# EVIDENCECARD: PLAN EVIDENCE FOR TOKEN-EFFICIENT CODING AGENTS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Tool-using coding agents can consume substantial input tokens while searching repositories, reading source files, and executing tests across long interaction trajectories. Existing token compression methods primarily reduce the size of acquired context or optimize individual context decisions. They do not explicitly track whether the observations collected across a trajectory have satisfied the requirements of the current coding plan. We formulate this limitation as a plan-level evidence management problem and introduce **EvidenceCard**, a lightweight framework that controls context acquisition through plan-conditioned evidence accounting. EvidenceCard exposes acquisition history and heuristic feedback, prompting the coding agent to reassess whether further context is needed. Its prompt architecture combines a stable, cache-friendly evidence-policy prefix with a small dynamic runtime tail. These signals guide the original agent to acquire, reuse, compress, or stop collecting evidence without additional training and auxiliary LLM calls. In contrast to retrospective compression, EvidenceCard acts prospectively by making evidence sufficiency visible before the next tool action. Empirical results show that EvidenceCard substantially reduces file-read tokens and requests, revealing reductions of up to an order-of-magnitude in read tokens relative to baselines, while maintaining robust task-solving performance[1].

## 1 INTRODUCTION

Large language models (LLMs) are increasingly deployed as tool-using coding agents that search repositories, inspect source files, edit code, and execute tests over multi-step interaction trajectories (Yao et al., 2023; Schick et al., 2023). However, these advanced capabilities incur substantial inference and tool-use overhead. The primary resource constraint for such agents is token consumption. In repository-level tasks, a single coding objective can require a significant volume of tokens, as directory listings, search results, source code, execution traces, and test logs accumulate rapidly across multiple turns, making input tokens a dominant component of the total operational cost.

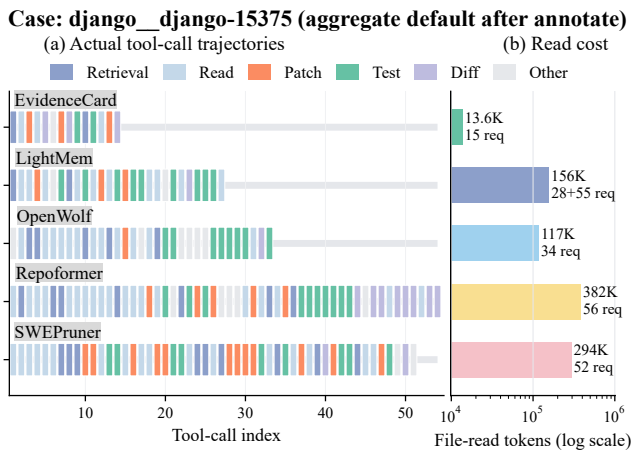



Figure 1: Comparison of (a) actual tool-call trajectories and (b) file-read cost on `SWE-Bench: django_django-15375` (Task example in Figure 2).

Most existing token-efficiency techniques operate on context that has already been retrieved. General *prompt-compression methods* remove natural-language redundancy or learn token-level keep/drop decisions (Li et al., 2023; Jiang et al., 2023; 2024; Pan et al., 2024).

[1]Source codes will be publicly available after acceptance.



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.

Table 1: Comparison of deployment requirements and trajectory-level context control.

| Method | Training free | No extra LLM call | Irrelevant read aware | Repeated read aware | Plan/phase aware |
|---|---|---|---|---|---|
| Prompt Compression | × | ✓ | × | × | × |
| Code Pruning | × | ✓ | × | × | × |
| Agent Memory | ✓ | × | ✓ | × | × |
| File Middleware | ✓ | ✓ | × | ✓ | × |
| Retrieval Gate | × | × | × | × | × |
| **EvidenceCard** | ✓ | ✓ | ✓ | ✓ | ✓ |

Such methods can shorten an assembled prompt, but source code is structurally sensitive: removing tokens may obscure syntax, scope, control flow, or dependencies. *Code-pruning methods* therefore preserve or select implementation-relevant content more carefully. For example, SWE-Pruner (Wang et al., 2026) reduces individual source-code observations by retaining lines judged relevant to the current step. However, they do not govern what should be read during an agent's trajectory. As a result, the redundancy of re-reading the same targets across turns causes these costs to accumulate rapidly. Beyond compressing observations, other approaches intervene at specific steps in the information-acquisition process. *Agent memory methods* govern specific components of the information acquisition process. For instance, LightMem (Fang et al., 2026) manages information after it has been acquired by consolidating long interaction histories into compact memories for subsequent reuse. *File middleware methods*, such as OpenWolf (OpenWolf Contributors, 2026), maintain repository-level state and issue warnings when an agent attempts to read the same file repeatedly. *Retrieval gate methods*, such as Repoformer (Wu et al., 2024), learn whether cross-file retrieval is likely to improve the current prediction and abstain when additional context appears unnecessary. However, these local optimizations do not control the global coding trajectory. Agents may still inspect irrelevant files, continue searching after hypotheses are supported, thereby failing to align their actions with the evolving state of the task. The different settings of token-efficiency methods are compared in Table 1.

We study this failure mode as a problem of *plan-level evidence management*. A coding plan implicitly creates evidence requirements: the agent may need evidence for the failure mechanism, the responsible implementation, the safety of a proposed edit, and the outcome of targeted verification. Yet conventional interaction histories do not explicitly indicate which observations have already supported the requirements, or when the available evidence is sufficient to proceed. As a result, the agent lacks an operational stopping condition. Retrospective compression can reduce the cost of retaining this history, but it cannot recover the tokens, latency, and tool execution already spent acquiring unnecessary observations.

Figure 1 illustrates this distinction on `SWE-Bench:django_django-15375`. All evaluated methods eventually solve the task, but their trajectories differ substantially after the relevant edit region has been identified. LightMem identifies an early patch yet subsequently rereads source code, inspects test infrastructure, and executes broader test suites. Similarly, although OpenWolf eliminates repeated-read tokens, it continues to explore distinct regions of the code repository and test suite. Repoformer retrieves relevant files early, yet its retrieval decision does not govern subsequent inspection and verification stages. While SWE-Pruner reduces the content retained within individual reads, the agent still generates multiple auxiliary test scripts and performs additional tool calls to establish confidence. Our method (EvidenceCard) instead remains focused on the evidence requirements associated with the current hypothesis and patch. In this example, it uses 13.6K file-read tokens and 15 requests, compared with 117K–382K read tokens and 34–83 requests for the comparison methods.

The example highlights a fundamental gap. Coding agents need more than a compressor, a retrieval gate, a memory store, or a code selector. They require a *plan-level mechanism* that actively governs the agent's trajectory by asking: what evidence is required by the current coding plan, which observations have already supplied it, which reads are redundant, and when it should stop exploring? To this end, we introduce **EvidenceCard**, a lightweight framework for plan-level evidence accounting in coding agents. EvidenceCard shifts the unit of trajectory control from raw context text to *evidence transactions*. At the beginning of a task, a stable instruction prefix specifies the evidence-aware operating policy: acquire context to address evidence gaps the agent identifies in its current plan, reuse recorded evidence before rereading a target, preserve resources for implementation and verification, and stop reading when the agent judges the evidence sufficient to patch or verify. Tool observations are then mapped to compact ledger entries. Across turns, the ledger exposes which targets have already been inspected, and how the trajectory's acquisition effort is distributed across phases. EvidenceCard provides this state to the coding agent through a cache-friendly prompt mech-

anism. A compact dynamic tail reports a weak-observation count, repeated-evidence count, phase distribution, conversation shape, and recent ledger entries. The coding LLM uses this feedback and the conversation history to assess which evidence its plan still requires; the runtime does not explicitly enumerate or mark those requirements as satisfied. Together, these signals guide the agent's cross-turn decisions about collecting new evidence, reusing existing evidence, and advancing once it judges the available evidence sufficient. The mechanism requires neither additional model training nor a separate LLM call for memory extraction, retrieval, or consolidation, and it does not modify the underlying coding model.

The conceptual distinction is that EvidenceCard acts *prospectively*. Before the next action, it presents accumulated acquisition state and prompts the coding LLM to reassess whether further evidence is needed for its current plan. This allows context cost to be treated as the consequence of a sequence of evidence-acquisition decisions. Our empirical results show that this trajectory-level feedback substantially reduces file-read tokens and tool requests, revealing reductions of up to an order-of-magnitude in read tokens relative to the evaluated baselines while retaining robust task-solving performance.

The contributions of our work are summarized as follows.

- We formulate token-efficiency for coding agents as a *plan-level evidence management problem*, distinguishing the prospective control of context acquisition from retrospective prompt compression and step-local retrieval decisions.
- We introduce a rule-based evidence ledger that supports the agent's plan-aware decisions by tracking repeated acquisition, recent observations, and phase-level expenditure across exploration, implementation, and verification.
- We develop a training-free and cache-friendly control mechanism that combines a stable evidence-policy prefix with a compact dynamic ledger tail, providing cross-turn acquisition and stopping feedback without modifying the base model or introducing auxiliary LLM calls.

