Reinforcement Learning from Compiler and Language Server Feedback

Yifan Zhang
Princeton University
yifzhang@princeton.edu

Abstract

Coding agents fail when text-level guesses outrun program facts: they hallucinate APIs, drift to the wrong symbol, and apply edits without evidence that the workspace remains valid. Compilers, type checkers, and language servers already compute the missing supervision signal, in the form of diagnostics, symbol resolution, type information, references, and refactoring preconditions, but expose it through interfaces designed for human-driven IDEs rather than learning loops. We introduce **Reinforcement Learning from Compiler and Language Server Feedback** (RLCSF) together with `Lanser-CLI`, a CLI-first orchestration layer that exposes this signal to agents and CI. RLCSF treats each tool interaction as a transition and computes a shaped process reward from deterministic changes in diagnostics, selector confidence, and edit safety. `Lanser-CLI`, in turn, converts ephemeral LSP sessions into replayable `Analysis Bundles` with pinned environment metadata and stable content hashes. Its core mechanisms are robust selectors that go beyond `file:line:col`, deterministic bundle normalization, preview-first guarded mutations, and a reward functional whose potential-based component is replayable under frozen snapshots. We formalize determinism for canonical bundles and prove that componentwise-improving transitions receive non-negative reward in the undiscounted setting. Together, these pieces yield a practical substrate for process supervision of coding agents.

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

1 Introduction

Large language models (LLMs) are increasingly capable coding assistants, yet their predictions about program structure, side effects, and symbol identity remain ungrounded unless checked against the actual workspace. The mismatch is especially costly for autonomous agents: a plausible edit can target the wrong definition, break a hidden type invariant, or leave diagnostics worse than before. Compilers, type checkers, and language servers already compute the facts agents need, definitions, references, types, diagnostics, and refactoring preconditions, but they were built for interactive editing rather than for optimization loops.

This raises a concrete question:

How can coding agents learn and plan from compiler and language server feedback?

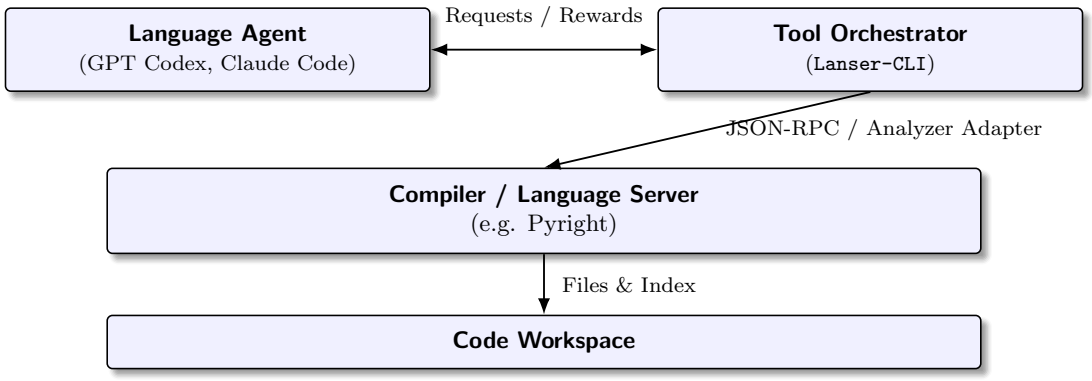

Figure 1 A language agent interacts with the `Lanser-CLI` orchestrator, which speaks JSON-RPC to a pinned language server or compiler-backed analyzer over a concrete workspace. The orchestrator turns transient protocol sessions into stable artifacts and transition rewards.

Our answer is RLCSF. Rather than treating tool feedback as a terminal pass/fail signal, RLCSF exposes dense, machine-checked feedback after each intermediate action, locating a symbol, requesting diagnostics, previewing a rename, or applying a guarded edit, and turns it into a transition-level reward suitable for planning, reinforcement learning, offline process supervision, and counterfactual evaluation.

Realizing this idea requires more than a thin wrapper around the Language Server Protocol. LSP sessions are stateful, positional, and difficult to replay; server defaults often disagree with agent-side encodings; and mutating operations can silently act on stale or ambiguous locations. `Lanser-CLI` addresses these issues with a CLI-first, agent-native contract that composes naturally with Unix tooling, serializes cleanly to JSONL artifacts, and is easy to containerize and gate in CI.

