EDO: EXPECTATION–OBSERVATION RESIDUALS FOR TOKEN-EFFICIENT LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model agents solve practical tasks through repeated tool interactions, but long tool outputs accumulate in their context and make subsequent inference increasingly expensive. Compressing these outputs requires more than retaining many relevant facts: feedback must also communicate how the actual result confirms or revises the agent's pre-execution expectations. We study this distinction between fact coverage and evidence updating, and introduce EDO, an expectation-conditioned representation of tool feedback. Before each tool call, the main agent writes a compact expected observation grounded in its current state. After execution, a smaller model compares the actual output with this expectation and constructs a semantic residual organized around what the result confirms, corrects, newly reveals, or leaves unresolved. The expectation remains an explicit reference rather than environmental evidence; for long outputs routed through model-based extraction, the expectation–residual pair replaces the raw output in subsequent context. Across six software-engineering and terminal-agent benchmarks, EDO achieves 98.97% of Raw's macro-averaged task score while using, on average, 59.06% of its main-agent tokens and 60.79% of its estimated cost after benchmark-wise normalization. On ProgramBench, it exceeds a matched summary by 4.10 percentage points; offline diagnostics show that it raises correction recall from 70.00% to 94.55% without increasing overall fact recall.

1 INTRODUCTION

Large language model (LLM) agents are becoming an important interface for automating complex tasks, extending language-based reasoning to actions in software environments (Yao et al., 2023; Yang et al., 2024). By repeatedly reasoning, invoking tools, and incorporating environmental feedback, they can carry out workflows that require multiple dependent decisions. Making this interaction process efficient is therefore important for the practical use of these systems. In software engineering and terminal interaction, a single tool call can return an entire source file, a long execution log, or a detailed test report. These observations are appended to the interaction history and carried into subsequent model calls, increasing the amount of context processed over a task. Yet their length can far exceed the evidence needed to decide what to do next: a small part of a long output may determine whether the agent should continue its plan or revise an incorrect assumption. The challenge is therefore to reduce the cost of accumulated tool feedback while preserving the evidence required for reliable subsequent decisions.

Existing approaches address this challenge by compressing individual observations or managing the accumulated trajectory. Input-level methods select tokens, generate summaries, or construct compact representations of long inputs (Jiang et al., 2023; Pan et al., 2024; Xu et al., 2024a; Ge et al., 2024). Trajectory-level methods remove redundant steps or summarize interaction histories as they grow (Xiao et al., 2026; Lu et al., 2026). Within this space, a task-conditioned summarizer can use the task and preceding interaction to prioritize relevant content in a tool result. This focus is useful, but task relevance alone does not specify how a new observation changes the hypothesis that motivated the current action. These approaches reduce the amount of feedback retained for later reasoning, with the quality of the retained representation determining how aggressively the context can be shortened.

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.

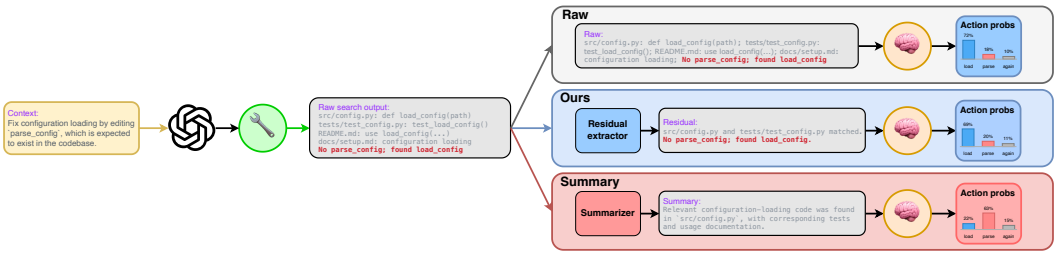

Figure 1: ProgramBench trajectory excerpt comparing Raw, residual feedback, and summarization. The summary omits the derived correction shown in red; right panels report offline next-action samples. Details are in Appendix C.2.

However, fact coverage and evidence updating are not interchangeable. A compressed output may retain many correct and relevant statements while failing to communicate that the observation invalidates an assumption behind the current action. This distinction matters because the next action depends not only on which facts remain, but also on whether the retained representation makes the required belief revision apparent. In the trajectory excerpt in Figure 1, a summary reports that configuration-loading code was found but omits that the predicted function was not found within the searched files. The missing information is not merely another description of the environment; it is the relation between the observed result and the prediction that preceded it. The central challenge is therefore to preserve both what the observation reports and what it confirms, contradicts, newly reveals, or leaves unresolved about the agent's explicit expectation.

We introduce EDO, an expectation-conditioned residual representation for tool outputs. The main agent first writes a compact expected observation alongside its tool call, exposing a reference grounded in the preceding trajectory. The tool executes normally, and a smaller model compares its actual output with this reference to extract a semantic residual. The two components play complementary roles: the expectation exposes the action-contingent hypothesis, while the residual records how the observed result updates it. The extractor retains discrepancies and newly revealed evidence, together with the confirmation and local context needed to interpret them. For outputs routed through model-based extraction, subsequent reasoning uses the expected observation and residual in place of the raw output. The purpose of prediction is not to replace observation, but to make the observation's evidential relation to the prediction explicit. Contradicted assumptions must therefore be corrected, and unsupported expectations must not be treated as confirmed facts.