## 2 Related Work

### 2.1 Agents for Coding.

Tool-using language agents combine reasoning with external actions such as search, execution, and environment feedback (Yao et al., 2023; Schick et al., 2023). Reflexion and Self-Refine further improve multi-step behavior through verbal or self-generated feedback (Shinn et al., 2023; Madaan et al., 2023). In software engineering, SWE-agent studies the agent–computer interface (Yang et al., 2024), while AutoCodeRover combines program-aware search with autonomous repair (Zhang et al., 2024). Agentless decomposes issue resolution into localization, repair, and validation (Xia et al., 2024), and OpenHands provides a general platform for developer-like agents (Wang et al., 2025).

### 2.2 Prompt and Context Compression.

General prompt compression methods reduce the size of long inputs before generation. Selective Context (Li et al., 2023) removes low-information lexical units using self-information. LLMLingua (Jiang et al., 2023) introduces a coarse-to-fine compressor with budget control and token-level pruning, while LongLLMLingua (Jiang et al., 2024) adapts compression to long-context scenarios through query-aware compression and document reordering. LLMLingua-2 (Pan et al., 2024) reformulates prompt compression as token classification learned from distilled data, improving latency and generalization. Recent empirical work further shows that context compression can be effective for repository-level code tasks under appropriate compression paradigms (Feng et al., 2026). These methods mainly answer how to shorten an input that has already been assembled.

### 2.3 Code-aware Retrieval and Pruning.

Repository-level tasks require cross-file context, but excessive retrieval adds latency and noise. CrossCodeEval demonstrates the importance of cross-file dependencies (Ding et al., 2023). RepoCoder iterates between retrieval and generation (Zhang et al., 2023), DraCo uses dataflow-guided

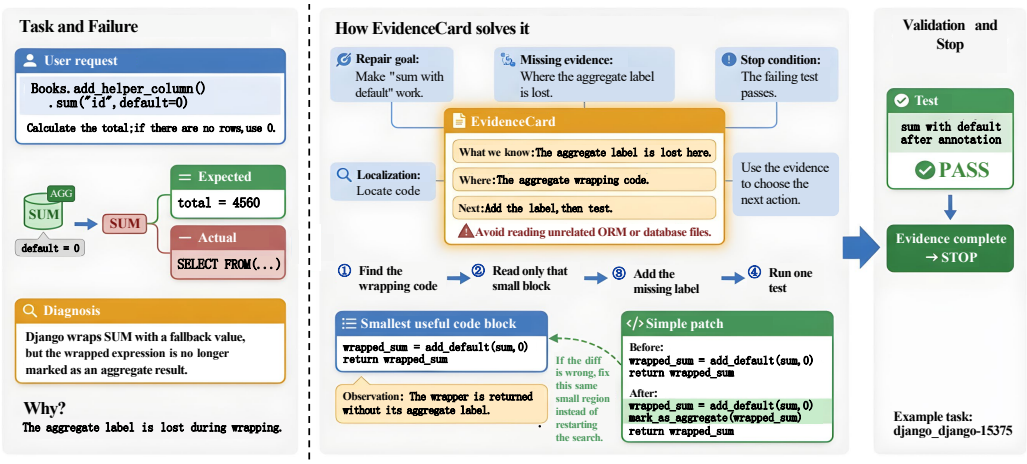



Figure 2: Task-grounded illustration of EvidenceCard. The left panel summarizes the bug: computing an aggregate with a default value after annotation produces invalid SQL because the fallback wrapper loses its aggregate identity. The right panel shows how EvidenceCard converts the task into a compact evidence plan, reads the smallest relevant code block, records the confirmed cause and edit target, applies an evidence-supported patch, and stops once the targeted test passes. The code is presented as simplified pseudocode for clarity.

retrieval (Cheng et al., 2024), and Repoformer selectively abstains when retrieval is unlikely to help (Wu et al., 2024). SWE-Pruner preserves step-relevant source lines (Wang et al., 2026), CodePromptZip applies code-specific prompt compression (He et al., 2026), and Squeez extracts compact evidence from mixed tool observations (Kovacs, 2026). OpenWolf instead maintains repository state and repeated-read warnings (OpenWolf Contributors, 2026). These methods optimize individual retrieval or observation decisions rather than evidence acquisition over the complete trajectory.

### 2.4 Agent Memory, Context, and Budget.

Long-horizon agents must decide what state remains active, since longer context windows do not guarantee reliable use of relevant information (Liu et al., 2024). MemGPT frames context management as virtual memory (Packer et al., 2024), LightMem reduces overhead through memory retrieval and consolidation (Fang et al., 2026), and PAACE engineers agent context with explicit plan awareness (Yuksel, 2025). Token-budget-aware reasoning (Han et al., 2025) and BudgetThinker (Wen et al., 2025) further show that explicit budget signals can shape reasoning length and cost.

## 3 Method

### 3.1 Overview

EvidenceCard does not ask another LLM whether to call a tool, and it does not rely on a separate memory model to summarize past tasks. Instead, it observes the result *after* each tool call and records the transaction in a lightweight evidence ledger. A search result, file read, test log, stack trace, or patch diff is treated as evidence with a phase, evidence type, target, token cost, repeated-read flag, score, decision, and stop rule. This makes EvidenceCard a plan-level mechanism: it aggregates evidence across localization, inspection, patching, and verification, rather than optimizing only the next retrieval or the next file read. The ledger is then replayed as a compact dynamic tail before the next model request. Thus, the agent is repeatedly reminded whether it has accumulated useful evidence, reread the same target, spent too much effort in one phase, or should move from exploration to patching and verification. Plan awareness therefore arises from the agent's interpretation of the stable policy and runtime feedback, without a separate LLM call or an explicit runtime representation of requirement closure. The ledger supplies acquisition statistics and heuristic cues; it does not itself establish the semantic relevance or sufficiency of an observation.

### 3.2 PLAN-EVIDENCE CONTRACT

EvidenceCard begins with a stable plan-evidence contract in the system prompt. The stable system-prompt section supplies an operating policy for the base agent's existing plan: spend context on evidence gaps relevant to localization, patching, or verification; reuse prior evidence before rereading a target; preserve capacity for implementation and verification; expand context gradually when evidence is weak; and stop reading when the agent judges the evidence sufficient to patch or verify. The coding LLM identifies these gaps and assesses progress during its ordinary reasoning. Here, plan conditioning refers to this prompt-guided behavior; the runtime neither maintains a machine-readable requirement list nor automatically verifies that a requirement has been satisfied.

The policy is stable across turns and is attached only once to the system prompt. This separation is important: the plan-level acquisition policy remains cache-friendly, while changing trajectory state is carried by the Card. Appendix A.1 gives the complete experimental text.

### 3.3 EVIDENCE LEDGER

The evidence ledger is the central state maintained by EvidenceCard, and ledger construction is deliberately rule-based. After each tool call, EvidenceCard records one evidence transaction. Deterministic rules derive the phase $p_i$, evidence kind $k_i$, and target $g_i$ from the tool command and visible result. The ledger entry retains these fields together with visible token cost $c_i$, a heuristic evidence score $s_i$, a repeat flag $r_i$, and a decision label $d_i$:

$$e_i = (p_i, k_i, g_i, c_i, s_i, r_i, d_i). \tag{1}$$

Searches and repository listings generally support localization; bounded and full file reads support inspection; diffs support patching; and test outcomes support localization or verification. Repetition is exact target reuse, $r_i = 1$ when the extracted target has appeared earlier. Decision label $d_i$ receives REUSE; otherwise, an observation exceeding its fixed type-specific allowance receives COMPRESS; all other observations receive COLLECT. The decision label COLLECT, REUSE, and COMPRESS make the cost and redundancy of context acquisition visible. These labels are feedback for the next turn and do not block tool execution.

The score $s_i$ is a deterministic content heuristic used only to mark observations that appear weak. The runtime accumulates

$$W_t = \sum_{i=1}^{t} \mathbb{I}[s_i < \theta], \qquad R_t = \sum_{i=1}^{t} r_i, \tag{2}$$

where $W_t$ is the number of low-scoring observations and $R_t$ the number of repeated targets. $W_t$ is a cumulative heuristic observation-quality signal, not a measure of unresolved plan requirements: it does not decrease when the agent resolves an evidence gap. Together with $R_t$ and the phase histogram $N_t$, it summarizes weak observations, repetition, and phase-level activity over the trajectory. The coding LLM interprets these statistics alongside the actual observations when assessing what evidence is still needed.

Appendix A.2 reports the exact parser rules, score increments, allowances, threshold, and repeat matching used in all experiments.

### 3.4 THE EVIDENCE CARD

EvidenceCard separates stable instructions from dynamic state. The stable prefix is appended to the system prompt once and contains the plan-evidence contract. Because this prefix is stable across turns, it is compatible with prompt caching and does not grow with the trajectory.

