

Prism: Persistent Faceted File Memory for Cost-Efficient Coding Agents

Ziyun Zhang^{1,2,*} Yihao Liu^{1,*} Yuzhe Zhang¹ Huaqian Cai^{1,2}

¹Peking University

²Beijing Tongming Lake Information Technology Application Innovation Center

*Equal contribution.

Abstract

Coding agents working on sequential maintenance tasks re-read the same files across related tasks, wasting more than a third of their input tokens on redundant context. The natural fix, caching file summaries, fails to reliably reduce cost: tool-use traces are consistent with agents over-relying on generic descriptions, skipping re-reads, and offsetting token savings with less effective downstream decisions, leaving net cost roughly unchanged. We introduce Prism, a content-hash-keyed file memory that augments summaries with bounded *facets*: identifier-grounded facts distilled from the agent’s own prior reasoning. On 12 SWE-Chain dependency-upgrade chains, Prism reduces cost by 24.9% on GPT-5 and 9.9% on GPT-5.4-mini, while maintaining comparable aggregate Build+Fix F1. Against no memory, a summary-only ablation yields only a nominal 2.4% cost decrease that falls within run-to-run variability, while a static Aider-style repo-map baseline costs substantially more at 13% over no memory. Two controlled ablations isolate the causal contributions: wiping the persistent store before every task raises cost by 37% over no memory, and replacing facets with a 3× larger unstructured summary raises it by 41%. Code: <https://github.com/WhiteFlamesCN/Prism-Opencode>.

1 Introduction

Coding agents are overwhelmingly benchmarked as stateless problem solvers: each task is presented in isolation, the agent reads what it needs, produces a patch, and forgets everything. Yet real-world code maintenance is inherently sequential: dependency upgrades, refactoring campaigns, and bug-fix chains all require an agent to revisit the same repository across related tasks. Despite growing interest in agent memory, persistent file-level memory for coding agents has received little direct empirical study, leaving open what to cache and how to key it.

When the same agent works on the same repository across multiple tasks, it repeatedly reads and reasons about identical files, incurring redundant token cost. In our main experiment (§4.2), Prism reduces fresh input tokens by 37%, indicating substantial redundancy in the stateless baseline.

A natural response is to cache file summaries. Yet our summary-only ablation is roughly as costly as no memory (§4.3). The limitation is not merely that summaries are short, but that they provide a single, largely static view of a file: its classes, methods, and module-level structure. Later tasks often depend on different knowledge—why the agent inspected the file, which identifiers mattered, what behavioral constraint or cross-file relationship it uncovered. Prism therefore treats file memory as accretive rather than one-shot. Each prior read can add a small, identifier-grounded *facet*: a task-relevant fact distilled from the agent’s own reasoning. Over time, these facets illuminate the same file from multiple angles, preserving the working knowledge that summaries omit and helping the agent decide when rereading is actually necessary.

We introduce Prism, a minimal-intervention file memory built on three design choices (Figure 1): (1) content-hash keying for automatic invalidation on edit; (2) facet accumulation: after each reasoning episode, Prism appends at most one identifier-grounded declarative fact, derived from the agent’s own reasoning, per recently-read file, each adding a new task-relevant angle to the file’s memory; (3) agent-controlled access via a single `read_summary` tool invoked at the agent’s discretion.

On SWE-Chain, Prism reduces cost by 24.9% with GPT-5 and 9.9% with GPT-5.4-mini, without measurable Build+Fix F1 loss (§4.6). The concise structural-summary-only variant yields only a nominal 2.4% cost decrease that falls within run-to-run variability, while a static repo-map baseline costs 13% more than no memory (§4.3). Causal ablations further show that resetting memory be-

tween tasks raises cost by 37%, and replacing facets with a $3\times$ larger unstructured summary raises it by 41% (§4.4). Together, the ablations show that Prism’s savings require both cross-task persistence and facet-structured facts produced while solving earlier tasks.

Contributions.

1. We introduce Prism, a content-hash-keyed file memory for coding agents that stores compact summaries plus bounded, task-derived facets distilled from prior agent reasoning. Prism adds only one new `read_summary` tool and makes memory reusable across tasks by keying entries to file contents rather than paths or episodes.
2. On SWE-Chain, Prism improves the cost-quality frontier: it reduces token cost by 24.9% with GPT-5 and 9.9% with GPT-5.4-mini across 12 dependency-upgrade chains, while keeping aggregate Build+Fix F1 within run-to-run noise of no-memory agents. Usage analysis shows the gains come from reuse: over 95% of `read_summary` calls target files not read in the current task, and about two-thirds target files seen in prior tasks.
3. We show why naive memory fails and which ingredients are necessary. Summary-only memory saves only 2.4% within noise, and a static repo map costs 13% more, so neither improves the frontier. Controlled ablations show that removing cross-task persistence raises cost by 37% over OFF, and replacing facets with a $3\times$ larger unstructured summary raises it by 41%, making memory more expensive than no memory.

2 Related Work

Agent memory systems. MemGPT (Packer et al., 2023) pages memory between context and storage, while Voyager (Wang et al., 2024) accumulates skills. ExpeL (Zhao et al., 2024) distills episodic experience and Reflexion (Shinn et al., 2023) reflects on failures. More recent work focuses on structured and long-term memory: A-MEM (Xu et al., 2025) introduces agentic self-organizing memory; MemoryBank (Zhong et al., 2024) maintains evolving long-term profiles. Generative Agents (Park et al., 2023) use retrieval

over event streams, and LongMem (Wang et al., 2023) augments language models with a residual side memory. These systems key memory on tasks, episodes, or events rather than content-hashed files, so answering file-specific questions generally requires additional retrieval or reprocessing. GPTCache (Bang and Feng, 2023) and Prompt Cache (Gim et al., 2024) reduce cost through query-similarity or prefix-reuse caching at the inference layer. Prism caches synthesized knowledge keyed by file content hash, and is orthogonal and composable with inference-level caching.

Code understanding and retrieval. RepoCoder (Zhang et al., 2023) and Repoformer (Wu et al., 2024) retrieve code snippets by embedding similarity. Repo-maps (Gauthier, 2024) provide static structural navigation, RepoAgent (Luo et al., 2024) generates repository-level documentation, and CodePlan (Bairi et al., 2024) uses planning over repository structure for multi-file edits. These primarily retrieve or generate text rather than synthesized understanding, and do not by themselves provide explicit content-hash invalidation under edits. Naive batch file summaries become stale after edits unless explicitly regenerated, and they cannot incorporate what the agent learned while solving previous tasks. Prism is complementary to retrieval: retrieval finds candidate code, but does not retain what an agent later worked out about it. We evaluate one such repo-map baseline in §4.3, where it costs more than no memory at all.