We evaluate EDO across six software-engineering and terminal-agent benchmarks. Among compressed methods, EDO achieves the highest observed task score on five of six benchmarks and the largest observed estimated cost saving on five. EDO achieves 98.97% of Raw's macro-averaged task score while using, on average, only 59.06% of its main-agent tokens and 60.79% of its estimated cost after benchmark-wise normalization. The strongest controlled comparison holds the expectation-generation protocol, inputs, models, and output cap fixed: on ProgramBench, EDO scores 26.59% versus 22.49% for the matched expectation-conditioned summary, a 4.10-point gain with a 95% confidence interval from 1.44 to 6.83. This advantage remains positive with a second solver and across three prompt paraphrases. Offline diagnostics clarify the representational difference: EDO raises correction recall from 70.00% to 94.55% and reduces unsupported expectations treated as facts from 21.88% to 12.50%, while overall fact recall remains close at 77.18% versus 79.19%. Together, these results identify the expectation–observation relation as a useful organizing target for compressed tool feedback: EDO preserves substantially more corrective evidence without retaining more facts overall.

Our main contributions are as follows:

- We distinguish fact coverage from the explicit communication of evidence updates in tool-feedback compression, emphasizing how observations relate to pre-execution expectations.
- We propose EDO, which pairs an explicit predictive reference with a semantic residual grounded in the actual tool output while keeping predicted content distinct from observed evidence.

- We demonstrate substantial inference savings across six agent benchmarks and higher ProgramBench scores than a matched expectation-conditioned summary under two solver configurations. Matched controls and evidence diagnostics characterize the value of the residual-oriented representation.

2 Related Work

2.1 Input and Tool-Output Compression

Prompt compression reduces language-model context through token selection, textual summarization, or learned compact representations. LLMLingua and LLMLingua-2 select tokens through iterative removal and distilled classification, respectively (Jiang et al., 2023; Pan et al., 2024); LongLLMLingua adds question relevance (Jiang et al., 2024). RECOMP trains extractive and abstractive compressors for retrieved documents (Xu et al., 2024a), while ICAE encodes context into learned memory slots (Ge et al., 2024). A parallel line compresses model-side state: SnapKV selects positions in the key–value cache (Li et al., 2024b), while Activation Beacon maps long sequences into compact activations (Zhang et al., 2025). These mechanisms reduce internal inference state rather than constructing a shorter textual record for later agent turns. In coding agents, SWE-Pruner uses an explicit goal hint and a learned skimmer to select relevant code lines (Wang et al., 2026). CoACT trains an observation compressor using whether the compressed result preserves the agent's next action (Chen et al., 2026). Deterministic code minification instead removes lexical redundancy from repository context (Hrubec & Cito, 2026), while tool-documentation compression shortens invocation instructions (Xu et al., 2024b). These methods emphasize structural redundancy, token importance, task relevance, or downstream behavior. In sequential tool use, another relevant property is whether the retained feedback communicates that an observation confirms or contradicts an assumption behind the current action; retaining many relevant facts alone does not guarantee that this correction is explicit.

2.2 Trajectory and Context Management for Agents

Agent context management addresses information accumulated over multiple reasoning and tool-use steps. AgentDiet analyzes trajectories for useless, redundant, and expired content and removes such material during inference (Xiao et al., 2026). Observation masking instead drops older observations using a simple recency policy, providing an inexpensive alternative that can match LLM-based summarization in software agents (Lindenbauer et al., 2025). ACON jointly compresses environment observations and interaction histories: it refines natural-language compression guidelines from cases in which full context succeeds but compressed context fails, then distills the optimized compressor into a smaller model (Kang et al., 2026). Other systems treat context as an evolving reasoning state rather than a passive log. ARC uses reflection to monitor this state and reorganize it when misalignment or degradation is detected (Yao et al., 2026). COMPASS separates tactical execution, strategic oversight, and a context manager that maintains concise progress briefs for different reasoning stages (Wan et al., 2026). Context as a Tool makes compression an explicit agent action over a structured workspace containing stable task semantics, condensed long-term memory, and recent high-fidelity interactions (Liu et al., 2026). SUPO periodically summarizes tool-use history and jointly optimizes the summarization strategy with tool-use behavior through reinforcement learning (Lu et al., 2026). These methods improve when, where, and how accumulated history is shortened or rewritten. Their central object, however, is the trajectory or working state as a whole; they do not explicitly organize each retained tool observation around whether it confirms or contradicts the action-specific expectation that preceded it. A compact long-term state can therefore still leave a locally decisive evidence update implicit.