`Lanser-CLI` makes compiler and language server feedback usable as a learning signal through four mechanisms:

- A `Selector DSL` that addresses code through symbols, AST paths, content anchors, and explicit coordinate encodings, reducing reliance on brittle `file:line:col` references.
- `Analysis Bundles` that normalize tool responses, record environment and capability metadata, and carry stable content hashes for replay and auditing.
- Preview-first rename and apply flows protected by workspace jails, Git-aware staging, conflict reporting, and ambiguity checks.
- An RLCSF reward functional that converts diagnostic deltas, selector confidence, safety readiness, and structured tool errors into a replayable transition signal.

We instantiate the system against the Language Server Protocol (LSP) (Microsoft, 2025a) using Pyright for Python (Microsoft, 2025b). The formal results in Section 5 establish that bundle hashes are stable under frozen snapshots, and that the undiscounted reward is non-negative whenever every tracked component weakly improves and no tool error is raised.

2 System Design for Feedback-Oriented Language Agents

2.1 Bridging the Agent-Feedback Gap

Language servers were designed for interactive IDEs, not autonomous optimization loops. Operating them at agent scale raises four first-class requirements:

- *Determinism.* Equivalent requests should produce byte-stable artifacts after response normalization, version pinning, and content hashing.
- *Robust addressing.* Agents need selectors that survive edits, expose ambiguity, and make encoding conventions explicit.
- *Safe mutation.* Refactors must be previewed, confined to the workspace, checked for conflicts, and recoverable when application fails.
- *Dense supervision.* Intermediate tool feedback should be transformed into a verifiable signal that correlates with successful repair and refactoring.

`Lanser-CLI` addresses these requirements by turning interactive compiler and language server sessions into verifiable artifacts. The resulting interface gives LLM agents protocol grounding: model speculation is replaced by machine-checked facts, and any pair of adjacent bundles can be scored by a deterministic reward functional.

Remark 2.1 (Bootstrapping)**.** `Lanser-CLI` is used 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.

2.2 Architecture Overview

At the core of `Lanser-CLI` is an orchestrator that mediates all agent–tool communication. It manages the language server lifecycle (start/stop, capability negotiation, cancellation, and restart with backoff), synchronizes document state, normalizes server responses, and emits `Analysis Bundles`. Feedback from compilers and type checkers enters either through language server diagnostics or through adapters that expose the same bundle schema.

Beyond session management, the orchestrator coalesces identical in-flight queries via a single-flight cache and serves later callers from memoized bundles. A tracing layer records JSON-RPC frames and workspace digests so that `Record/Replay` can regenerate byte-stable outputs offline for auditing and regression testing.

Environment capture. Each `Analysis Bundle` records {`toolVersion, 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. The `bundleId` is the SHA-256 of a JCS-canonicalized subset of fields that excludes volatile timestamps and run-local trace data. Given an identical workspace snapshot, tool version, encoding, and request, `Lanser-CLI` yields a byte-identical hash-domain bundle under deterministic tool semantics (see Theorem 5.1). Replayability extends to any scalar computed from canonical bundle contents and recorded parameters; in particular, transition rewards are computed from ordered pairs of adjacent bundles.

3 Selectors and Repositioning

3.1 The `Selector DSL` for Robust Addressing

Agents need references that survive edits, but raw `file:line:col` coordinates do not. The `Lanser-CLI Selector DSL` captures *intent* rather than absolute byte offsets by unifying several addressing strategies under a single contract. Selectors are represented programmatically as a `PositionSpec` tagged union and textually as a canonical string; both forms resolve to concrete ranges through a deterministic relocation procedure.

PositionSpec. The `PositionSpec` is the internal, structured representation shared by all selector kinds:

- `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 that pins relocation to a known file version.

Canonical string form. For CLI usage, logging, and human readability, the `Selector DSL` provides a compact, canonical syntax that maps directly to `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

For coordinate-based selectors such as `Cursor` and `Range`, position encoding is a frequent source of off-by-one errors. `Lanser-CLI` negotiates `positionEncoding` with the server at `initialize`, preferring `utf-16` per the LSP specification while also supporting `utf-8`.