The ledger becomes useful to the policy through a small dynamic Card appended before the next model request. Let $S$ be the stable evidence policy, $H_t$ the ordinary conversation history, and $\mathcal{L}_t$ the tool-observation ledger. EvidenceCard renders

$$z_{t+1} = \mathrm{Card}(q, \mathrm{Shape}(H_t), W_t, R_t, N_t, \mathrm{Tail}_K(\mathcal{L}_t)), \tag{3}$$

where $N_t$ is the phase histogram. Each recent entry displays its phase, evidence kind, target, visible tokens, score, decision, execution status, and repeat marker, subject to the corresponding ablation

**Algorithm 1** EvidenceCard runtime loop

**Require:** Query $q$, base policy $\pi_\phi$, tool environment $\mathcal{T}$
Initialize ledger $\mathcal{L}_0 \leftarrow \emptyset$ and $W_0, R_0 \leftarrow 0$
Attach the stable plan-conditioned evidence policy $S$
**for** $t = 0, 1, \ldots$ **do**
  $z_{t+1} \leftarrow \mathrm{Card}(q, H_t, W_t, R_t, N_t, \mathrm{Tail}_K(\mathcal{L}_t))$
  $a_{t+1} \sim \pi_\phi(\cdot \mid S \oplus H_t \oplus z_{t+1})$
  **if** $a_{t+1}$ terminates the task **then**
    **return** final answer or verified patch
  **end if**
  $o_{t+1} \leftarrow \mathcal{T}(a_{t+1})$
  Parse $p_{t+1}, k_{t+1}, g_{t+1}$; compute $c_{t+1}, s_{t+1}, r_{t+1}, d_{t+1}$
  Append $e_{t+1}$ to $\mathcal{L}_t$; update $W_{t+1}, R_{t+1}, N_{t+1}$
  $H_{t+1} \leftarrow H_t \oplus (a_{t+1}, o_{t+1})$
**end for**

switches. The next action is generated by the unchanged base policy:

$$a_{t+1} \sim \pi_\phi(\cdot \mid S \oplus H_t \oplus z_{t+1}), \tag{4}$$

where $\oplus$ denotes concatenation. Only the latest $K$ entries are rendered, so the Card does not grow with the full trajectory. In our implementation, tool results entering $H_t$ are separately processed by the host agent's bounded-preview path, which remains constant throughout all experiments; the ledger scores and charges the text actually exposed to the agent. Consequently, EvidenceCard reduces dynamic context growth while preserving a stable prefix that remains compatible with prompt caching. The objective is to reduce total input cost, rather than to maximize the cached-token ratio itself. We document this integration boundary in Appendix A rather than treating preview generation as part of the Card. The Card is advisory: the base agent still chooses tools, edits code, judges whether its current hypothesis is supported, and decides when to advance or finish.

### 3.5 Workflow

The complete runtime loop is summarized in Algorithm 1. It shows the intended causal chain without introducing a learned objective: the stable policy conditions plan execution, deterministic bookkeeping builds the ledger, and the Card returns that trajectory state to the same agent.

## 4 Experiments

### 4.1 Experimental Setup

**Datasets.** We use SWE-Bench (Jimenez et al., 2024), SWE-QA (Peng et al., 2026), and Terminal-Bench2.0 (Merrill et al., 2026) as test datasets. SWE-Bench is a benchmark for evaluating coding agents on real-world software engineering issues. The original SWE-Bench contains 2,294 instances; SWE-Bench-Verified contains 500 human-validated instances intended to filter underspecified issues and unfair tests; SWE-Bench-Lite consists of 300 carefully selected instances from SWE-Bench, ideal for rapid experimentation. We evaluate on the intersection between the Verified and Lite test sets, yielding 93 instances (called SWE-Bench-93). SWE-QA is a repository-level code question-answering benchmark containing 720 human-validated questions from 15 Python repositories. Following SWE-Pruner (Wang et al., 2026), we evaluate 144 questions from `conan`, `reflex`, and `streamlink`, covering cross-file and multi-hop code reasoning.

**Baselines.** We compare EvidenceCard against four methods: LightMem, OpenWolf, Repoformer, and SWE-Pruner. For a controlled comparison, each method is ported into an isolated copy of the same base agent with unrelated efficiency modules disabled. We reuse the official source code, checkpoints, and inference procedure whenever available; when direct integration is infeasible, we follow the published workflow as closely as possible and document the deviation. All methods use the same base models, task prompts, tool interface, benchmark instances, and execution limits, and are evaluated by the same trajectory-level accounting procedure.

Table 2: Comparison on SWE-Bench-93 with different LLMs. The EvidenceCard row additionally reports Δ against Baseline.

| Method | Request ↓ | Input ↓ | Output ↓ | Tool ↓ | Repeated ↓ | Cached ratio ↑ | ACC ↑ |
|---|---|---|---|---|---|---|---|
| **GLM-5** | | | | | | | |
| Baseline | 5.2K | 89.70M | 937.3K | 54.63M | 1.46M | 93.3% | 80.6% |
| OpenWolf | 4.2K | 58.15M | 684.1K | 34.45M | 0.73M | **95.6%** | 79.6% |
| Repoformer | 4.8K | 101.77M | 1387.7K | 46.95M | 4.43M | 90.7% | 79.6% |
| SWE-Pruner | 5.0K | 83.20M | 894.0K | 49.21M | 1.53M | 93.6% | 80.6% |
| LightMem | 8.8K | 68.16M | 3681.1K | 38.43M | 0.83M | 89.6% | 78.5% |
| **EvidenceCard** | **1.7K** Δ - 66.5% | **12.45M** Δ - 86.1% | **485.0K** Δ - 48.3% | **3.44M** Δ - 93.7% | **0.13M** Δ - 91.4% | 80.6% Δ - 12.6% | **81.7%** Δ + 1.1% |
| **Qwen3.7-Plus** | | | | | | | |
| Baseline | 2.7K | 27.99M | 590.0K | 13.49M | 0.23M | 82.8% | **72.0%** |
| OpenWolf | 2.6K | 29.67M | 626.1K | 14.15M | 0.37M | 83.1% | 71.0% |
| Repoformer | 2.9K | 132.99M | 1234.0K | 27.79M | 17.86M | **88.6%** | 71.0% |
| SWE-Pruner | 3.0K | 33.74M | 679.7K | 16.44M | 0.37M | 84.7% | 71.0% |
| LightMem | 4.2K | 11.39M | 5345.2K | 3.30M | 0.09M | 41.0% | 69.9% |
| **EvidenceCard** | **1.4K** Δ - 45.6% | **7.96M** Δ - 71.6% | **465.2K** Δ - 21.2% | **2.46M** Δ - 81.8% | **0.08M** Δ - 65.7% | 60.6% Δ - 22.2% | **72.0%** Δ + 0.0% |
| **DeepSeek-V4-Pro** | | | | | | | |
| Baseline | 3.2K | 59.53M | 833.6K | 32.53M | 0.81M | 93.7% | **83.9%** |
| OpenWolf | 3.9K | 61.21M | 844.4K | 29.23M | 0.56M | **94.5%** | 82.8% |
| Repoformer | 3.9K | 64.59M | 895.2K | 32.25M | 0.81M | 93.2% | 82.8% |
| SWE-Pruner | 3.8K | 68.37M | 830.9K | 34.83M | 0.58M | 93.6% | 83.9% |
| LightMem | 5.5K | 33.35M | 4394.9K | 14.67M | 0.67M | 69.5% | **83.9%** |
| **EvidenceCard** | **2.2K** Δ - 29.7% | **23.68M** Δ - 60.2% | **614.3K** Δ - 26.3% | **9.79M** Δ - 69.9% | **0.42M** Δ - 48.0% | 84.2% Δ - 9.5% | **83.9%** Δ + 0.0% |
| **GPT-5.2** | | | | | | | |
| Baseline | 4.0K | 58.01M | 656.3K | 37.57M | 1.00M | 94.6% | 81.7% |
| OpenWolf | 4.2K | 64.73M | 709.1K | 40.01M | 1.03M | **94.9%** | 81.7% |
| Repoformer | 3.5K | 68.63M | 844.6K | 25.01M | 1.20M | 92.3% | 80.6% |
| SWE-Pruner | 3.8K | 54.45M | 636.3K | 33.53M | 1.03M | 94.7% | 80.6% |
| LightMem | 6.3K | 42.87M | 9655.1K | 24.53M | 0.52M | 89.5% | 80.6% |
| **EvidenceCard** | **2.7K** Δ - 32.9% | **22.46M** Δ - 61.3% | **468.6K** Δ - 28.6% | **10.02M** Δ - 73.3% | **0.31M** Δ - 68.7% | 86.3% Δ - 8.3% | **82.8%** Δ + 1.1% |

**Metrics.** We report six episode-level efficiency metrics aggregated over all benchmark tasks. *Request* denotes the number of LLM inference requests. *Input* and *Output* denote the total provider-reported input and output tokens across all requests, respectively. *Tool* measures the cumulative token contribution of tool observations visible in the model context, including file contents, search results, command outputs, test logs, and error messages. *Repeated* measures tool-result tokens attributable to repeated read or retrieval operations, identified using normalized tool-command signatures. *Cached ratio* is the fraction of total input tokens served from the provider's prompt cache. All efficiency metrics except Cached ratio are lower-is-better.