3 Method

3.1 Problem Formulation and System Overview

We consider an LLM agent that solves a task through repeated tool interactions. Let $u_0 = (s, x)$ contain the system instruction s and task input x. At interaction t, the agent selects a tool call a_t from its visible history and receives a raw observation o_t. A conventional agent retains the complete

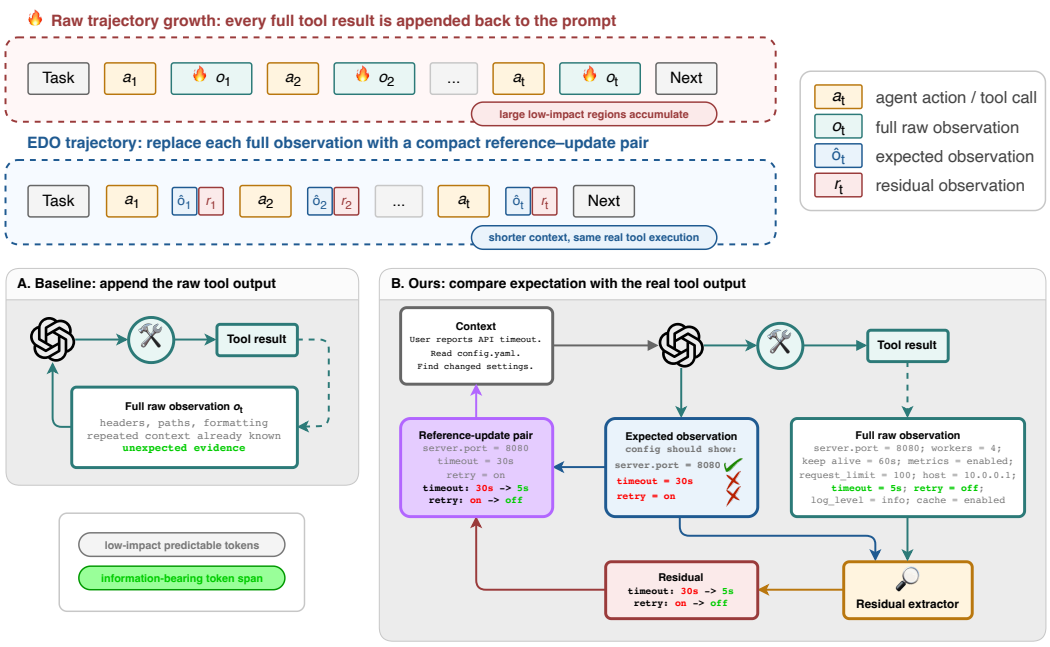

Figure 2: Conceptual residual branch of EDO: the agent predicts an observation before execution, and the extractor retains compact feedback relative to that reference.

sequence $\tau_t = (u_0, a_1, o_1, \ldots, a_{t-1}, o_{t-1})$. Consequently, a long return is processed again in every later model call even when only a small part remains relevant to the next decision.

Our objective is to replace a long observation with a shorter update while preserving both its factual content and its evidential relation to the agent's current hypothesis. Fact coverage concerns which observed statements remain available. Evidence updating concerns whether the retained feedback communicates what the result supports, contradicts, extends, or leaves unresolved. These properties are related but not interchangeable: a representation can retain many facts while omitting a decisive correction, or preserve a correction while losing the context needed to interpret it. Appendix F gives a minimal analysis of why high fact coverage alone cannot guarantee preservation of decision-relevant distinctions.

Accordingly, EDO targets decision-relevant distinctions rather than reconstruction of o_t. Two observations may overlap in most facts yet require different revisions when one confirms the premise of a_t and the other refutes it. Conversely, a compact update can avoid restating background already represented in context while retaining necessary observed confirmations and consequential changes. The resulting update is conditioned on the chosen action but does not prescribe the next one; the same solver remains responsible for continued execution. End-to-end task performance tests whether this shorter representation preserves the distinctions the solver needs.

EDO exposes the current hypothesis as an explicit pre-execution reference and expresses the subsequent feedback relative to it. The main agent emits an expected observation $\hat{o}_t$ together with the tool call. After the tool executes, the system derives compact feedback $\tilde{o}_t$ from the real return. The next solver-visible history is

$$\bar{\tau}_{t+1} = \bar{\tau}_t \parallel (a_t, \hat{o}_t, \tilde{o}_t),$$

where $\parallel$ denotes ordered message concatenation and $\bar{\tau}_1 = (u_0)$. The reference states what the agent anticipated before execution; the retained feedback states what the environment actually established afterward. Sections 3.2 and 3.3 define these two components, and Section 3.4 describes how they are integrated into the complete interaction.

3.2 PRE-EXECUTION REFERENCE

Conditioned on the visible history and selected call, the main model produces

$$\hat{o}_t = f_\theta(\bar{\tau}_t, a_t)$$

in the same assistant message as a_t, so producing the reference requires no additional solver round. Because the tool has not yet executed, $\hat{o}_t$ cannot depend on o_t. It describes the anticipated status and the key observable values, artifacts, or failures that expose the agent's current understanding of the pending interaction. Its purpose is to provide a compact, action-specific comparison point, not to reconstruct the future output in full.