The server operates on its negotiated encoding, but CLI I/O can be declared independently via `--index-io=utf-8|utf-16|codepoint`, where *codepoint* denotes Unicode scalar values. When the

two indexings differ, `Lanser-CLI` emits both coordinate systems in verbose mode and records the server-side encoding in bundle metadata. This makes explicit, and resolves, the long-standing ambiguity between LSP's default UTF-16 indexing and the UTF-8 conventions common in downstream tools (Microsoft, 2025a).

3.3 Repositioning and Ambiguity Resolution

Even with encodings resolved, selectors must remain meaningful as code evolves. The RELOCATE algorithm (Algorithm 1) resolves a possibly stale selector against the current workspace and reports ambiguity with ranked, deterministic evidence.

Strategy. `Lanser-CLI` first consults an exact `docVersion` map when the referenced snapshot is available. Symbolic and AST-based selectors trigger a reparse of the current workspace and a structural match. Content-anchored selectors invoke a fuzzy search using winnowed k-grams within the recorded context window. The resulting candidates are scored, sorted, and returned with deterministic evidence; mutating commands require explicit confirmation whenever the accepted target is ambiguous.

Scoring. Exact snapshot maps and exact anchor-hash matches are treated as certified candidates and assigned score 1; all other candidates are ranked by a deterministic score. 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 $[0, 1]$, with inapplicable features set to 0 and recorded in the explanation. Non-certified candidates are ranked by the convex combination

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

Candidates are sorted by descending score and then by ascending $(\texttt{uri}, \texttt{range})$, inducing a total order. Let Δ be the margin between the top two scores, with $\Delta = \infty$ when only a single candidate exists. A selector is accepted automatically only when the top score is at least τ and $\Delta \geq \delta$; otherwise the bundle surfaces top-k alternatives with explanations, and mutating commands require explicit target confirmation.

Correctness sketch. Under a frozen snapshot, an exact `docVersion` map returns the corresponding certified range. Symbolic and AST selectors either resolve to a unique structural target or produce a ranked candidate set with explicit disambiguation evidence; uniqueness is not assumed in the presence of overloads, shadowing, or duplicate names. For anchors, if the snippet hash and context match exactly and no conflicting exact matches exist, RELOCATE returns a certified candidate with score 1; otherwise it ranks candidates by Eq. (1). 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 a CLI-first interface aimed at interactive debugging, automated agent loops, and CI. The surface is organized around navigation, mutation, batch execution, and schema validation.

Algorithm 1 `Lanser-CLI` Repositioning (RELOCATE)

Require: Selector s, workspace W, optional snapshot v, thresholds τ, δ
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** $\{(\text{map}(s, v), 1.0, \text{certified})\}$
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**
    if c is certified **then**
        $c.score \leftarrow 1.0$
    else
        $c.score \leftarrow f(s, c)$ ▷ Eq. (1): deterministic weights
    end if
end for
$\mathcal{C} \leftarrow \text{sort}(\mathcal{C}, -\text{score}, \text{uri}, \text{range})$
if $\mathcal{C} = \emptyset$ **then**
    return ERROR(`E/NOT_FOUND`)
end if
if $\mathcal{C}[1].score < \tau$ or $(|\mathcal{C}| > 1$ and $\mathcal{C}[1].score - \mathcal{C}[2].score < \delta)$ **then**
    attach `disambiguation` evidence
end if
return $\mathcal{C}[1..k]$

Navigation. Read-only commands, `lanser def`, `refs`, `hover`, `symbols`, and `diag`, all accept any POSITIONSPEC. A dedicated `lanser locate` command resolves abstract selectors into concrete ranges and can preview the targeted source span, making selector intent auditable before any downstream action is taken.

Safe mutation. Mutating commands default to preview. For instance, `lanser rename` is gated by `prepare-rename`: it emits a workspace edit and unified diff before any write and requires an explicit `--apply` flag to modify files. Application is protected by a workspace jail, allow/deny path filters, staged validation, conflict detection, and a clean-worktree guard that can only be overridden with `--allow-dirty`.