**Implementation details.** EvidenceCard is enabled through a single task-level switch; its four prompt-facing components are enabled by default. The stable prefix states the plan-evidence contract, while the dynamic tail exposes recent ledger entries, repeat/reuse signals, and weak-observation signals. The heuristic weak-observation threshold is $\theta = 0.45$; it does not determine semantic evidence sufficiency. The runtime replays $K = 20$ recent entries by default.

**Runtime protocol.** We evaluate the API model identifiers GLM-5, Qwen3.7-Plus, DeepSeek-V4-Pro, and GPT-5.2, using high reasoning effort for all four models. We do not override the sampling temperature and therefore use each provider's default. Agents run headlessly with the same unrestricted tool interface. We record one completed trajectory for each method–task pair and rerun only interrupted, missing, or infrastructure-failed trials. Because provider-default sampling is used, the reported ACC values are descriptive single-run outcomes; we therefore make no statistical claims about accuracy improvements.

Table 3: Cross-benchmark comparison with and without EvidenceCard on GLM-5.

| Method | Request ↓ | Input ↓ | Output ↓ | Tool ↓ | Repeated ↓ | Cached ratio ↑ | ACC ↑ |
|---|---|---|---|---|---|---|---|
| **SWE-Bench-Lite** | | | | | | | |
| w/o EvidenceCard | 17.7K | 363.97M | 3377.6K | 225.06M | 6.24M | **95.2%** | 74.0% |
| w EvidenceCard | **7.1K** | **79.94M** | **2271.2K** | **24.97M** | **0.72M** | 87.0% | **76.0%** |
| | Δ - 60.0% | Δ - 78.0% | Δ - 32.8% | Δ - 88.9% | Δ - 88.4% | Δ - 8.2% | Δ + 2.0% |
| **SWE-QA** | | | | | | | |
| w/o EvidenceCard | 3.5K | 35.90M | 798.6K | 30.18M | 0.08M | **91.5%** | 68.1% |
| w EvidenceCard | **1.7K** | **8.84M** | **542.4K** | **3.67M** | **0.07M** | 81.8% | **72.2%** |
| | Δ - 51.4% | Δ - 75.4% | Δ - 32.1% | Δ - 87.8% | Δ - 12.5% | Δ - 9.7% | Δ + 4.1% |
| **Terminal-Bench 2.0** | | | | | | | |
| w/o EvidenceCard | 3.3K | 109.6M | 1.59M | 58.74M | 1.15M | **90.4%** | 44.9% |
| w EvidenceCard | **1.4K** | **30.5M** | **1.53M** | **15.40M** | **0.30M** | 76.9% | **47.2%** |
| | Δ - 57.6% | Δ - 72.2% | Δ - 3.8% | Δ - 73.8% | Δ - 73.9% | Δ - 13.5% | Δ + 2.3% |

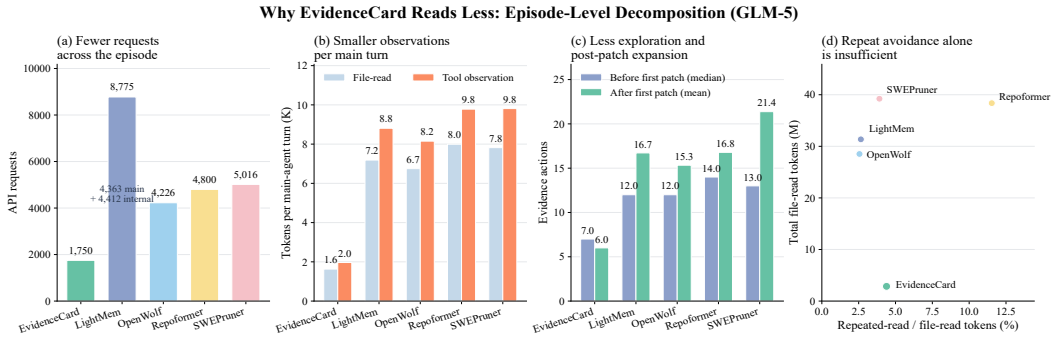



Figure 3: Episode-level efficiency decomposition on GLM-5. EvidenceCard uses fewer API requests, smaller file-read and tool-observation payloads per main-agent turn, less exploration before the first patch, and fewer evidence-acquisition actions after patching. The repeated-read scatter further shows that avoiding rereads alone is insufficient to minimize total context cost.

### 4.2 Numerical Analysis

Table 2 shows that EvidenceCard achieves the lowest request, input-token, tool-observation, and repeated-read costs across GLM-5, Qwen3.7-Plus, DeepSeek-V4-Pro, and GPT-5.2 while maintaining the highest or tied highest task accuracy. Relative to the unmodified agent, it reduces requests by 29.7%–66.5%, input tokens by 60.2%–86.1%, and tool-observation tokens by 69.9%–93.7%. It also consistently outperforms OpenWolf, Repoformer, and SWE-Pruner because their local retrieval or pruning decisions do not control evidence acquisition over the complete trajectory. LightMem compresses history but incurs extra API calls and output tokens for summary generation and merging, whereas EvidenceCard maintains a lightweight evidence ledger without additional LLM calls. These results demonstrate that EvidenceCard reduces the amount of context acquired and propagated throughout the episode, rather than merely compressing or caching an otherwise long trajectory.

### 4.3 Cross-benchmark analysis.

Table 3 further evaluates the method beyond patch generation (using GLM-5). Across the three benchmarks, EvidenceCard consistently reduces trajectory-level resource consumption while maintaining comparable observed task success. On SWE-Bench-Lite, it reduces requests by 60.0%, input tokens by 78.0%, tool-observation tokens by 88.9%, and repeated-read tokens by 88.4%. On SWE-QA, EvidenceCard reduces requests by 51.4%, input tokens by 75.4%, and tool-observation tokens by 87.8%. On the longer-horizon Terminal-Bench 2.0 tasks, it reduces requests by 57.6%, input tokens by 72.2%, tool-observation tokens by 73.8%, and repeated-read tokens by 73.9%. The output-token reduction is substantially smaller on Terminal-Bench 2.0 (3.8%), indicating that EvidenceCard primarily saves the context consumed while acquiring and carrying evidence rather than shortening the final reasoning or answer. Accuracy is treated as a task-success guardrail rather than an optimization target.

Table 4: Ablation study. "w/o" denotes "without."

| Variant | Request | Input | Output | Tool | Repeated | ACC ↑ |
|---|---|---|---|---|---|---|
| w/o stable prefix | 2.9K | 31.57M | 749.0K | 12.65M | 0.82M | 80.6% |
| w/o dynamic ledger tail | 4.4K | 51.40M | 742.5K | 25.93M | 0.87M | 80.6% |
| w/o repeat/reuse signal | 2.0K | 16.44M | 562.4K | 4.83M | 0.14M | **81.7%** |
| w/o low-scoring observations $W_t$ | 1.9K | 15.16M | 549.8K | 4.50M | 0.13M | **81.7%** |
| Full EvidenceCard | **1.7K** | **12.45M** | **485.0K** | **3.44M** | **0.13M** | **81.7%** |

Table 5: Hyperparameter analysis.

| Group | Setting | Request | Input | Output | Tool | Repeated | ACC ↑ |
|---|---|---|---|---|---|---|---|
| Ledger entries | $K = 10$ | 1.9K | 13.69M | 542.4K | 3.77M | **0.09M** | **81.7%** |
| Ledger entries | $K = 20$ | **1.7K** | **12.45M** | **485.0K** | **3.44M** | 0.13M | **81.7%** |
| Ledger entries | $K = 40$ | 1.9K | 14.87M | 546.0K | 4.01M | 0.13M | **81.7%** |
| Evidence threshold | $\theta = 0.30$ | 1.9K | 13.73M | 549.5K | 3.98M | 0.15M | **81.7%** |
| Evidence threshold | $\theta = 0.45$ | **1.7K** | **12.45M** | **485.0K** | **3.44M** | **0.13M** | **81.7%** |
| Evidence threshold | $\theta = 0.60$ | 1.9K | 15.97M | 572.0K | 4.24M | 0.14M | **81.7%** |

### 4.4 Episode-Level Mechanism Analysis

Figure 3 explains why EvidenceCard is more efficient at the trajectory level. Compared with the baselines, it uses fewer requests (1,750 versus at least 4,226), admits substantially fewer file-read and tool-observation tokens per main-agent turn, and requires fewer evidence actions both before and after the first patch. In particular, EvidenceCard receives only 1.630K file-read tokens per main-agent turn, compared with 6.748K–7.993K for the four alternatives, and limits the total file-read volume to 2.853M tokens. Although its repeated-read rate (4.391%) is not the lowest among all methods, its total reading cost is approximately one order of magnitude lower than the baselines.