Coding-agent frameworks and benchmarks. SWE-agent (Yang et al., 2024), Agentless (Xia et al., 2024), AutoCodeRover (Zhang et al., 2024), and Devin (Cognition AI, 2024) solve isolated tasks; SWE-Chain (Lam et al., 2025) extends SWE-bench (Jimenez et al., 2024) to sequential chains. SWE-Gym (Pan et al., 2025) provides training environments for coding agents; SWE-Lancer (Miserendino et al., 2025) evaluates on real freelance tasks. Prism is orthogonal to these: a persistent state layer composable with tool-using agents that expose file reads and visible intermediate text.

3 Prism

Real-world code maintenance consists of related tasks over a repository that is both revisited and changing. This creates three practical pressures: memory must survive beyond a single task without becoming stale when files are edited; it must

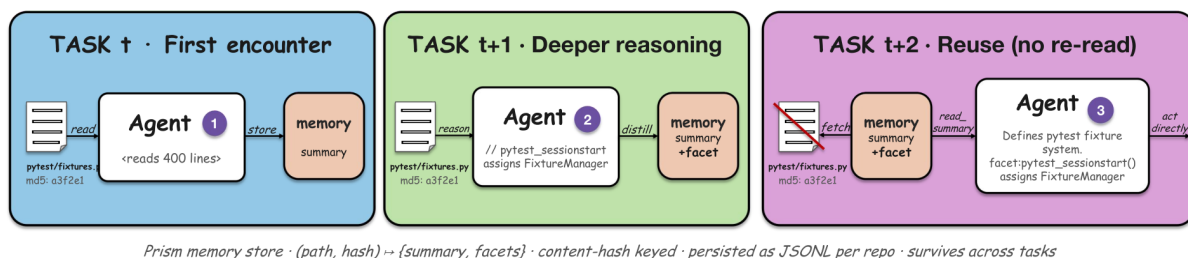


Figure 1: How Prism’s file memory evolves across tasks. In Task 1, the agent reads the full file and Prism generates a base summary. In Task 2, the agent’s reasoning is distilled into a facet: a concrete operational fact. By Task 3, the accumulated memory (summary + facets) is rich enough for the agent to act without re-reading the file. Content-hash checks automatically invalidate stale entries on edit.

reduce redundant file reads rather than merely summarize past conversations; and it must capture operational facts the agent actually used, which may not be apparent from static structure alone. Prism is a thin layer on file access that addresses these pressures by caching file-scoped, versioned facets derived from the agent’s prior interactions and refining them across tasks.

3.1 Architecture Overview

Prism preserves the agent’s ordinary file-system workflow and adds one optional tool, `read_summary(path)`. It interposes a memory layer that observes file reads and reasoning steps, runs LLM workers at three points, and keeps their outputs in an in-memory cache backed by an append-only on-disk store for invalidation and reuse across tasks.

At runtime, Prism acts on three triggers (Figure 2). (1) When the agent reads a file, Prism schedules a background, content-hash-keyed summary (§3.2). (2) After each agent reasoning chunk, a distiller extracts at most one new facet for each recently read file (§3.3). (3) When the agent calls `read_summary(path)`, an aggregator synthesizes stored facets into a compact briefing (§3.4), letting the agent avoid rereading full contents when cached understanding suffices.

Memory-store notation. Let $H(p)$ denote the hash of p ’s current contents. Prism’s file memory is a partial map $M : (p, h) \mapsto (s, F)$, where s is the base summary and F is a set of accumulated facets with $|F| \leq K$. A lookup for p consults only $M[p, H(p)]$; entries $M[p, h]$ with $h \neq H(p)$ may remain on disk but are stale for the current file and are not returned.

3.2 Content-Hash-Based Summaries

Each file’s memory entry is keyed by (path, $\text{hash}(\text{content})$). On first read, Prism schedules a background summarization (non-blocking). The summary is generated from a bounded prefix of the file: a 2–4 sentence sketch of its top-level structure and key interfaces.

Automatic invalidation. Any edit effectively changes the content hash; the next lookup misses and triggers re-generation (old entries remain on disk but are not returned unless the file content again matches their hash). No TTL or explicit invalidation is needed.

Two-tier lookup. Before invoking the summarizer LLM, Prism checks: (1) in-memory cache, (2) on-disk entries matching the current hash. It calls the LLM only on a complete miss, preventing repeated summarizer calls for the same unchanged file.

3.3 Facet Accumulation

Summaries describe a file’s static structure; facets fill in the operational knowledge the agent develops while working in the repo. Prism builds this knowledge up lazily, one fact at a time: after each agent reasoning step, it tries to distill at most one new facet for each recently-read file.

Distillation trigger. After each non-trivial reasoning or text block the agent emits between tool calls, Prism pairs the text with files most recently read by the agent and, for each, attempts to distill one declarative fact. Prism records any inter-tool content the model API exposes: visible assistant text and, for reasoning-capable models, the emitted reasoning/thinking stream. It does not require logs, activations, attention, or other model-internal

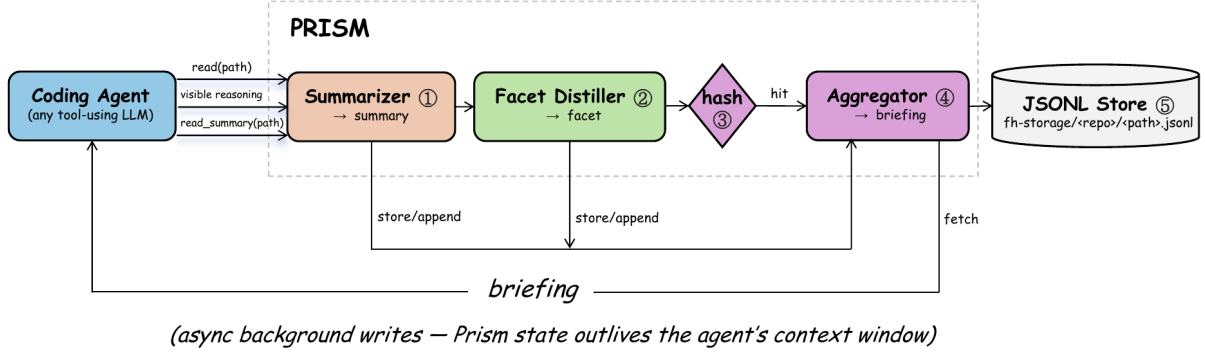


Figure 2: Prism architecture. The agent’s read triggers background summarization; visible reasoning feeds facet distillation; read_summary returns an aggregated briefing. Content-hash checks gate all lookups. The dashed boundary encloses Prism components; only the read_summary tool is visible to the agent.

access, so the same facet-distillation hook applies to both reasoning models and ordinary tool-using agents.