The visible history supplies context for the pending call, but it need not state the anticipated outcome in a compact form. EDO makes this comparison point explicit before execution, allowing the extractor to relate the actual return to a stated prediction rather than infer that prediction solely from the accumulated trajectory. The reference introduces no additional environmental observation; it specifies what the incoming evidence should be compared against. The matched comparison in Section 4.3 keeps the expectation-generation protocol, conditioning fields, solver, compressor, and feedback-generation cap fixed while varying the requested feedback representation.

A useful expectation is specific enough to be checked by the pending call and selective enough to expose what matters for that action. In Figure 1, a goal hint could broadly request the implementation related to configuration loading. The expectation instead predicts that the search will locate a definition and uses of *parse_config*. The returned matches identify *load_config* and contain no *parse_config* match within the searched files. Both references can guide evidence selection, but only the latter states the concrete assumption that the result must confirm or correct.

Because the reference is produced after a_t is selected, it is tied to the exact invocation rather than the task in general. It states a small set of observable claims, such as execution status, expected values or artifacts, and important absences. Restating the task provides no concrete comparison, whereas predicting the full return duplicates low-value content. The fixed prompt therefore requests a concise, checkable reference without separate labels or supervision.

We treat $\hat{o}_t$ as a stated reference rather than an estimate of the agent's complete internal belief state. It must remain grounded in the visible history and pending action, but it need not be correct in every detail: a relevant error can expose a discrepancy that the feedback should communicate. Crucially, predicted content is not environmental evidence. The retained feedback must ground every factual claim in o_t, explicitly correct inaccurate expectations, and preserve important observations outside the reference. Information already stated in $\hat{o}_t$ may be shortened, but the feedback must still indicate whether it was confirmed; the expectation alone never establishes what occurred.

3.3 SEMANTIC RESIDUAL FEEDBACK

For observations routed to the model-based branch, the smaller model constructs

$$r_t = g_\phi(\bar{\tau}_t, a_t, \hat{o}_t, o_t), \qquad \tilde{o}_t = r_t.$$

The extractor receives the task, solver-visible compressed history, command, expectation, and current raw return. It produces natural-language feedback rather than a literal text diff. The extraction prompt asks it to preserve the meaning and local evidence of an update, including exact paths, values, errors, test names, and snippets when they affect later decisions. Lexical change alone is not necessarily informative, whereas a small change in a number, unit, status, identifier, or return code can reverse the action implied by the result. The residual is therefore a semantic update rather than the set of tokens that differ between two strings. Appendix A.3 provides the fixed prompt.

Figure 2 illustrates the representation. Suppose the expectation states that the server uses port 8080, the timeout is 30 seconds, and retry is enabled. The actual output confirms port 8080, reports a 5-second timeout, and disables retry. A suitable residual records the observed port, corrects the timeout from 30 seconds to 5 seconds, and states that retry is disabled rather than enabled. The confirmation is not redundant: the predicted port and the observed port have the same propositional content but different evidential status. Likewise, reporting *timeout=5s* preserves a fact, whereas relating it to the 30-second prediction explicitly identifies the belief that must be revised.

Table 1: Conceptual obligations for feedback relative to the expected observation.

Relation	Extraction obligation
Support	Compact observed confirmation; the prediction itself is not evidence.
Contradiction	Identify the contradicted expectation and retain the observed correction.
Extension	New task-relevant evidence outside the expectation, with necessary context.
Unresolved	Record when important expected facts are neither confirmed nor contradicted.

Table 1 summarizes four complementary obligations. Support communicates necessary observed confirmation; contradiction identifies the prediction that failed and retains the correction; extension preserves task-relevant evidence outside the expectation; and unresolved marks important expected facts that the observation did not settle. These relations may overlap. They define what the feedback must communicate, not four disjoint output classes or a mandatory structured format.

Together, these obligations induce an evidence-first organization: establish the observed outcome, relate checkable expected claims to the return, retain consequential evidence outside the reference, and mark important unsettled claims as unresolved. The extractor does not recommend the next action; that decision remains with the main agent. Natural language can combine overlapping relations without repeating unchanged formatting or unrelated output regions, keeping the representation structured by evidential role while accommodating heterogeneous tools.

A conditioned summary may receive the same task, history, expectation, and raw observation and may also preserve corrections. The distinction is the requested organization: the matched summary prioritizes and connects evidence by general task relevance, whereas the residual makes the expectation–observation relation its organizing structure. Section 4.3 tests the downstream value of this representational choice.

3.4 Routing, Context Construction, and Accounting

The full system uses three routes rather than applying model-based extraction to every return. Short or simple outputs pass through unchanged. Recognized file reads and ANSI/TUI outputs receive deterministic compaction. Other long outputs are processed by the residual extractor defined above. Figure 2 depicts only this third route; its trajectory labels do not describe outputs handled by passthrough or deterministic compaction. Routing avoids an auxiliary-model call when the full return is already short or a reliable rule is sufficient. Appendix Sections A.3 and A.4 provide the prompts and branch details.