### 4.5 Ablation Study

Table 4 isolates the contribution of the four prompt-facing components. Removing the stable prefix increases requests from 1.7K to 2.9K and tool from 3.44M to 12.65M, showing that the plan-evidence contract provides an important episode-level operating policy. Removing the dynamic ledger tail causes the largest degradation. Thus, recording evidence in the background is insufficient unless the accumulated state is replayed to inform the next action. The two targeted signals provide complementary gains. The full method performs best on every reported efficiency metric, indicating that the gains do not come at the cost of task success.

### 4.6 Hyperparameter Sensitivity

Table 5 examines the ledger-tail length $K$ and weak-observation threshold $\theta$. The moderate settings $K = 20$ and $\theta = 0.45$ achieve the best overall efficiency, whereas longer ledger replay or a stricter threshold increases context acquisition.

## 5 Conclusion

We present EvidenceCard, a framework that guides coding agents to manage context acquisition according to their evolving plans. A stable instruction prefix prompts the coding LLM to assess evidence needs and sufficiency, while a compact rule-based ledger supplies recent observations, repetition, phase distribution, and heuristic observation-quality signals without extra LLM calls or fine-tuning. The coding LLM remains responsible for interpreting these signals and deciding whether to acquire more context, reuse prior observations, or proceed to implementation and verification. Experiments show substantial reductions in read tokens and tool requests while maintaining task success. These results support prompt-guided, trajectory-level evidence management as a practical approach to improving coding-agent efficiency.

## DISCLOSURE OF AI USE

We used LLMs as the underlying models of the agents evaluated in this work to execute benchmark tasks, as described in the experimental setup. We also used generative AI tools to polish the manuscript for language and readability and to obtain feedback on the presentation of the methodology, experimental design, and interpretation of results. The authors are responsible for verifying AI-assisted content and for the final text, claims, experimental results, and artifacts reported in this paper.

## REPRODUCIBILITY STATEMENT

To support reproducibility, we describe the EvidenceCard runtime loop in Algorithm 1 and provide the stable policy prompt, heuristic ledger rules, and dynamic Card template in Appendix A. Appendix B documents the benchmark sources and task subsets, the base agent and execution environment, and the single-run evaluation and aggregation protocol. Appendix C describes the paired per-task cost analysis and task-resampling procedure. We will publicly release the source code for EvidenceCard upon acceptance of the paper.

## REFERENCES

Wei Cheng, Yuhan Wu, and Wei Hu. Dataflow-guided retrieval augmentation for repository-level code completion. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024. doi: 10.18653/v1/2024.acl-long.431. URL https://doi.org/10.18653/v1/2024.acl-long.431.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-2023. URL https://doi.org/10.52202/075280-2023.

Jizhan Fang, Xinle Deng, Haoming Xu, Ziyan Jiang, Yuqi Tang, Ziwen Xu, Shumin Deng, Yunzhi Yao, Mengru Wang, Shuofei Qiao, Huajun Chen, and Ningyu Zhang. LightMem: Lightweight and efficient memory-augmented generation, 2026. URL https://arxiv.org/abs/2510.18866.

Jia Feng, Zhanyue Qin, Cuiyun Gao, Ruiqi Wang, Chaozheng Wang, Yingwei Ma, and Xiaoyuan Xie. On the effectiveness of context compression for repository-level tasks: An empirical investigation, 2026. URL https://arxiv.org/abs/2604.13725.

Tingxu Han, Zhenting Wang, Chunrong Fang, Shiyu Zhao, Shiqing Ma, and Zhenyu Chen. Token-budget-aware LLM reasoning, 2025. URL https://arxiv.org/abs/2412.18547.

Pengfei He, Shaowei Wang, and Tse-Hsun Chen. CODEPROMPTZIP: Code-specific prompt compression for retrieval-augmented generation in coding tasks with LMs, 2026. URL https://arxiv.org/abs/2502.14925.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMLingua: Compressing prompts for accelerated inference of large language models, 2023. URL https://arxiv.org/abs/2310.05736.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression, 2024. URL https://arxiv.org/abs/2310.06839.

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?, 2024. URL https://arxiv.org/abs/2310.06770.

Adam Kovacs. Squeez: Task-conditioned tool-output pruning for coding agents, 2026. URL https://arxiv.org/abs/2604.04979.

Yucheng Li, Bo Dong, Chenghua Lin, and Frank Guerin. Compressing context to enhance inference efficiency of large language models, 2023. URL https://arxiv.org/abs/2310.06201.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 2024. doi: 10.1162/tacl_a_00638. URL https://doi.org/10.1162/tacl_a_00638.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-2019. URL https://doi.org/10.52202/075280-2019.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

OpenWolf Contributors. OpenWolf: Open-source middleware for Claude Code. https://github.com/openwolfai/openwolf, 2026. Accessed: 2026-07-02.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems, 2024. URL https://arxiv.org/abs/2310.08560.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Ruhle, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Dongmei Zhang. LLMLingua-2: Data distillation for efficient and faithful task-agnostic prompt compression, 2024. URL https://arxiv.org/abs/2403.12968.

Weihan Peng, Yuling Shi, Yuhang Wang, Xinyun Zhang, Beijun Shen, and Xiaodong Gu. SWE-QA: Can language models answer repository-level code questions?, 2026. URL https://arxiv.org/abs/2509.14635.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-0377. URL https://doi.org/10.52202/075280-0377.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, et al. OpenHands: An open platform for AI software developers as generalist agents, 2025. URL https://arxiv.org/abs/2407.16741.

Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. SWE-Pruner: Self-adaptive context pruning for coding agents, 2026. URL https://arxiv.org/abs/2601.16746.

Hao Wen, Xinrui Wu, Yi Sun, Feifei Zhang, Liye Chen, Jie Wang, Yunxin Liu, Yunhao Liu, Ya-Qin Zhang, and Yuanchun Li. BudgetThinker: Empowering budget-aware LLM reasoning with control tokens, 2025. URL https://arxiv.org/abs/2508.17196.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion, 2024. URL https://arxiv.org/abs/2403.10059.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents, 2024. URL https://arxiv.org/abs/2407.01489.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Kamer Ali Yuksel. PAACE: A plan-aware automated agent context engineering framework, 2025. URL https://arxiv.org/abs/2512.16970.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. doi: 10.18653/v1/2023.emnlp-main.151. URL https://doi.org/10.18653/v1/2023.emnlp-main.151.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement, 2024. URL https://arxiv.org/abs/2404.05427.

# A EvidenceCard Implementation

## A.1 Complete Stable Contract

When the stable-prefix switch is enabled, the following text is appended verbatim to the base system prompt:

```
# EvidenceCard
- Method: Plan-Evidence Token Budgeting for Coding Agents.
- Maintain a plan-evidence ledger. Reuse prior evidence before
  rereading the same file, symbol, test, or log.
- Keep patch and verification reserves; do not exhaust the
  budget on broad exploration or verbose explanations.
- If evidence is weak, escalate context one level at a time;
  if evidence is sufficient, stop reading and patch/verify.
```

The line breaks above wrap long source lines for typesetting; the wording is unchanged.

## A.2 Parser and Ledger Rules

Rules are applied in the order shown in Table 6; the first matching evidence-kind rule wins. Matching is case-insensitive. A test command is labeled `error_trace` when its output contains an error indicator and `failing_test` otherwise. For phase assignment, a passing test without a failure/error indicator is `verify`; other test or error evidence is `localize`.

The exact command families are: `git diff`, `apply_patch`, `patch -p`, and `diff -u` for edits; Pytest, NPM/PNPM/Yarn tests, Cargo, Go, Maven, Gradle, and `test.sh` for tests; `rg --files`, `git ls-files`, `Get-ChildItem`, `ls`, and `find` for repository maps; `rg`, `grep`, `Select-String`, and `ripgrep` for searches; and `Get-Content`, `cat`, `sed`, `nl`, `type`, `head`, and `tail` for reads. The error indicators are `traceback`, `assertionerror`, `error`, `failed`, `failing`, and `exception`. Passing verification requires `pass`, `passed`, or `success` and no `fail`, `failed`, or `error` indicator. Bounded reads are recognized by head/tail commands, PowerShell first/last/total-count options, or `sed -n`; other read commands are full-file reads.

The target extractor first searches the command for a path associated with file-read, search, or find syntax. For a repository map it next uses the working directory; otherwise it uses the normalized command, working directory, or `unknown`. Targets are truncated to 180 characters. Repeat detection is exact string matching against prior extracted targets; it performs no semantic file, symbol, range, or content equivalence.

Table 6: Deterministic evidence-kind and phase rules.

| Observed command/result | Evidence kind | Phase |
|---|---|---|
| diff or patch command | `edit_region` | patch |
| test command | `failing_test` or `error_trace` | verify if passing; otherwise localize |
| error indicator in other output | `error_trace` | localize |
| file enumeration/listing | `repo_map` | localize |
| text search | `symbol_signature`; `caller_callee` if the command names callers, callees, references, or usages | localize or inspect |
| configuration-file command | `config_block` | inspect |
| bounded file read | `function_body` | inspect |
| unbounded file read | `full_file` | inspect |
| anything else | `tool_observation` | other |

For an observation $i$, the score begins at 0.40 for successful execution and 0.22 otherwise. It adds 0.22 for a symbol signature, function body, caller/callee relation, configuration block, or edit region; 0.25 for test/error evidence; and 0.08 for patch or verification phase. It adds a further 0.15 when the visible result matches any of `pass`, `fail`, `error`, `traceback`, `assert`, `expected`, `actual`, `diff`, `modified`, `created`, `function`, `class`, `export`, or `def` followed by whitespace. Empty output subtracts 0.18. The result is rounded to two decimals and clipped to $[0, 1]$. We mark it weak when $s_i < \theta$, with $\theta = 0.45$ in the main experiments.