Distiller interface. The distiller receives two classes of input: material already in memory—the current base summary and existing facets—which serve as negative constraints against restating or paraphrasing known content, and source material—the file path and a bounded window of the agent’s reasoning—from which any new facet must be drawn. It must output either a single sentence satisfying two requirements or the literal token `NO_NEW_FACET`. A *timeless* facet states a reusable fact about the code, not an action or observation tied to the current task; an *identifier-grounded* facet names concrete code identifiers rather than vague components. For example, “Query.clone() preserves the combined_queries tuple when duplicating combinator querysets” satisfies both requirements. By contrast, “Add a regression test for Query.clone()” fails timelessness, while “The pagination test suite covers combinator querysets” fails identifier-grounding. If no non-duplicative facet satisfies both requirements, the distiller emits `NO_NEW_FACET`.

Bounded accumulation. At most K facets per (file, hash) pair. Prism drops new facets beyond this small constant cap, bounding per-file context cost.

Example. A file `config.py` might accumulate: base summary “Defines the Config class with YAML loading and validation”; facets: (1) “load_yaml() raises ConfigError on missing ‘version’ key (line 47)”; (2) “validate() accepts optional `strict=True` that rejects unknown keys.”

Algorithm 1 Prism: update and lookup

```

1: On read( $p$ ):
2:    $h \leftarrow H(p)$ 
3:   if  $M[p, h].s = \emptyset$  then
4:     schedule background:  $M[p, h].s \leftarrow$ 
       SUMMARIZE( $p$ )
5:     push  $p$  to read-queue (bounded)
6:
7:   On reasoning chunk  $r$ :
8:   for each  $p$  in read-queue:
9:      $h \leftarrow H(p)$ 
10:    if  $|F_{p,h}| < K$  then
11:       $f \leftarrow \text{DISTILL}(p, M[p, h].s, F_{p,h}, r)$ 
12:      if  $f \neq \text{NO\_NEW\_FACET}$  then
13:         $F_{p,h} \leftarrow F_{p,h} \cup \{f\}$ ; invalidate agg. cache
14:
15:   On read_summary( $p$ ):
16:      $h \leftarrow H(p)$ 
17:     if  $(p, h) \notin \text{dom}(M)$  or  $M[p, h].s = \emptyset$  then
18:        $M[p, h].s \leftarrow \text{SUMMARIZE}(p)$  {synchronous on miss}
19:      $(s, F) \leftarrow M[p, h]$ 
20:     if  $F = \emptyset$  then return  $s$ 
21:   return  $\text{CACHEDAGGREGATE}(p, h, |F|, s, F)$  {cache keyed by  $(p, h, |F|)$ }

```

Each facet pins down one concrete, reusable angle on the file.

3.4 On-Demand Aggregation

When the agent calls `read_summary(path)`, Prism returns the cached base summary if no facets have been accumulated yet, or invokes an aggregator LLM otherwise. The aggregator receives both the base summary $s_{p,h}$ and the current facets $F_{p,h}$, and synthesizes them into a coherent 2–4 sentence briefing intended to substitute for the base summary in downstream reads. The aggregated result is cached, keyed by (path, hash, facet_count), and invalidated when a new facet arrives or the content hash changes. If `read_summary` is called for a file

with no usable cached summary (never read, or background summarization still pending), Prism generates the summary synchronously before returning.

3.5 Cross-Task Persistence and Amortization

The on-disk cache persists across sequential tasks within the same repository; content-hash checks handle files modified between tasks (stale entries are simply missed and regenerated). Knowledge accumulated in task t amortizes cost in tasks $t + 1, t + 2, \dots$

Let M be the one-time cost of constructing memory for a file, A the per-use aggregation cost, R the cost of a full re-read, and S the cost of reading only the briefing. Assuming $(R - S) - A > 0$, Prism pays off when:

$$n > \frac{M}{(R - S) - A} \quad (1)$$

where n is the number of future re-reads and M is the one-time memory-construction overhead for that file. A is amortized: the aggregator is invoked once per facet update and its result is cached across subsequent `read_summary` calls (§3.4), so A is paid roughly per facet addition rather than per call. Long chains thus amortize memory construction; short chains may not reach break-even.

3.6 Implementation Details

Background writes are non-blocking; failures never interrupt the agent. Summarization and facet distillation run asynchronously and do not add to the agent’s wall-clock latency; only aggregation (on `read_summary` calls) is synchronous. The summarizer, distiller, and aggregator share one auxiliary LLM (in our experiments, the same model as the backbone); its calls are included in all reported costs. All Prism hyperparameters are listed in Appendix A.1.

4 Experiments

4.1 Experimental Setup

Benchmark. We evaluate on SWE-Chain (Lam et al., 2025): 12 publicly available dependency-upgrade chains across 9 Python repositories (poetry, attrs, conan×2, flask, jinja2, pyjwt, pytest×2, urllib3, xarray×2), covering 150 sequential build-and-fix tasks (6–17 per chain).

Metrics. We report Build+Fix F1 following the SWE-Chain leaderboard convention: for each task, the agent must both build the environment under the new dependency version and fix any resulting test failures. We also report total USD cost and token breakdowns for fresh input, output, and cached input.

Backbones. Our main experiments use GPT-5 via Azure OpenAI at temperature 0. We use GPT-5.4-mini, also via Azure OpenAI at temperature 0, to test generalization (§4.5).

Agent framework. We integrate Prism into OpenCode (Anomaly, 2024), an open-source tool-using coding-agent framework for software engineering tasks. Prism adds a single `read_summary` tool to the agent’s tool registry; no other changes to the agent loop are required.