Batch execution and tracing. `lanser batch` consumes JSONL command queues and emits JSONL bundles, supporting high-throughput planners. Any command can additionally emit an execution trace containing orchestrator metadata and JSON-RPC traffic; `lanser trace replay` uses this trace, together with a frozen workspace digest, to regenerate byte-stable outputs for auditing and regression tests.

Schema contracts. `lanser schema` exports and validates JSON Schemas for selectors and bundle outputs. Agents can validate payloads before execution, and CI systems can catch incompatible schema changes before they corrupt historical traces or reward logs.

5 Reinforcement Learning from Compiler and Language Server Feedback

Planner-act loops benefit from verifiable intermediate signals. In RLCSF, the feedback source is not a human preference model but a compiler, type checker, or language server operating on the current workspace. A state contains the task, workspace snapshot, and selector context; an action is a tool query or mutation proposal; an observation is the resulting `Analysis Bundle`.

The reward is designed to be online-computable, replayable under frozen snapshots, and useful as *shaping* rather than as a replacement for terminal task success. It follows the potential-based reward-shaping template of Ng et al. (1999), with the potential grounded in machine-checked program facts.

State features. Let B_t be the canonical bundle after step t, and let σ_t be the diagnostic scope recorded in that bundle (a workspace, file, or resolved range). Let $D_t \in \mathbb{N}$ be the diagnostic count over σ_t. Diagnostic deltas are credited only when $\sigma_t = \sigma_{t-1}$; otherwise the bundle records `scope_changed` and the diagnostic component is set to zero unless the evaluator pins a common scope. Let $S_t \in [0, 1]$ denote safety readiness for a prospective mutation, with read-only steps carrying forward the previous value unless a safety check is observed. Let $A_t \in [0, 1]$ denote the top selector-resolution confidence, and let $E_t \in \{0, 1\}$ indicate a structured tool error such as `E/AMBIGUOUS`, `E/APPLY_CONFLICT`, or `E/INDEXING_MISMATCH`.

Potential and reward. For non-negative weights w_D, w_S, w_A, w_E, define

$$\Phi(B_t) \;=\; -w_D D_t \;+\; w_S S_t \;+\; w_A A_t. \tag{5.1}$$

The RLCSF process reward is

$$r_t^{\text{csf}} \;=\; \gamma\,\Phi(B_t) - \Phi(B_{t-1}) - w_E E_t, \tag{5.2}$$

where $\gamma \in [0, 1]$ is the RL discount used for shaping. In the common undiscounted online-planning case $\gamma = 1$,

$$r_t^{\text{csf}} = w_D(D_{t-1} - D_t) + w_S(S_t - S_{t-1}) + w_A(A_t - A_{t-1}) - w_E E_t.$$

The reward thus credits diagnostic reduction, safety improvement, and selector-confidence improvement while penalizing structured tool failures. When an external task reward is available, an RL learner can optimize $R_t^{\text{task}} + r_t^{\text{csf}}$. With $w_E = 0$ and bundle features included in the Markov state, the shaping term reduces to the standard potential-based form of Ng et al. (1999); with $w_E > 0$, the tool-error penalty is an explicit task-design choice that may shift the optimal policy.

5.1 Deterministic Analysis Bundles

`Analysis Bundles` normalize compiler and language server payloads and pin environment metadata. Lists are deterministically ordered by $(\texttt{uri}, sL, sC, eL, eC)$ with explicit tie-breakers, and each bundle carries a stable `bundleId` computed as a hash over a canonicalized subset of its fields, excluding volatile timestamps and run-local trace data.