The tool call and expected observation remain in the assistant message, while $\tilde{o}_t$ occupies the subsequent tool-response message. In the passthrough route, $\tilde{o}_t$ is the short return itself; in the deterministic route, it is the compacted return; and in the model-based route, it is r_t. Except for deliberate passthrough, the raw return is used only to construct $\tilde{o}_t$ and is not reinserted into later solver context. The main agent retains responsibility for choosing actions and producing $\hat{o}_t$. The smaller extractor sees the current raw return only while constructing r_t, and neither model receives a hidden copy of earlier raw observations.

The relevant compression unit is the complete representation visible to the next solver turn. For the model-based branch, its length is

$$L_t^{\mathrm{pair}} = \ell(\hat{o}_t \parallel r_t),$$

where $\ell(\cdot)$ counts tokens after $\hat{o}_t$ and r_t are concatenated with their separator and retokenized. The joint representation includes both the predictive reference and execution feedback. Expectation generation and subsequent rereading are included in main-agent usage, while residual extraction is included in auxiliary-model usage and total estimated cost. We therefore evaluate efficiency over complete agent trajectories, accounting for both components of the representation and the extraction overhead.

The residual generator has an output-token limit $L_{\max}^{\mathrm{res}}$ that excludes the separately generated expectation. Section 4.4 instead jointly caps the final expectation–feedback pair relative to the raw observation length, contrasting a fixed feedback-generation limit with a retained-representation budget; implementation details are in Appendix Sections A.2 and B.1.

Table 2: Task score across six benchmarks. Raw is uncompressed; among compressed methods, best is bold and second-best is underlined.

Benchmark	Raw	Generic	Task-Cond.	LLMLingua-2	Head-Tail	AgentDiet	EDO
SWE-bench Lite	49.00%	45.33%	43.33%	42.67%	44.00%	42.67%	**47.33%**
ProgramBench	29.41%	25.59%	24.40%	24.35%	14.53%	22.17%	**26.59%**
TerminalBench	52.33%	22.09%	26.74%	25.58%	23.26%	22.09%	**54.65%**
QuixBugs	77.50%	75.00%	78.75%	77.50%	**80.00%**	75.00%	75.00%
DevEval	22.73%	12.12%	10.61%	15.15%	9.09%	18.18%	**19.70%**
CompileBench	30.00%	30.00%	20.00%	15.00%	20.00%	25.00%	**35.00%**

Table 3: Main-agent token saving / estimated cost saving relative to Raw, in percent. Best is bold; second-best is underlined.

Benchmark	Generic	Task-Cond.	LLMLingua-2	Head-Tail	AgentDiet	EDO
SWE-bench Lite	18.57 / 16.48	14.69 / 14.36	22.12 / 20.85	24.91 / 23.29	23.11 / 20.55	**27.55 / 28.71**
ProgramBench	53.28 / 55.99	56.20 / 58.55	3.69 / 6.00	47.10 / 50.12	64.35 / 63.48	**69.02 / 68.72**
TerminalBench	-7.28 / -15.39	-9.27 / -21.25	-8.61 / -14.19	9.04 / -1.04	7.53 / -11.81	**41.43 / 34.13**
QuixBugs	-14.37 / -23.28	-9.50 / -17.61	-12.29 / -19.28	-15.19 / -22.01	-2.52 / -17.12	**25.93 / 33.19**
DevEval	14.16 / 12.02	10.95 / 8.27	-30.36 / -45.28	-12.57 / -26.33	0.48 / -14.83	**29.23 / 31.50**
CompileBench	46.69 / 31.23	39.17 / 26.24	**57.98 / 50.15**	40.65 / 34.62	28.52 / 14.44	52.46 / 39.04

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

We evaluate task performance and inference efficiency on SWE-bench Lite (Jimenez et al., 2024), ProgramBench (Yang et al., 2026), TerminalBench (Merrill et al., 2026), QuixBugs (Lin et al., 2017), DevEval (Li et al., 2024a), and CompileBench (Quesma, 2025). The main agent uses Qwen/Qwen3.7-flash, and model-based compression uses Qwen/Qwen3.5-4B. Each task is run three times. Raw retains tool outputs. The general compression baselines are Generic Summary, Task-Conditioned Summary, LLMLingua-2 (Pan et al., 2024), Head-Tail truncation, and AgentDiet (Xiao et al., 2026). Our key controls are Expected + Conditional Summary, Goal Hint + Evidence, and Recent-Observation Masking. They test whether the same conditioning signals, a non-predictive information need, or a simple recency policy can explain the result.

We report benchmark score, main-agent token saving, and estimated cost saving relative to Raw. The cross-benchmark task summary is the ratio of equal-weight macro-averaged scores; token and cost summaries average benchmark-wise usage ratios after normalization to Raw. Token saving includes expectation tokens but excludes auxiliary-model tokens, whereas estimated cost includes auxiliary calls. Cache-aware cost uses the observed solver-token categories and conservatively imputes the auxiliary-model price as one tenth of the corresponding solver price. Default runs use a fixed 1,024-token cap on model-generated compressed feedback. Joint-budget experiments instead cap each retained representation relative to its raw observation. Appendix A.1 defines benchmark scoring, Appendix Sections A.3 and A.4 give complete prompts and implementations, and Appendix D.1 gives the accounting protocol; the remaining supplementary sections report full statistical tests and robustness details.