Conditions. We evaluate six conditions spanning three intuitive memory strategies and two causal controls. OFF is a no-memory baseline. NOFACET uses Prism base summaries only: facet distillation is disabled, and `read_summary` returns the base summary directly without aggregation. REPOMAP adds an Aider-style repository map (Gauthier, 2024) of class and function signatures to the system prompt, providing a static code-structure view without Prism’s summarization or persistence. LONGSUMMARY keeps persistence on but replaces facets with a single longer unstructured summary: 1500 output tokens ($3 \times$ NOFACET’s 500), a 20K-character input window ($2.5 \times$ NOFACET’s 8K), and a prompt requesting one to two detailed paragraphs. PRISM-RESET is identical to PRISM but clears both the in-memory cache and on-disk file-memory store before every task in the chain, preserving per-task summary and facet generation while removing cross-task amortization. PRISM is the full system.

Protocol and statistics. Each condition is run 3 times per chain, and we report arithmetic means over replicates. Caches persist forward within each chain but are not shared across chains; each replicate starts from an empty memory store and is reset before the next replicate, condition, or chain. Sign tests, bootstrap confidence intervals, and per-chain cost-accounting details are provided in Appendix B and Appendix A.2.

4.2 Main Result: GPT-5

Table 1 shows per-chain results. Prism reduces total cost from \$151.88 to \$114.11, a 24.9% reduction, while maintaining comparable F1 at macro 0.153 versus 0.150; fresh input tokens drop 36.9% from 57,568K to 36,329K. The 24.9% dollar reduction is smaller than the 36.9% fresh-token reduction for two reasons: (1) output tokens, which are $8\times$ more expensive per token than fresh input under GPT-5 pricing, drop only 6%; (2) Prism’s own auxiliary LLM calls for summarization, distillation, and aggregation add a small overhead under 5% of total cost that partially offsets the input savings. The largest savings occur on high-baseline-cost chains: xarray_2022 saves 53.2%, pytest_8.0 48.4%, and xarray_2025 46.9%. These chains involve many sequential tasks in the same repository, maximizing the amortization of memory-construction cost.

One chain shows a cost regression: poetry rises 12.7% because memory-construction overhead exceeds future-task amortization on this shortest chain. F1 is unchanged on 10 of 12 chains and slightly higher on jinja2 from 0.624 to 0.653 and on urllib3 from 0.503 to 0.508, indicating that the cost savings do not come at the expense of patch quality. On chains where Prism costs more, patch quality is likewise not degraded.

4.3 Why Naive Memory Fails

Three intuitive approaches to file memory all fail to improve over the no-memory baseline (Table 2).

NOFACET (summary-only, persistent). Removing facets and the aggregation step yields a system effectively indistinguishable from OFF in cost: \$148.30 versus \$151.88, a 2.4% drop that falls within the $\sim 8\%$ inter-run cost CV, with comparable F1. Base summaries capture file structure (“defines class X with methods Y, Z”) but not the behavioral details the agent needs. NOFACET produces the highest no-follow-up rate among all conditions: 70.5% of read_summary calls are not followed by a full read within 10 tool calls. It still matches OFF in cost, while PRISM has a lower 52.6% no-follow-up rate but is the cheapest. This pattern is consistent with structural summaries giving the agent false confidence: it acts without sufficient grounding, producing longer trajectories that offset the token savings from skipped reads (Appendix D.3).

REPOMAP (static structure, no persistence). The repo-map baseline costs \$171.56, +13% over

OFF and 50.3% more than PRISM. It adds fresh input on every session start across all 150 chain steps; the overhead is acute on large repositories, where xarray chains increase by $\sim 85\%$. attrs and pyjwt benefit modestly from navigational shortcuts, dropping 25% and 13% respectively, but the aggregate is net-negative.

LongSummary (more tokens, persistent). If Prism’s advantage stemmed from carrying more total context per file, a longer summary should recover the savings. LongSummary uses $3\times$ the output budget and $2.5\times$ the input window of NOFACET, requesting detailed paragraphs instead of bounded facets. Instead it is the most expensive condition: \$213.96, +40.9% over OFF and +87.5% over PRISM, with 11/12 chains more expensive than OFF. Raw token budget without query-conditioned structure becomes overhead rather than memory.

Why facets succeed where summaries fail. Figure 3 illustrates the contrast qualitatively. The quantitative signal is in the no-follow-up rates above: NOFACET skips re-reads most often at 70.5% yet saves no cost, while PRISM skips less often at 52.6% but is the cheapest condition overall. This suggests that facets do not merely suppress re-reads—they help the agent choose correctly when to skip, by providing identifier-level operational facts (return values on edge cases, parameter constraints) rather than generic structural descriptions.

4.4 Causal Isolation: Cross-Task Persistence

The preceding section shows that three alternative memory designs fail, but leaves one confound open: Prism’s faceted memory could be acting as a per-task summary cache rather than a true cross-task store. PRISM-Reset isolates this by clearing both the in-memory cache and the on-disk file-memory store before every task in the chain, preserving per-task summary and facet generation while removing cross-task amortization.

Cost rises to \$208.44, +37.2% over OFF and +82.7% over Prism, with 12/12 chains more expensive than OFF (per-chain detail in Appendix C.2). Summary generation, facet distillation, and aggregation are therefore not net-beneficial as a per-task cache; the cross-task persistence layer is what turns extraction overhead into a net saving.

Factorial interpretation. The conditions in Table 2 approximate a factorial contrast over persis-

Chain	OFF Cost (\$)	Prism Cost (\$)	Δ Cost (%)	OFF F1	Prism F1
poetry	6.16	6.94	+12.7	0.160	0.160
attrs	15.21	13.20	-13.2	0.033	0.033
conan 2.12	10.92	7.34	-32.8	0.154	0.154
conan 2.23	9.13	8.97	-1.8	0.125	0.125
flask	14.83	13.56	-8.6	0.013	0.013
jinja2	11.53	11.00	-4.6	0.624	0.653
pyjwt	9.57	7.76	-19.0	0.034	0.034
pytest 7.0	19.52	14.49	-25.7	0.077	0.077
pytest 8.0	17.05	8.80	-48.4	0.028	0.028
urllib3	11.61	8.92	-23.2	0.503	0.508
xarray 2022	13.58	6.35	-53.2	0.014	0.014
xarray 2025	12.77	6.78	-46.9	0.031	0.031
Aggregate	151.88	114.11	-24.9	0.150	0.153

Table 1: GPT-5: Prism reduces cost by 24.9% while maintaining comparable F1 (0.150→0.153).

