8 encoding; `meta.hashing.algo = "sha256-jcs-v1"`.
- Dual coordinates: when CLI I/O differs from server encoding, include both coordinate systems in verbose traces; bundles retain server coordinates.

C Exit Codes

D Worked Example

Definition query.

```
lanser def py://pkg.mod#Class.method:sig --json
```

Returns a `Analysis Bundle` with the resolved range, hover signature, and environment metadata (`serverVersion=1.1.406`, `positionEncoding=utf-16`).

Code	Symbol	Meaning	Retryable
0	`OK`	Success	—
2	`E/BAD_SELECTOR_SYNTAX`	Selector parse error	No
3	`E/NOT_FOUND`	No resolvable target	Sometimes
4	`E/AMBIGUOUS`	Multiple candidates	Yes
10	`E/VERSION_SKEW`	Snapshot mismatch	Yes
64	`E/LS_TIMEOUT`	Server timeout	Yes
65	`E/LS_CRASH`	Server crashed	Yes
70	`E/APPLY_CONFLICT`	Patch could not be applied	Manual
71	`E/FS_PERMISSIONS`	Write denied	No
72	`E/UNSUPPORTED_CAP`	Server lacks capability	No
73	`E/REQUEST_CANCELLED`	Request was cancelled	Yes
74	`E/CONTENT_MODIFIED`	Content changed mid-request	Yes
75	`E/INDEXING_UNSUPPORTED`	IO indexing unsupported	No
76	`E/REPLAY_MISMATCH`	Trace/workspace digest mismatch	No

Rename.

```
lanser prepare-rename py://pkg.mod#load_data:def --json
lanser rename py://pkg.mod#load_data:def new_name --dry-run
lanser rename py://pkg.mod#load_data:def new_name --apply
```

The preview includes a unified diff; the apply path enforces workspace jail and dirty-repo policies.

E Process Reward Signals: Worked Examples

Signals and weights. We instantiate Eq. (5.1) with $(\alpha, \beta, \gamma) = (0.5, 0.4, 0.1)$.

Example 1: Diagnostic reduction, safe apply, confident resolution. An agent proposes to rename `load_data` to `read_data`. Pyright reduces relevant diagnostics from D_{t-1}=5 to D_t=2 after a dry-run, `prepareRename` succeeds and the workspace jail holds (S_t=1), and the selector relocation reports α_t=0.94. Then

$$r_t = 0.5 \cdot (5 - 2) + 0.4 \cdot 1 - 0.1 \cdot (1 - 0.94) \approx 1.5 + 0.4 - 0.006 = 1.894.$$

The bundle records {`"diag_delta": 3, "safety": 1, "ambiguity_penalty": 0.06`}.

Example 2: Ambiguous selector, no safety clearance. The agent attempts a refactor with unresolved imports. Diagnostics stagnate (D_{t-1}=7, D_t=7), safety checks fail (S_t=0), and ambiguity remains (α_t=0.62). Then $r_t = 0 - 0.4 - 0.1 \cdot 0.38 = -0.438$, discouraging application until ambiguity is resolved.

Replayability. Because `processReward` is computed from deterministic bundle contents and fixed weights, the same r_t is recovered by `lanser trace replay`. This supports offline evaluation and counterfactual policy analysis without re-running the language server.

Design note. The reward is *shaping*, not a replacement for task success metrics. It is intended for online guidance and offline process supervision, and is safe under Theorem 5.2 when the invariants hold.