4.2 END-TO-END QUALITY–COST TRADE-OFFS

We evaluate end-to-end task quality and inference efficiency across all six benchmarks. Tables 2–4 report task scores, savings, and the ProgramBench usage and latency decomposition. Among compressed methods, EDO achieves the highest observed score on five benchmarks and the largest observed estimated cost saving on five. It achieves 98.97% of Raw's macro-averaged task score while using, on average, only 59.06% of its main-agent tokens and 60.79% of its estimated cost after benchmark-wise normalization. Thus, EDO preserves nearly all of Raw's macro-averaged task score at substantially lower inference cost. These are system-level results for the complete routing pipeline; Section 4.3 separately isolates the requested feedback representation.

Table 4: ProgramBench usage, normalized cost, and latency per task; expectation tokens are included in main-model usage and shown separately for transparency.

Method	Main input K tokens	Main output K tokens	Compressor K tokens	Expectation K tokens	Est. cost % of Raw	Latency seconds
Raw	363.98	13.48	0	0	100.00	125.48
Generic	171.90	4.43	13.08	0	44.01	91.82
Task-Cond.	161.05	4.29	12.83	0	41.45	94.69
LLMLingua-2	351.83	11.70	0	0	94.00	124.44
Head-Tail	194.22	5.45	0	0	49.88	62.46
AgentDiet	129.47	5.09	12.35	0	36.52	63.52
EDO	113.56	3.37	41.92	0.60	31.28	44.36

Table 5: Default ProgramBench controls. Joint tokens combine reference and feedback; best is bold, second-best underlined; paired 95% CIs are unadjusted.

Method	Score	Mean joint tokens / obs.	Cost saving	EDO − baseline 95% CI
Expected + Conditional Summary	22.49%	484	<u>67.80%</u>	+4.10 [1.44, 6.83]
Goal Hint + Evidence	<u>24.96%</u>	<u>387</u>	<u>67.80%</u>	+1.63 [−1.04, 4.45]
Recent-Observation Masking	24.12%	682	52.40%	+2.47 [−0.38, 5.14]
EDO	**26.59%**	**286**	**68.72%**	–

ProgramBench makes this gain concrete. EDO scores 26.59%, higher than every general compression baseline, while its estimated cost is only 31.28% of Raw and is also the lowest among the compared methods. Main-agent input falls from 363.98K to 113.56K tokens per task, a 68.8% reduction, and measured end-to-end latency falls from 125.48 to 44.36 seconds, a 64.6% reduction. The estimated cost and measured latency reductions account for auxiliary extraction, which uses 41.92K tokens per task; the 0.60K expectation tokens are included within main-agent usage. The complete system therefore delivers lower estimated cost and measured latency after accounting for residual construction. A descriptive, completed-step-matched prefix comparison still shows 61.80% lower main-agent token use; Table 32 reports the full analysis.

4.3 Controlled Comparisons

We test whether EDO's advantage comes from explicit expectation access alone or from the feedback representation. The matched summary uses the same input fields and correction instruction; Goal Hint and Masking test a non-predictive information need and simple recency. Table 5 reports all three controls under the default ProgramBench configuration.

EDO attains the highest observed score and lowest estimated cost among the four methods. Its strongest controlled result is against Expected + Conditional Summary: with the same expectation-generation protocol, solver, compressor, inputs, and output cap, EDO improves score by 4.10 points, with a 95% confidence interval from 1.44 to 6.83, while shortening the observed mean joint representation from 484 to 286 tokens. Point estimates also favor EDO over Goal Hint + Evidence by 1.63 points and Masking by 2.47 points, although their intervals include zero. The matched-summary result shows that the requested compression target changes which evidence is selected, connected, and presented to the main agent.

4.4 Evidence and Budget Sensitivity

We examine evidence selection through offline replay of the same saved ProgramBench observations from EDO trajectories. Figure 3a and Figure 3b report the resulting diagnostics. Relative to the matched summary, EDO increases correction recall from 70.00% to 94.55% and reduces unsupported expectations asserted as facts from 21.88% to 12.50%. Overall fact recall remains close, at 77.18% for EDO and 79.19% for the matched summary. In this offline diagnostic, EDO's retention

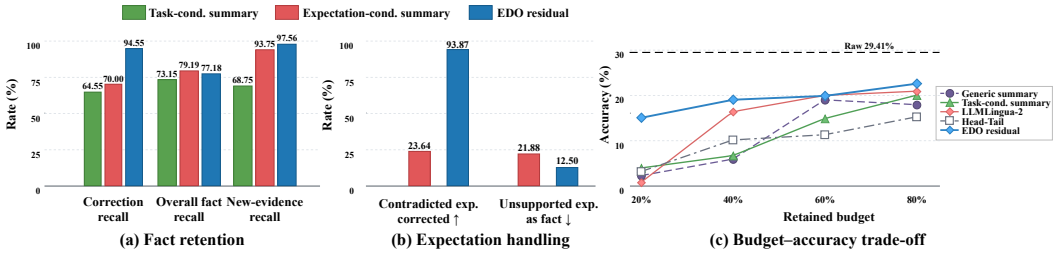

Figure 3: Evidence-retention diagnostics under offline replay and ProgramBench score under matched joint representation budgets.