Condition	Persist	Facets	Cost (\$)	Δ OFF	F1
OFF	—	—	151.88	—	.150
NOFACET	✓	—	148.30	-2.4%	.147
REPOMAP	—	—	171.56	+13.0%	.147
LongSummary	✓	— [†]	213.96	+40.9%	.149
PRISM-Reset	—	✓	208.44	+37.2%	.154
PRISM (full)	✓	✓	114.11	-24.9%	.153

Table 2: Aggregate comparison of all six conditions (GPT-5, 12 chains). Three intuitive memory strategies fail: summary-only (NOFACET) is cost-neutral, static maps (REPOMAP) add overhead, and a $3\times$ larger unstructured summary (LongSummary[†]) is the most expensive. The causal control PRISM-Reset (facets without cross-task persistence) is equally harmful. Only the full Prism yields a meaningful cost reduction. Per-chain details in Appendix C.1 and C.2.

tence and facet structure:

- Persistence without facets is break-even: NOFACET saves only 2.4%.
- More tokens without structure is harmful: LongSummary costs 40.9% more.
- Facets without persistence is harmful: PRISM-Reset costs 37.2% more.
- Both together yield the 24.9% saving of full Prism.

Only the combination—persistent storage and bounded, agent-derived facets—yields cost reduction.

4.5 Generalization: GPT-5.4-mini

Table 3 shows that Prism also reduces cost with GPT-5.4-mini, lowering aggregate spend from \$79.47 to \$71.58, a 9.9% reduction. The reduction is broad: 11 of 12 chains are cheaper, and xarray_2025 is the only regression at 22.1% over OFF.

Aggregate F1 is largely unchanged: 9 of 12 chains have identical F1, with only 3 showing non-zero deltas. Although the aggregate rises from

0.112 to 0.157, the bootstrap confidence interval spans zero, so we can only conclude that Prism does not degrade task performance rather than claiming a reliable quality improvement.

4.6 Analysis

Savings are predominantly cross-task. Over 95% of read_summary calls target files not yet read in the current task: 95.9% on GPT-5 and 95.2% on mini. Of these, 62.8% target files encountered in earlier chain tasks (Table 6). The sole GPT-5 cost regression on poetry is the shortest chain, where memory-construction overhead has fewer future tasks to amortize against, consistent with the break-even model (§3.5).

Token-level picture. On GPT-5, fresh input tokens drop 37% from 57,568K to 36,329K while output tokens are largely unchanged with a 6% drop; on mini, output tokens drop 40% from 10,253K to 6,117K and total API calls rise modestly by 14% as the agent issues additional read_summary calls. Prism’s auxiliary LLM calls for summarization, distillation, and aggregation constitute under 5% of total cost.

Chain	OFF Cost (\$)	Prism Cost (\$)	Δ Cost (%)	OFF F1	Prism F1
poetry	5.79	4.48	-22.7	0.160	0.160
attrs	8.06	7.43	-7.9	0.033	0.033
conan 2.12	3.27	3.08	-5.9	0.048	0.185
conan 2.23	2.24	2.14	-4.7	0.165	0.136
flask	7.45	7.15	-3.9	0.013	0.013
jinja2	6.19	4.83	-22.0	0.226	0.662
pyjwt	6.20	4.10	-33.8	0.034	0.034
pytest 7.0	8.67	8.01	-7.5	0.077	0.077
pytest 8.0	7.04	6.45	-8.4	0.028	0.028
urllib3	7.21	5.68	-21.3	0.508	0.508
xarray 2022	8.96	7.99	-10.9	0.014	0.014
xarray 2025	8.39	10.24	+22.1	0.031	0.031
Aggregate	79.47	71.58	-9.9	0.112	0.157

Table 3: GPT-5.4-mini: Prism reduces cost by 9.9% (11/12 chains cheaper), confirming that the mechanism transfers across model scales.

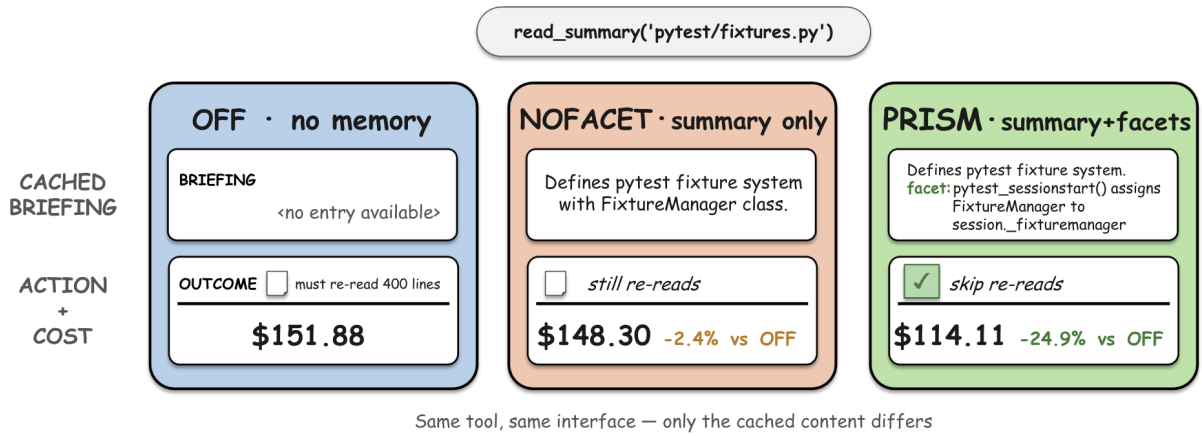


Figure 3: What the agent sees under each condition when calling `read_summary` for the same file. OFF does not have this tool. NOFACET returns a structural summary, which often does not prevent a full re-read. PRISM returns a facet-augmented briefing the agent can act on.

Mechanism evidence. Execution logs match the intended mechanism (Appendix D). The tool-call reduction is concentrated in full file reads, which drop 27% with 11 of 12 chains showing fewer full reads; editing activity is unchanged (Figure 4). Cumulative fresh tokens diverge from task 2 onward (Figure 5), consistent with cross-task accumulation. PRISM maintains a low full-read probability of 25–40% across repeated file encounters; NOFACET degrades toward 55% by the 8th encounter (Figure 6): faceted summaries stay useful across encounters, structural ones decay.

Cost regressions. The two backbones each have one cost regression, but for different reasons. On GPT-5, poetry rises 13% and is the shortest chain at 6 tasks, leaving memory-construction overhead too few future tasks to amortize against. On GPT-5.4-mini, xarray_2025 rises 22.1%, stemming

from a different mechanism: GPT-5.4-mini invokes `read_summary` at roughly half the rate of GPT-5 under the same prompt, so memory-construction cost is not recouped through sufficient reuse. This lower tool-adoption rate also explains why mini’s overall 9.9% savings are smaller than GPT-5’s 24.9%: different models exhibit different adoption rates for novel optional tools, and mini’s lower `read_summary` usage leaves more cached memory under-utilized.