The decision rule is deterministic: an already seen target is REUSE; otherwise an observation whose visible token cost exceeds its planned allowance is COMPRESS; all remaining observations are COLLECT. The ledger also stores a phase-specific advisory stop string, tool name, execution status, argument summary, remaining global budget, and ordinal budget pressure. For a global budget $B$, pressure is *over* below zero remaining tokens, *high* below $0.15B$, *medium* below $0.35B$, and *low* otherwise. Pressure and stop strings are stored for diagnostics but are not rendered in the current Card.

### A.3 DYNAMIC CARD TEMPLATE

Before each model request, the implementation appends the following runtime structure:

```
[Internal runtime note from EvidenceCard. This is not a new user request.]
# EvidenceCard Runtime
- enabled: true
- current_query: <latest user query, at most 240 characters>
- context_shape: messages(user=<u>, assistant=<a>, compact_boundaries=<c>)
- evidence_debt_count: <weak-evidence count>
- repeated_evidence_count: <repeat count>
- phase_counts: <nonzero phase=count pairs>

## Recent Evidence Ledger
- <id>: tool phase=<phase> evidence=<kind> target=<target>
  tokens=<visible tokens> score=<score> decision=<decision>
  status=<status> [repeated]
```

The serialized key `evidence_debt_count` is a legacy implementation name; throughout the paper we interpret it only as the weak-evidence count in Eq. 2. Disabling the weak-evidence signal removes both this count and per-entry scores. Disabling repeat/reuse removes the repeat count, repeat markers, and reuse decisions. Disabling the dynamic Card returns an empty string while ledger recording continues.

# B Detailed Experimental Settings

## B.1 Benchmark Sources and Task Subsets

We use three benchmark families, represented by four local task collections. Each collection is stored under its benchmark directory as `terminal-bench/`, with a task instruction, environment definition, verifier, and `task.toml` configuration for each instance. The runner selects these local tasks rather than downloading a new dataset at evaluation time. Table 4 gives the correspondence between the paper's benchmark names and the local directories.

**SWE-Bench subsets.** SWE-Bench-Lite uses the 300 instances in the `test` split of `princeton-nlp/SWE-bench_Lite`. SWE-Bench-93 contains the 93 instance identifiers shared by Lite and the `test` split of `princeton-nlp/SWE-bench_Verified`; its task records are taken from Verified. This is a deterministic intersection, not a random sample of Verified. Consequently, the Lite experiment extends coverage within the same benchmark family and is not independent of the 93-instance experiment.

Figure 4: Task collections.

| Benchmark | Tasks |
|---|---|
| SWE-Bench-93 | 93 |
| SWE-Bench-Lite | 300 |
| SWE-QA | 144 |
| Terminal-Bench 2.0 | 89 |

**SWE-QA.** The local `SWE` directory contains repository question-answering tasks rather than SWE-Bench patch-generation tasks. We use 48 questions from each of `conan`, `reflex`, and `streamlink`, for 144 questions in total. These records are imported from the SWE-Pruner `downstream_eval/multi_turn/sweqa` collection and converted into local Harbor tasks. The reported SWE-QA results use the official evaluator and its success criterion.

**Terminal-Bench 2.0.** We use the 89 tasks in the collection. Each task supplies its own instruction, container environment, and task-specific verification tests. The agent operates in that environment, and the verifier evaluates the resulting task artifacts.

## B.2 Base Agent and Execution Environment

The base agent is a minimal headless Agent CLI implementation. It executes multi-turn model interactions and tool calls without a graphical interface. Method variants use the shared agent setup described in the main text. The benchmark runner runs the agent inside a task-specific Linux Docker container.

The Windows host builds the Linux executables. The runner mounts the agent executable and configuration into the container. Host-side Python commands run using Python 3.11.15; source files and generated textual outputs use UTF-8. Repository dependencies and test runtimes inside containers are supplied by the individual task environments.

## B.3 Single-Run Evaluation and Aggregation

Each model–method–task combination contributes one agent episode and one final binary evaluation result. An episode may include multiple tool calls, edits, and agent-initiated tests, but a failed final evaluation does not trigger a second episode or a further repair attempt. We do not select the best result from multiple attempts. ACC is the percentage of evaluated tasks whose recorded verifier outcome is success; all other outcomes count as failures. We report success and failure only, without separate accuracy categories for timeouts, missing patches, or evaluation-environment errors.

The task wrappers write their binary outcomes, and the runner aggregates the resulting trial records. Episode-level request and token measurements are aggregated across both successful and unsuccessful tasks, rather than only across solved tasks. Thus, the efficiency totals describe the cost of attempting the evaluated collection. As each combination is run once with provider-default sampling, equal observed ACC values do not establish statistical equivalence, and observed differences are not interpreted as demonstrated accuracy improvements.

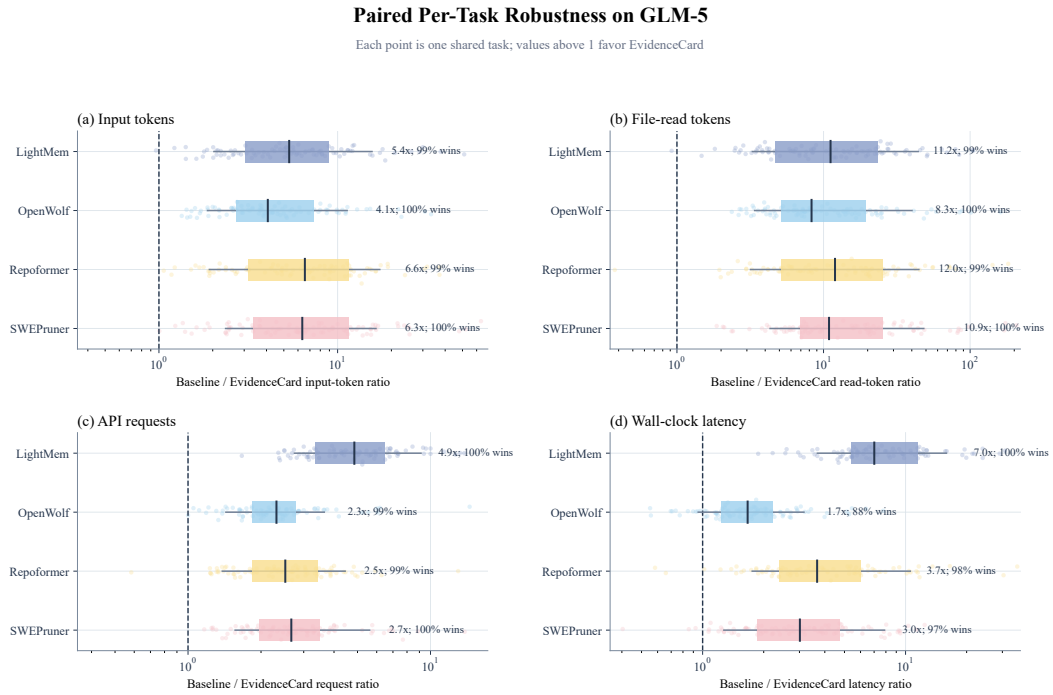



Figure 5: Paired per-task efficiency on GLM-5. Each point compares a baseline with EvidenceCard on the same SWE-Bench-93 instance; values above one favor EvidenceCard. Boxes show the interquartile range, center lines show the median, and horizontal segments span the 10th–90th percentiles. Labels report the median ratio and the fraction of tasks on which EvidenceCard is lower.

## C Task-Level Robustness and Cost Diagnostics

The aggregate results in Table 2 show the total cost of attempting SWE-Bench-93. We additionally analyze the 93 matched tasks individually to determine whether the aggregate advantage is broadly distributed or driven by a small number of unusually long baseline trajectories. For every metric below, a paired ratio is the baseline value divided by the EvidenceCard value on the same task, so a ratio above one favors EvidenceCard. The task-resampling intervals use 5,000 bootstrap samples and characterize sensitivity to the composition of this task set; because every method–task pair is run once, they do not estimate model-sampling variance across repeated episodes.

### C.1 Paired Per-Task Coverage

Figure 5 shows that the aggregate result is not explained by a few outliers. EvidenceCard uses fewer input tokens on 92 of 93 tasks against LightMem and Repoformer and on all 93 tasks against OpenWolf and SWE-Pruner. File-read tokens follow the same coverage pattern. It also uses fewer requests on all tasks against LightMem and SWE-Pruner and on 92 of 93 tasks against OpenWolf and Repoformer. Across the four comparisons, the median baseline/EvidenceCard ratios range from 4.07–6.56 for input tokens, 8.28–11.98 for file-read tokens, and 2.32–4.85 for API requests.