Response envelope.


```
{
  "version": "1.2",
  "bundleId": "sha256:...",
  "status": "ok",
  "request": {"cmd": "definition", "selector": {...}},
  "resolution": {"original": "...", "resolved": {...}, "disambiguation": [...]},
  "facts": {"definitions": [...], "hover": {...}, "provenance": "lsp"},
  "edits": {"workspaceEdit": null, "diff": null},
  "processReward": {
    "version": "rl-csf-v1",
    "previousBundleId": "sha256:...",
    "r": 1.924,
    "components": {"diag_delta": 3, "safety_delta": 1,
                   "confidence_delta": 0.24, "tool_error": 0},
    "weights": {"wD":0.5,"wS":0.4,"wA":0.1,"wE":0.5,"gamma":1.0},
    "source": "compiler+lsp",
    "explanation": "Eq. (\\ref{eq:proc-reward}) over adjacent frozen bundles"
  },
  "environment": {"tool": {"name":"pyright","version":"1.1.406"},
  "positionEncoding":"utf-16","python":{"version":"3.12.0"}, ...},
  "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, a tool binary and configuration (V, Π), a negotiated `positionEncoding`, and a request Q. Assume the pinned tool has deterministic semantics under these inputs, up to unordered result sets normalized by `Lanser-CLI`. Then `Lanser-CLI` produces identical hash-domain canonical bundles across runs; in particular, `bundleId`(B) is constant. Fields explicitly excluded from the hash domain, such as timestamps and run-local trace spans, are not covered by this claim.

Proof sketch. The orchestrator enforces deterministic sorting, canonicalizes JSON via the JSON Canonicalization Scheme (JCS) (Rundgren et al., 2020), records environment invariants in the envelope, and excludes non-deterministic fields from the hash domain. Given identical inputs and deterministic tool semantics, the normalized semantic facts are a function of (S, V, Π, Q), so the resulting hash-domain canonical JSON, and hence `bundleId`, is invariant. □

Proposition 5.2 (Non-negativity under componentwise improvement)**.** Consider Eq. (5.2) with $\gamma = 1$ and fixed non-negative weights. If a transition weakly decreases diagnostics over the same scope ($D_t \le D_{t-1}$), weakly improves safety readiness ($S_t \ge S_{t-1}$), weakly improves selector confidence ($A_t \ge A_{t-1}$), and produces no structured tool error ($E_t = 0$), then $r_t^{\text{csf}} \ge 0$.

Proof sketch. For $\gamma = 1$, Eq. (5.2) expands to $w_D(D_{t-1} - D_t) + w_S(S_t - S_{t-1}) + w_A(A_t - A_{t-1}) - w_E E_t$. Each difference term is non-negative by assumption and the error penalty vanishes, so the sum is non-negative. Determinism of the underlying bundles (Theorem 5.1) makes the reward replayable. □

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: stage and apply with jail + filters; **if** conflict **then return** `E/APPLY_CONFLICT`
6: notify server via `didChange`; **return** success bundle with D

5.2 Editing and Guardrails

`Lanser-CLI` applies workspace edits via a staged, fail-closed workflow. Each change is written to a temporary file in the target directory while preserving detected line endings, character encoding, and (where permitted) file mode; the data is then `fsync`ed and the original is replaced via `rename(2)`. This guarantees per-file atomic replacement, but not, by itself, crash-atomicity for multi-file edits.

For multi-file edits, `Lanser-CLI` validates the full edit set before any replacement and records rollback metadata; whole-patch conflict detection and rollback can additionally be delegated to `git apply --3way`. Merge conflicts are surfaced as a structured `E/APPLY_CONFLICT` carrying machine-readable hunks. Additional file-system checks are enforced as well; for example, a case-only rename such as `file.py` to `File.py` on a case-insensitive file system is rejected with `E/FS_PERMISSIONS`.

Threat model and safety envelope. Automated editing exposes several failure modes: selector resolution can target the wrong span, system failures can leave partially applied changes, writes can escape the project root, stale configuration can invalidate analysis, and encoding mismatches can corrupt positions.

`Lanser-CLI` counters them with a layered safety envelope. Operations are preview-by-default (`--dry-run`); a realpath-normalized workspace jail confines all file modifications to the project root, supplemented by explicit allow/deny path filters; and mutating operations require a clean Git working tree unless overridden (`--allow-dirty`). Encoding is detected automatically and, in verbose mode, dual coordinates (`UTF-16` and `UTF-8`) are reported to prevent indexing errors. Ambiguous selectors are surfaced with confidence scores and evidence, and staged application, preflight validation, and Git-backed rollback together reduce the risk of partial failures.

Safety trade-offs are exposed as policy hooks (`--deny-apply-on-ambiguous`, `--workspace-jail`, `--allow-dirty`) so that CI systems and planning agents can configure them explicitly, making automation auditable rather than implicit.

These guardrails complement established program-transformation and differencing tools, e.g., GumTree (Falleri et al., 2014) and RefactoringMiner (Tsantalis et al., 2018), but emphasize determinism, auditability, and CI-grade safety envelopes.

6 Related Work

Language servers, compilers, 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 instantiate `Lanser-CLI` with Pyright for Python (Microsoft, 2025b). Relative to AST differencing and refactoring systems such as GumTree (Falleri et al., 2014) and

RefactoringMiner (Tsantalis et al., 2018), `Lanser-CLI` targets deterministic resolution, replayable artifacts, and reward construction for agent loops.

Anchoring and robust localization. Content-anchored relocation in `Lanser-CLI` builds 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.

Tool-using agents and process supervision. 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), and step-level guidance schemes such as Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023). RLCSF differs by grounding the process signal directly in compiler and language server facts and packaging those facts into deterministic bundles for replay, supervision, and credit assignment.

Compiler feedback as reward. Coding agents already use tests, builds, and type checkers as terminal validators. `Lanser-CLI` makes this feedback finer-grained: diagnostics, selector confidence, prepare-rename checks, and apply conflicts become transition-level signals with deterministic provenance. This enables online search guidance and offline process supervision even when final success labels are sparse.

7 Conclusion

We presented RLCSF and `Lanser-CLI`, a practical substrate for grounding coding agents in compiler and language server feedback. The underlying idea is simple: when machine-checked intermediate facts are available, agents need not learn only from terminal success or failure. Deterministic bundles make those facts replayable, robust selectors preserve intent across edits, guardrails keep mutations auditable, and the RLCSF reward translates diagnostics, disambiguation confidence, and safe-apply checks into transition-level supervision. Together, these components support safer refactors, reproducible CI, and both online planning and offline credit assignment for language agents.

Acknowledgements

We sincerely thank the anonymous reviewers for their helpful feedback. We used LLMs to polish the writing and refine word choice in this work.

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:/...` with an 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 object containing `(request, resolution, facts, edits, environment, capabilities, stableMeta)`, excluding timestamps, trace spans, `processReward`, and other run-local 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.
- Reward reproducibility: `processReward` is computed from the current and previous canonical hash-domain bundle contents plus recorded weights, and is intentionally outside the current bundle's hash domain.

C Exit Codes

D Worked Example

Definition query.

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

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

Returns a `Analysis Bundle` with the resolved range, hover signature, and environment metadata such as `toolVersion=pyright@1.1.406` and `positionEncoding=utf-16`.

Diagnostics and reward state.