5 Conclusion

We introduced Prism, a persistent file-level memory that augments structural summaries with experience-derived facets, keyed by content hash for automatic invalidation. On 12 SWE-Chain dependency-upgrade chains, Prism reduces cost by 24.9% (GPT-5) and 9.9% (GPT-5.4-mini), with F1 within run-to-run noise of the no-memory base-

line on both backbones. Controlled ablations confirm both Prism ingredients are necessary: removing cross-task persistence or replacing facets with a $3\times$ larger unstructured summary each pushes cost above the no-memory baseline. More broadly, Prism suggests a design principle for cost-efficient coding agents: leave the agent loop unchanged, but cache hash-keyed operational facts learned from experience rather than structure alone.

Limitations

- **File-level granularity.** Prism uses files as the unit of persistence, which keeps invalidation simple through content hashes and makes the method easy to attach to existing agents. Finer-grained symbol- or region-level memories could be layered on top of Prism to capture additional reuse within very large files.
- **Reliance on agent-initiated memory access.** Prism’s savings depend on the agent choosing to call `read_summary`; a populated cache that is never queried only adds construction overhead. We observe that backbones differ substantially in how often they adopt this optional tool (GPT-5.4-mini invokes it at roughly half the rate of GPT-5), which directly caps Prism’s savings on lower-adoption models. A more proactive policy that injects summaries on selected reads rather than waiting for the agent to ask could narrow this gap.
- **Memory-model overhead.** Prism currently invokes the same backbone LLM for summary and facet generation. Distilling these into a smaller, dedicated memory model would reduce the per-file write cost and make the approach more practical for high-frequency editing sessions.

References

- Anomaly. 2024. OpenCode: An open-source tool-using coding agent. <https://github.com/anomalyco/opencode>. Open-source software.
- Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sri-ram Rajamani, B Ashok, and Shashank Shet. 2024. CodePlan: Repository-level coding using LLMs and planning. In *FSE*. ArXiv:2309.12499.
- Fu Bang and Di Feng. 2023. GPTCache: An open-source semantic cache for LLM applications enabling faster answers and cost savings. In *NLP-OSS Workshop at EMNLP*.
- Cognition AI. 2024. Introducing devin, the first AI software engineer. <https://cognition.ai/blog/introducing-devin>.
- Paul Gauthier. 2024. Aider: AI pair programming in your terminal. <https://aider.chat>. Open-source software.
- In Gim, Guojun Chen, Seung-Seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. In *MLSys*.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*.
- Man Ho Lam, Chaozheng Wang, Hange Liu, Jingyu Xiao, Haau-sing Li, Jen-tse Huang, Terry Yue Zhuo, and Michael R. Lyu. 2025. SWE-Chain: Benchmarking coding agents on chained release-level package upgrades. <https://github.com/CUHK-ARISE/SWE-Chain>.
- Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. 2024. RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation. In *EMNLP (System Demonstrations)*. ArXiv:2402.16667.
- Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. 2025. SWE-Lancer: Can frontier LLMs earn \$1 million from real-world freelance software engineering? arXiv preprint arXiv:2502.12115.
- Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. 2023. MemGPT: Towards LLMs as operating systems. arXiv preprint arXiv:2310.08560.
- Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training software engineering agents and verifiers with SWE-Gym. In *ICML*. ArXiv:2412.21139.
- Joon Sung Park, Joseph C O’Brien, Carrie J Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *UIST*.
- Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research (TMLR)*.

Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, and Furu Wei. 2023. Augmenting language models with long-term memory. In *NeurIPS*. ArXiv:2306.07174.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoforger: Selective retrieval for repository-level code completion. arXiv preprint arXiv:2403.10059. Also in ICML 2024.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based software engineering agents. arXiv preprint arXiv:2407.01489.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. A-Mem: Agentic memory for LLM agents. In *NeurIPS*. ArXiv:2502.12110.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. arXiv preprint arXiv:2303.12570. Also in EMNLP 2023.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous program improvement. arXiv preprint arXiv:2404.05427. Also in ISSTA 2024.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. ExpeL: LLM agents are experiential learners. In *AAAI*.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2024. MemoryBank: Enhancing large language models with long-term memory. In *AAAI*.

A Experimental and Implementation Details

A.1 Prism Hyperparameters

All experiments use the following Prism parameters: facet distillation input capped at 2,500 characters; at most $K = 8$ facets per (file, hash); recent-read queue length 16; memory-model output capped at 500 tokens.

A.2 Cost Accounting and Pricing

GPT-5 is priced at $(f \cdot 1.25 + c \cdot 0.125 + o \cdot 10)/10^6$ and mini at $(f \cdot 0.25 + c \cdot 0.025 + o \cdot 2)/10^6$, where f , c , o are fresh, cached, and output tokens. Memory-model calls share the same pricing tier and are already counted in PRISM’s reported cost, where they account for under 5%.

B Robustness Statistics

All statistics use a paired bootstrap over the 12 chains with 10K resamples.

Cost savings (OFF – PRISM). On GPT-5, the mean per-chain saving is \$3.15 (95% CI [0.62, 5.10]), corresponding to an aggregate reduction of 24.9% (95% CI [4.5, 40.0]). On GPT-5.4-mini, the mean saving is \$0.66 (95% CI [0.06, 1.15]), i.e. 9.9% (95% CI [1.1, 17.4]). Both intervals exclude zero. A non-parametric sign test agrees: 11 of 12 chains favor PRISM on each backbone ($p = .003$).

F1 changes (PRISM – OFF). Effects are small and not uniformly robust. The mean per-chain change is +0.003 on GPT-5 with 95% CI [−.001, +.012] and +0.045 on mini with 95% CI [−.005, +.129]; only 2 of 12 GPT-5 chains and 3 of 12 mini chains show non-zero changes. Removing jinja2 drops the GPT-5 aggregate F1 gain to +0.000, and the mini aggregate drops to +0.010. We therefore treat F1 as a safety check rather than a claimed improvement. The tiny negative GPT-5 CI lower endpoint of −0.001 reflects resampling uncertainty, not an observed chain-level degradation; the mini CI includes zero outright.