The geometric-mean task ratios lead to the same conclusion: the lower endpoint of every task-resampling interval exceeds 3.9 for input tokens and 8.8 for file-read tokens. Wall-clock latency is also lower on 88.2%–100% of tasks, with median ratios of 1.67–7.01, but we treat latency as descriptive because it can additionally reflect provider queueing and execution-environment variation. The paired token and request comparisons are the stronger evidence that EvidenceCard's efficiency is broadly distributed across the benchmark.

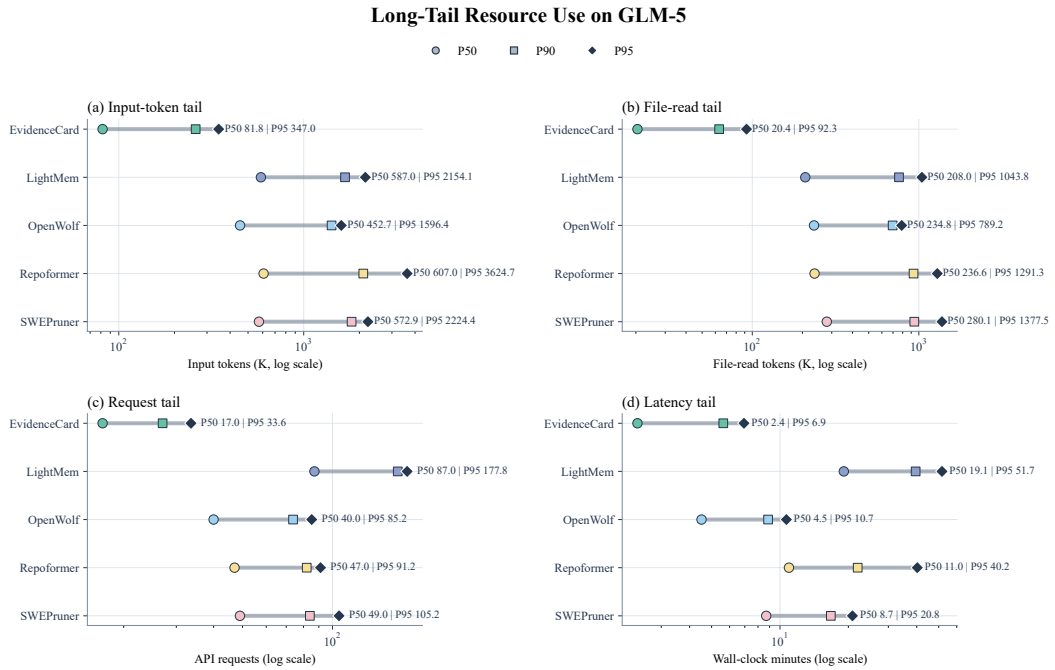



Figure 6: Long-tail resource use across the 93 GLM-5 tasks. Circles, squares, and diamonds denote the 50th, 90th, and 95th percentiles, respectively. All horizontal axes use logarithmic scales.

### C.2 Long-Tail Resource Use

Figure 6 complements the paired win rates by quantifying the expensive tail. EvidenceCard's 95th-percentile input cost is 347.0K tokens, compared with 1,596.4K for the lowest baseline at that percentile. Its 95th-percentile file-read cost is 92.3K tokens, compared with 789.2K for the lowest baseline, and its 95th-percentile request count is 33.6, compared with 85.2. The corresponding wall-clock values are 6.9 and 10.7 minutes. Thus, the aggregate savings do not trade a low median for occasional very expensive episodes: EvidenceCard also substantially compresses the high-cost tail in acquisition volume and episode length.

### C.3 Where the Read Savings Arise

Figure 7 identifies three complementary sources of the reduction. First, most savings come from avoiding unnecessary *first-time* reading rather than only suppressing rereads. Relative to the four baselines, 88%–98% of the read-token difference is attributable to first-time reads. This distinction is important because OpenWolf has a lower repeated-read share than EvidenceCard (2.6% versus 4.4%) but still reads 28.52M tokens in total, compared with 2.85M for EvidenceCard. A repeat warning alone therefore does not control how much new code the agent opens.

Second, EvidenceCard reaches the first patch after a median of seven evidence actions, compared with 12–14 for the baselines, and performs a mean of 6.0 evidence actions after the first patch, compared with 15.3–21.4. This phase balance is consistent with narrower localization and a shorter verification tail. Third, per-task file-read cost is strongly associated with request and read-call counts (Spearman $\rho$ is typically 0.77–0.90), whereas repeated-read tokens have only weak associations for EvidenceCard, LightMem, and OpenWolf ($\rho = 0.17$–$0.19$). Together, these results support the episode-level interpretation: EvidenceCard reduces total acquisition by constraining new exploration and post-patch expansion in addition to exposing repeated targets.

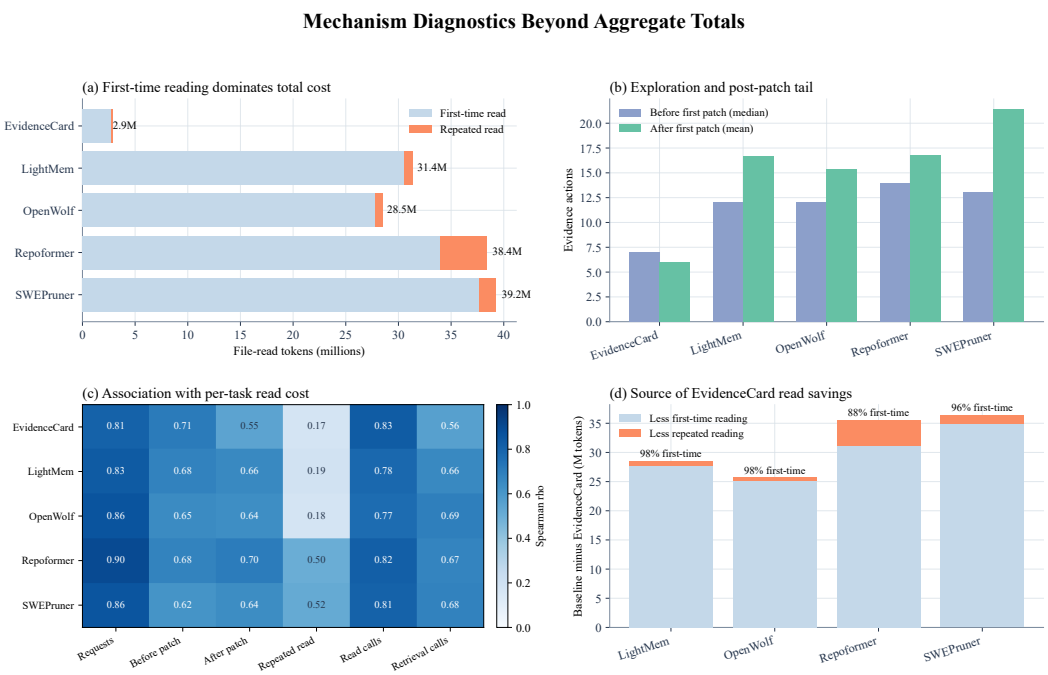



Figure 7: Mechanism diagnostics beyond aggregate totals. (a) File-read tokens are decomposed into first-time and repeated reads. (b) Evidence actions are separated into exploration before the first patch and acquisition after it. (c) Cells report Spearman associations between per-task file-read cost and trajectory properties. (d) Baseline-minus-EvidenceCard read savings are decomposed by their source. The correlations are descriptive and are not interpreted as causal effects.

# D GLM-5 Trajectory Case Studies

This appendix grounds the aggregate efficiency results in the actual tool-call trajectories of representative SWE-Bench-93 tasks. In each case figure, panel (a) shows the chronological sequence of repository retrieval, code reading, patching, testing, diff inspection, and other operations; panel (b) reports the corresponding file-read tokens. The trajectory strip counts parsed tool calls, whereas *Request* counts model API requests. These quantities differ when a model turn issues multiple tools and, for LightMem, when memory-maintenance calls do not invoke repository tools.

## D.1 Early Local Fix versus Post-Patch Drift

The issue in django_django-13033 points to ordering a self-referential foreign-key `_id` path. The decisive change is local: when comparing the relation attribute name, the compiler must use the final path component rather than the complete name. EvidenceCard searches the named condition, reads a narrow surrounding region, and patches on its third parsed tool call. It completes the episode in 13 requests and reads 9.28K file tokens. SWE-Pruner also remains relatively short at 17 requests, but reads 21.89K tokens.