```
lanser diag py://pkg.mod#Class.method:body --json
lanser locate py://pkg.mod#Class.method:body --json
```

The resulting bundles record diagnostic counts, selector confidence, structured errors, and provenance needed to compute the RLCSF transition reward.

Rename.

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

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

E RLCSF Reward Signals: Worked Examples

We instantiate Eq. (5.2) with $\gamma = 1$ and $(w_D, w_S, w_A, w_E) = (0.5, 0.4, 0.1, 0.5)$.

Example: 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, safety readiness improves from S_{t-1}=0 to S_t=1, selector confidence improves from A_{t-1}=0.70 to A_t=0.94, and no structured tool error occurs (E_t=0). Then

$$r_t^{\text{csf}} = 0.5 \cdot (5 - 2) + 0.4 \cdot (1 - 0) + 0.1 \cdot (0.94 - 0.70) - 0.5 \cdot 0 = 1.924.$$

The bundle records {`"diag_delta": 3, "safety_delta": 1, "confidence_delta": 0.24, "tool_error": 0`}.

Example: Ambiguous selector and apply conflict. The agent attempts a refactor with unresolved imports. Diagnostics stagnate (D_{t-1}=7, D_t=7), safety readiness does not improve (S_{t-1}=0, S_t=0), selector confidence remains low (A_{t-1}=0.62, A_t=0.62), and the preview reports `E/APPLY_CONFLICT` (E_t=1). Then

$$r_t^{\text{csf}} = 0.5 \cdot 0 + 0.4 \cdot 0 + 0.1 \cdot 0 - 0.5 \cdot 1 = -0.5,$$

discouraging application until ambiguity and conflicts are resolved.

Replayability. Because `processReward` is computed from adjacent deterministic bundle contents and fixed weights, the same r_t^{csf} 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 non-negative under Theorem 5.2 when the stated invariants hold.