C Per-Chain Results for Additional Conditions

C.1 Alternative Baselines: NOFACET and REPOMAP

Table 4 reports per-chain cost and F1 for NOFACET and REPOMAP under GPT-5, whose aggregates appear in §4.3. NOFACET’s aggregate cost matches OFF within run-to-run noise: summary-only memory is cost-neutral. REPOMAP’s regressions cluster on large repositories (xarray) and on the shortest chain (poetry).

C.2 Causal-Isolation Ablations: PRISM-Reset and LongSummary

Table 5 reports per-chain results for the two causal-isolation conditions of §4.4. PRISM-Reset (Prism

Chain	NOFACET			REPOMAP		
	Cost (\$)	Δ OFF (%)	F1	Cost (\$)	Δ OFF (%)	F1
poetry	5.98	-2.9	0.160	11.52	+87.0	0.160
attrs	12.76	-16.1	0.033	11.47	-24.6	0.033
conan 2.12	12.01	+10.0	0.154	8.83	-19.1	0.069
conan 2.23	10.48	+14.8	0.099	8.25	-9.6	0.136
flask	14.30	-3.6	0.013	14.05	-5.3	0.013
jinja2	9.67	-16.1	0.611	11.65	+1.0	0.660
pyjwt	7.28	-23.9	0.034	8.36	-12.6	0.034
pytest 7.0	17.90	-8.3	0.077	19.53	+0.1	0.077
pytest 8.0	13.19	-22.6	0.028	14.97	-12.2	0.028
urllib3	11.31	-2.6	0.510	14.07	+21.2	0.510
xarray 2022	15.92	+17.2	0.014	24.04	+77.0	0.014
xarray 2025	17.50	+37.0	0.031	24.80	+94.2	0.031
Aggregate	148.30	-2.4	0.147	171.56	+12.9	0.147

Table 4: Per-chain cost and F1 for the two baseline conditions (GPT-5). Δ OFF is the cost change relative to the OFF baseline (\$151.88 aggregate).

with the file-memory store wiped before every task) is more expensive than OFF on all 12 chains, and LongSummary (NOFACET with a $3\times$ larger summary output and $2.5\times$ larger input window) is more expensive than OFF on 11 of 12 chains, confirming that the aggregate regressions are not driven by a few outliers.

D Mechanism Analyses

The following analyses test whether PRISM’s savings match the intended mechanism: persistent file memories should replace repeated full-file reads for files encountered in earlier tasks. Editing behavior and task quality should be largely unchanged.

D.1 Tool-Call Distribution

Figure 4 compares the tool-call mix across the four file-memory conditions, restricted to the file-access, editing, and search families. PRISM’s reduction concentrates on full read calls; apply_patch and search counts are largely unchanged, isolating the savings to how files are accessed rather than how often the agent edits or searches. REPOMAP has the highest read count.

D.2 Cross-Task File Reuse

Because Prism creates a memory entry after each full read and validates it by content hash, a read_summary call on a previously encountered file whose content has not changed is an operational cache hit; calls without a valid (path, hash) entry return a miss. We therefore treat prior-task file reuse (Table 6) as a practical proxy for the summary-cache hit rate.

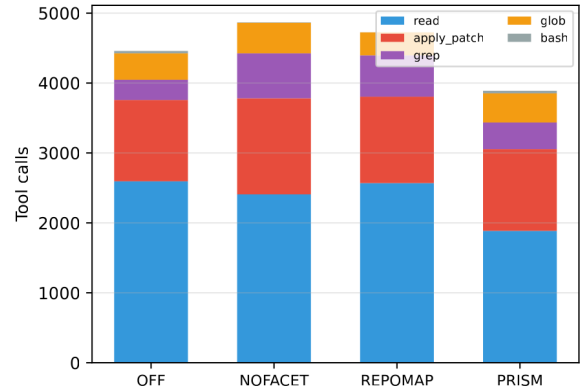


Figure 4: Tool-call distribution across conditions (GPT-5, all 12 chains), counting only the file-access (read), editing (apply_patch), and search (grep/glob/bash) families.

We also examined alignment between pre-patch read_summary calls and eventually-patched files: PRISM achieves slightly higher recall of 0.608 versus NOFACET’s 0.578 at comparable precision of 0.322 versus 0.318 over 138 versus 143 tasks. The effect is modest but points toward faceted summaries aiding pre-edit file selection.

D.3 Follow-Up Reads After Summary Access

For each read_summary call, we check whether the agent follows up with a full read of the same file within 10 subsequent tool calls. A lower follow-up rate suggests the briefing was sufficient for the agent to proceed. On GPT-5, NOFACET has a no-follow-up rate of 70.5% versus 52.6% for PRISM; on GPT-5.4-mini the rates are 63.8% for NOFACET and 58.5% for PRISM.

NOFACET has the highest no-follow-up rate

Chain	PRISM-Reset			LongSummary		
	Cost (\$)	ΔOFF (%)	F1	Cost (\$)	ΔOFF (%)	F1
poetry	10.07	+63.5	0.160	11.20	+81.8	0.160
attrs	20.36	+33.9	0.033	21.96	+44.4	0.033
conan 2.12	12.60	+15.4	0.154	15.32	+40.3	0.154
conan 2.23	10.15	+11.2	0.125	10.86	+18.9	0.125
flask	17.24	+16.3	0.013	20.22	+36.3	0.013
jinja2	12.57	+9.0	0.673	13.51	+17.2	0.610
pyjwt	11.88	+24.1	0.034	14.37	+50.2	0.034
pytest 7.0	29.26	+49.9	0.077	28.13	+44.1	0.077
pytest 8.0	18.08	+6.0	0.028	16.73	-1.9	0.028
urllib3	18.93	+63.0	0.510	16.94	+45.9	0.510
xarray 2022	23.16	+70.5	0.014	19.90	+46.5	0.014
xarray 2025	24.14	+89.0	0.031	24.82	+94.4	0.031
Aggregate	208.44	+37.2	0.154	213.96	+40.9	0.149

Table 5: Per-chain cost and F1 for the two causal-isolation conditions (GPT-5). ΔOFF is the cost change relative to the OFF baseline (\$151.88 aggregate).

Condition	1st	Prev	Total	Prev%
OFF read	954	1,640	2,594	63.2
PRISM read	600	1,286	1,886	68.2
PRISM read_sum.	979	1,654	2,633	62.8

Table 6: Full read calls split by whether the file was previously accessed in an earlier task. PRISM reduces total reads by 27% (2,594 $\rightarrow$ 1,886), with savings on both first-seen and previously-seen files. PRISM’s read_summary calls are similarly distributed, with 62.8% targeting previously-seen files. Within-task novelty (file not read earlier in the same task) is 95.9% on GPT-5 and 95.2% on GPT-5.4-mini; prior-task reuse therefore dominates.

yet shows no meaningful cost savings vs. OFF. The structural summary appears to give false confidence: the agent trusts it more often and skips re-reads, but then makes suboptimal decisions that produce longer trajectories. PRISM agents follow up more often, but on better-chosen files: the facet briefing helps identify which files need deeper inspection. Net cost still drops.