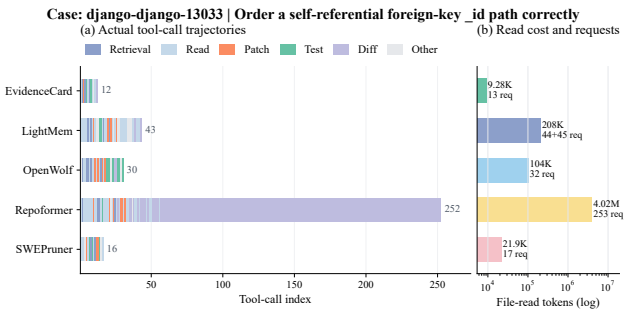



Figure 8: Comparison of (a) actual tool-call trajectories and (b) file-read cost on `SWE-Bench: django_django-13033` under GLM-5.

Repoformer reaches the relevant patch early as well—its first patch is the tenth parsed call—but then continues to 252 parsed tool calls and 253 requests. Its trajectory accumulates 4.02M file-read tokens, including 2.73M attributed to repeated reads. The remaining calls include repeated inspection

of the edit region, tests, diff/status checks, and temporary patch handling. This example illustrates the distinction between a local retrieval decision and trajectory-level feedback. EvidenceCard replays recent acquisition records and phase statistics, prompting the coding LLM to reassess whether its current plan warrants further inspection or verification. The shorter trajectory is consistent with this intended behavior; the ledger itself does not certify that an evidence gap has been closed.

### D.2 Excessive First-Time Reading without Heavy Repetition

In `psf_requests-2317`, the report directly identifies the problematic conversion of a byte-valued HTTP method and suggests the native-string conversion path. EvidenceCard follows this symbol-level clue, checks the relevant definition and import path, performs the one-line replacement, and verifies it with a compact reproduction. It uses 12 requests and 7.42K file-read tokens. The closest baseline in read cost is SWE-Pruner at 13.86K, while Repoformer reads 36.59K.

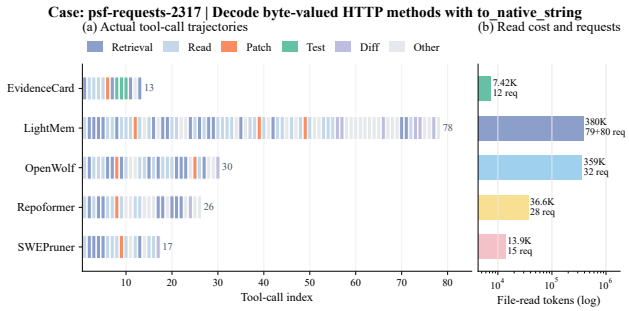



Figure 9: Comparison of (a) actual tool-call trajectories and (b) file-read cost on `SWE-Bench: psf_requests-2317` under GLM-5.

OpenWolf uses only 32 requests but reads 358.96K file tokens. Its repeated-read component is just 4.82K, so rereading cannot explain the large total. The trajectory instead expands through substantial first-time reads of `sessions.py`, `compat.py`, `utils.py`, and related tests and reproductions. LightMem similarly reaches 379.67K read tokens, with 79 main-agent turns accompanied by 80 extraction calls. The case demonstrates that a file-state or reread warning answers whether a target has been visited, not whether its first observation should be a whole file. The evidence policy encourages the coding LLM to relate further acquisition to its current plan and to stop expanding context when it judges a symbol search and bounded local context sufficient.

### D.3 Auxiliary Memory Cost on a Long Localization Path

The task django_django-11964 is intrinsically less local. It requires tracing why newly assigned TextChoices/IntegerChoices values behave as enum members while values reloaded from the database behave as primitive strings or integers. EvidenceCard therefore does not produce an unrealistically short trajectory: it takes 30 requests and reads 79.40K file tokens while following the enum, field-conversion, and assignment paths. Even so, it remains substantially below Repoformer's 257.78K and the other baselines' 575.39K–1.08M.

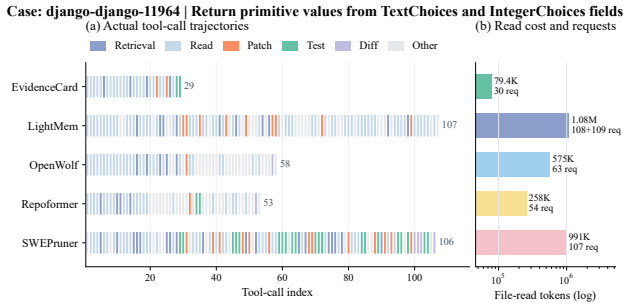



Figure 10: Comparison of (a) actual tool-call trajectories and (b) file-read cost on `SWE-Bench: django_django-11964` under GLM-5.

LightMem uses 108 main-agent turns and 109 extraction calls, for 217 requests and 1.08M file-read tokens. No consolidation call occurs. SWE-Pruner reaches 107 requests and 991.25K read tokens, with 61 evidence-acquisition actions after its first patch. Thus, compressing retained history or pruning the lines within one observation does not necessarily shorten the sequence of hypotheses and inspections that generates that history. EvidenceCard feeds acquisition state back before the next action: it exposes recent observations, previously acquired targets, and phase statistics. The

coding LLM uses this feedback to judge which plan requirements those observations support and whether further evidence is needed before advancing.

### D.4 A Long Verification Tail after the First Patch

In `django_django-15252`, `MigrationRecorder` must respect `router.allow_migrate` before creating the migration table in a multi-database setup. EvidenceCard uses nine evidence actions before its first patch to connect the recorder, router, and database-creation tests. It completes the task in 27 requests with 94.36K file-read tokens. Its post-patch work includes targeted tests and corrections, but remains bounded to 16 evidence-acquisition actions.

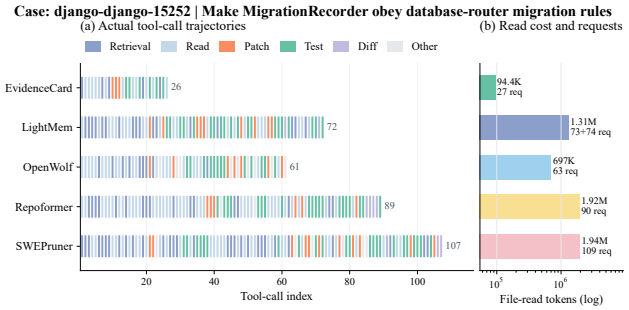



Figure 11: Comparison of (a) actual tool-call trajectories and (b) file-read cost on `SWE-Bench: django_django-15252` under GLM-5.

SWE-Pruner performs 20 pre-patch evidence actions and 71 after the first patch, reaching 109 requests and 1.94M read tokens. Repoformer similarly consumes 90 requests and 1.92M read tokens, while LightMem combines 73 main turns with 74 extraction calls. These trajectories repeatedly broaden into migration loader/executor code, test directories, alternative patches, and verification scripts. Line-level pruning can reduce irrelevant content in a chosen observation, but it does not determine how many locations to open, how many repair hypotheses to keep active, or when verification is sufficient. The phase distribution and accumulated evidence state in EvidenceCard directly target this post-patch expansion.

### D.5 Boundary Case: EvidenceCard Is Not Uniformly Minimal

EvidenceCard is a soft control signal rather than a guarantee of the globally shortest trajectory. In `django_django-10914`, the required functional change is simply to set a consistent default file-upload permission. EvidenceCard finds and patches the setting on its third tool call, but then expands into release-note editing, test correction, and repeated documentation inspection. It finishes successfully in 24 requests and reads 45.63K file tokens.

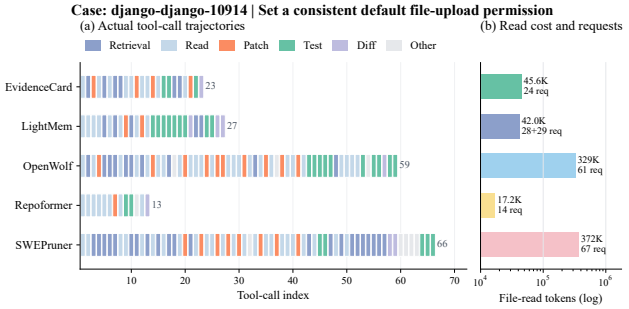



Figure 12: Comparison of (a) actual tool-call trajectories and (b) file-read cost on `SWE-Bench: django_django-10914` under GLM-5.

Repoformer is shorter on this instance: it makes 14 requests, reads 17.19K tokens, changes the setting, and performs focused verification. LightMem also reads slightly less than EvidenceCard at 41.98K, although its 28 main turns trigger 29 extractions and therefore 57 total requests. Conversely, OpenWolf and SWE-Pruner expand much further to 329.19K and 371.83K read tokens. This counterexample bounds the claim made in the paper. EvidenceCard substantially reduces aggregate and long-tail context cost, but the base agent may still decide that documentation or additional tests are necessary after the ledger reports an early patch. The Card advises evidence reuse, phase transition, and stopping; it does not override the agent's task interpretation.

Across the cases, the baseline overhead has four distinct sources: broad first-time reads, repeated local inspection after a viable patch, prolonged post-patch verification, and auxiliary memory-maintenance requests. These sources are not captured by any single reread or per-observation metric.

EvidenceCard addresses these sources through prompt-guided planning and acquisition feedback across the complete episode, while the boundary case shows why the resulting claim is aggregate rather than per-instance optimality.