D.4 Cumulative Fresh-Token Savings

Figure 5 traces cumulative mean fresh input tokens across task position. All conditions start similarly at task 1, when no memory has yet been accumulated; PRISM diverges from task 2 onward as the persistent file-memory store first contains entries from a prior task, and the gap widens monotonically with chain length. REPOMAP stays highest throughout because its repo-map injection is per-session rather than amortized.

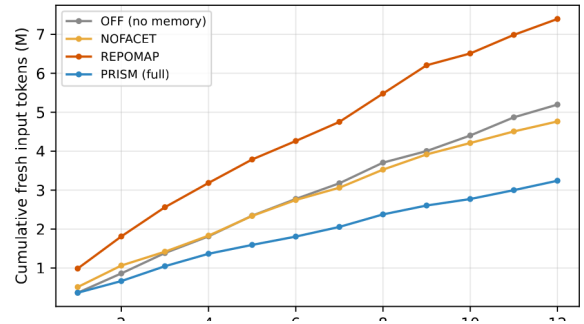


Figure 5: Cumulative mean fresh input tokens across task position (GPT-5, averaged over 12 chains, first 12 tasks).

D.5 Repeated File Encounters

Figure 6 reports the probability that the agent issues a full read call at each successive encounter with the same file, conditioning on encounter index. OFF stays in the 51–68% band regardless of how many times the file has been seen; PRISM starts at 29% and stays in a low 25–40% range across encounters; NOFACET starts comparably low but decays toward 55% by the 8th encounter. Faceted memory keeps replacing full reads as the chain progresses, while structural-only summaries lose sufficiency.

E Facet Quality Analysis

We randomly sampled 200 facets accumulated across our experiments and conducted a blinded two-annotator quality study.

Two-annotator quality study. Two annotators independently judged each facet on three binary dimensions, with access to the corresponding source

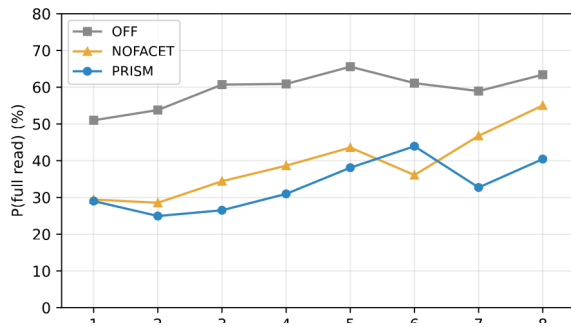


Figure 6: Probability of a full read call at each successive file encounter (GPT-5).

summarizer (§3.2), the facet distiller (§3.3), the aggregator invoked on `read_summary` (§3.4), and the tool description visible to the agent. No other Prism-specific text is added to the system prompt; the agent loop is otherwise unchanged.

file but blinded to chain identity, condition, and task outcome:

1. **Factual correctness:** does the stated behavior match the source code?
2. **Identifier grounding:** does the facet reference concrete code identifiers rather than vague descriptions?
3. **Informativeness:** does the facet convey non-trivial information useful for a future maintenance task?

Inter-annotator agreement (Cohen’s κ) was 0.87 for correctness, 0.83 for grounding, and 0.79 for informativeness. After adjudication, 199/200 facets (99.5%, Wilson 95% CI [97.2%, 99.9%]) were factually correct. 189/200 (94.5%) were identifier-grounded; 182/200 (91.0%) were rated informative. The single factual error overgeneralized a method’s scope.

Representative facets:

- “`pytest_sessionstart(session)` assigns a `FixtureManager` instance to `session._fixturemanager`, initializing the fixture system for the test session.”
- “`HTTPConnectionPool.urlopen()` removes headers listed in `Retry.remove_headers_on_redirect` when following a redirect to a different host.”
- “`ECDSAAlgorithm.prepare_key` enforces that an ECDSA key uses the required curve, rejecting key-family mismatches.”

F Prompts

Figure 7 lists, verbatim, the four prompts that constitute Prism’s only LLM-facing surface area: the

Prism Prompts and Tool Description	
1. Summary Generation. Invoked on first read of a file not yet in the cache.	Summarize this source file in 2-4 sentences. Focus on: purpose, key classes/functions, public API surface, and important invariants or constraints. Do NOT describe the file path or language—focus on behavior. FILE: {filepath} CONTENT (truncated to 8000 chars): {content}
2. Facet Distillation. Invoked after each reasoning episode; variables filled from the file cache and the agent’s visible reasoning window.	An AI coding agent just read a file, then made the following observation/reasoning. FILE PATH: {filepath} EXISTING BASE SUMMARY (do not restate): {summary} EXISTING FACETS (do not paraphrase): {facets} AGENT’S REASONING (after reading): {text, truncated to 2500 chars} TASK: Extract ONE specific, durable, file-level fact. Prefer concrete identifier-level details. Rephrase any task-specific framing into a timeless statement about the file. REQUIREMENTS: mention at least one concrete identifier; phrase as timeless statement about the file’s behavior, structure, or contract; respond NO_NEW_FACET only if reasoning has no concrete file-level details. GOOD: ‘Query.clone()’ preserves the ‘combined_queries’ tuple when duplicating combinator querysets. BAD: The pagination test suite where a regression test should be added.
3. Aggregation. Invoked on read_summary when at least one facet has been distilled for the current (path, hash); when no facets exist, the base summary is returned directly without aggregation.	Multiple AI coding agents have observed the file at {filepath}. Their individual observations (facets) are: {facet_list} TASK: Synthesize these into a coherent 2-4 sentence summary of what this file is and what’s notable about it. Preserve specific symbol names (functions, classes, line numbers) and concrete details. Remove redundancy. If facets contradict, note both views briefly.
4. Agent Tool Description: read_summary. The only Prism-specific text visible to the agent; no instruction to prefer summaries over full reads is given.	read_summary(path): Returns a brief summary of the specified file. Use alongside read whenever helpful—the summary may contain enough context to avoid a full file read.

Figure 7: All Prism prompts and the agent-visible tool description.