Table 6: ProgramBench score under joint representation budgets. Best is bold; second-best is underlined.

Budget	Exp.+Cond. Summary	Goal Hint	Masking	EDO
20%	2.28%	<u>10.65%</u>	8.21%	**15.27%**
40%	13.41%	<u>17.17%</u>	16.52%	**19.25%**
60%	17.72%	19.25%	<u>19.61%</u>	**20.09%**
80%	20.84%	<u>21.74%</u>	19.34%	**22.79%**

profile therefore shifts toward corrective evidence and explicit handling of the predictive reference rather than greater overall fact coverage.

We next evaluate a deterministic joint-budget stress test, shown in Figure 3c and Table 6. EDO achieves the highest observed score at all four budgets. At the primary 20% operating point, where compression materially reduces observation context, its 15.27% score exceeds the matched summary, Goal Hint, and Masking by 12.99, 4.62, and 7.06 points; the latter paired gains have 95% confidence intervals of [2.06, 7.21] and [4.28, 9.87]. Because the hard cap applies to the serialized reference–feedback pair, this result jointly reflects evidence selection, information ordering, and truncation robustness, and shows that expectation-relative feedback retains more task utility in this aggressive-compression regime.

4.5 Robustness Analysis

Finally, we test the controlled comparison across a second solver, prompt paraphrases, expectation-quality conditions, and additional benchmarks. Across two Qwen solvers, EDO improves over the matched summary by 4.10 and 3.76 points. Under Qwen/Qwen3.8-27B, all three evaluated prompt paraphrases preserve the advantage, with gains from 3.36 to 4.18 points.

In the fixed-observation stress test, EDO explicitly corrects 92.18% of contradicted predictions even when the reference is relevant but partially incorrect. On the separate 80-task subset, replacing these references with irrelevant ones lowers task score by 3.56 points, highlighting the importance of action relevance. On TerminalBench, EDO leads the matched summary by 8.53 points, with a 95% confidence interval from 0.78 to 15.89. On DevEval, the estimated gain is 3.03 points, with a 95% confidence interval from -2.53 to 9.09. Appendix C analyzes failures from lost context and tight reference–feedback budgets.

5 Conclusion

We introduced EDO, an expectation-conditioned approach that represents tool feedback as an update to the agent's pre-execution expectation. Across six agent benchmarks, EDO preserves 98.97% of Raw's macro-averaged score while substantially reducing main-agent token usage and estimated cost. Controlled comparisons and evidence diagnostics suggest that explicitly organizing feedback around confirmations, corrections, and new evidence improves compression quality, particularly under tight context budgets. These results highlight expectation–observation relations as a useful principle for efficient agent context management.

Reproducibility Statement

We provide detailed benchmark versions, task counts, scoring definitions, model and runtime configurations, prompts, method and baseline implementations, token and cost accounting procedures, and statistical evaluation protocols in the appendix. We additionally report random seeds, the number of runs, complete benchmark-level results, controlled comparisons, sensitivity analyses, and diagnostic procedures. These details are intended to support independent reimplementation and verification of the reported findings.

AI Use Statement

Artificial intelligence tools were used to assist with post-writing language polishing and preliminary exploration of related work. AI assistance was also used to implement portions of the experimental code. The authors reviewed the AI-assisted code and take full responsibility for its correctness, the validity of the experiments, and the claims reported in this paper.

References

Haorui Chen, Yuancheng Zhu, Yitong Zhang, and Jia Li. CoACT: Action-preserving observation compression for coding agents. *arXiv preprint arXiv:2607.02911*, 2026. URL https://arxiv.org/abs/2607.02911.

Tao Ge, Hu Jing, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. In-context autoencoder for context compression in a large language model. In *International Conference on Learning Representations*, 2024.

Nicolas Hrubec and Jürgen Cito. Reducing token usage of state-in-context agents using minification. In *Proceedings of the 2026 34th IEEE/ACM International Conference on Program Comprehension*, pp. 537–546. Association for Computing Machinery, 2026.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMLingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 13358–13376. Association for Computational Linguistics, 2023.

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. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1658–1677. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.91. URL https://aclanthology.org/2024.acl-long.91/.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A. Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. ACON: Optimizing context compression for long-horizon LLM agents. In *International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2510.00615.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3603–3614, 2024a. doi: 10.18653/v1/2024.findings-acl.214. URL https://aclanthology.org/2024.findings-acl.214/.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM knows what you are looking for before generation. In *Advances in Neural Information Processing Systems*, 2024b.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey challenge. In *Companion Proceedings of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pp. 55–56. Association for Computing Machinery, 2017. doi: 10.1145/3135932.3135941.

Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. The complexity trap: Simple observation masking is as efficient as LLM summarization for agent context management. In *NeurIPS 2025 Workshop on Deep Learning for Code*, 2025. URL https://arxiv.org/abs/2508.21433.

Shukai Liu, Bo Jiang, Jian Yang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon SWE-agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 20604–20617, San Diego, California, United States, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.findings-acl.1032. URL https://aclanthology.org/2026.findings-acl.1032/.

Miao Lu, Weiwei Sun, Weihua Du, Zhan Ling, Xuesong Yao, Kang Liu, and Jiecao Chen. Beyond the context window: Scaling agentic RL via end-to-end optimized context compression. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, pp. 21074–21125. Association for Computational Linguistics, 2026.

Mike A. Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Jenia Jitsev, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, 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. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 963–981. Association for Computational Linguistics, 2024.

Quesma. CompileBench. https://github.com/QuesmaOrg/CompileBench, 2025. Benchmark website and GitHub repository.

Guangya Wan, Mingyang Ling, Xiaoqi Ren, Rujun Han, Sheng Li, and Zizhao Zhang. COMPASS: Enhancing agent long-horizon reasoning with evolving context. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, pp. 3360–3380. Association for Computational Linguistics, 2026.

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. *arXiv preprint arXiv:2601.16746*, 2026. URL https://arxiv.org/abs/2601.16746.

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. Reducing cost of LLM agents with trajectory reduction. *Proceedings of the ACM on Software Engineering*, 3(FSE), 2026.

Fangyuan Xu, Weijia Shi, and Eunsol Choi. RECOMP: Improving retrieval-augmented LMs with context compression and selective augmentation. In *International Conference on Learning Representations*, 2024a.

Yang Xu, Yunlong Feng, Honglin Mu, Yutai Hou, Yitong Li, Xinghao Wang, Wanjun Zhong, Zhongyang Li, Dandan Tu, Qingfu Zhu, Min Zhang, and Wanxiang Che. Concise and precise context compression for tool-using language models. *arXiv preprint arXiv:2407.02043*, 2024b.

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

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Program-Bench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026. URL https://arxiv.org/abs/2605.03546.

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

Yilun Yao, Shan Huang, Elsie Dai, Zhewen Tan, Zhenyu Duan, Shousheng Jia, Yanbing Jiang, and Tong Yang. ARC: Active and reflection-driven context management for long-horizon information seeking agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 18644–18659. Association for Computational Linguistics, 2026.

Peitian Zhang, Zheng Liu, Shitao Xiao, Ninglu Shao, Qiwei Ye, and Zhicheng Dou. Long context compression with activation beacon. In *International Conference on Learning Representations*, 2025.

SUPPLEMENTARY IMPLEMENTATION AND EVALUATION DETAILS

A EXPERIMENTAL SETUP AND IMPLEMENTATIONS

A.1 BENCHMARK DATA AND SCORING DEFINITIONS

Benchmark	Version	Complete task count	Brief description
SWE-bench Lite	Official SWE-bench Lite split	300	GitHub issue resolution tasks from real Python repositories.
ProgramBench	ProgramBench local package snapshot	200	Repository-level programming tasks scored by official test suites; the bundled fixture is excluded.
TerminalBench	terminal-bench-core-0.1.1	86	Interactive terminal tasks requiring command-line investigation, file edits, and validation.
QuixBugs	Official QuixBugs Python + Java benchmark	80	Algorithmic bug-fixing tasks counted by language-specific task id.
DevEval	terminal-bench-datasets DevEval snapshot	66	Software development tasks with repository context and evaluator-defined tests.
CompileBench	CompileBench snapshot	20	Compilation and build-repair tasks scored by successful build/test verification.

All scores are computed on the complete benchmark listed for each row. No additional split, random subset, or method-specific task exclusion is applied. All methods are evaluated on the same task set within each benchmark. For binary benchmarks, the reported score is the pooled solved or resolved fraction over all task-runs across the complete task set and three runs; ProgramBench uses the graded score-rate definition below.

Timeouts, invalid answers, tool failures, and model-interface failures are counted as zero for the affected task or test under the same rule for all methods. Each method uses the same environment and evaluation script for a given benchmark.

A.1.1 PROGRAMBENCH SCORE AND TASK-LEVEL SUCCESS

ProgramBench reports a graded score rather than a binary resolved/not-resolved outcome. The ProgramBench scores in the main tables are computed as the total achieved score over the total available score across all evaluated task-runs:

$$\text{ProgramBench Score} = \frac{\sum_i \text{score}_i}{\sum_i \text{max_score}_i} \times 100,$$

where i ranges over all $200 \times 3 = 600$ ProgramBench task-runs. This score is not a complete-program resolved rate and is not a macro-average of per-task test-pass rates. Because each run uses the same 200 tasks and the same per-task maximum scores, this pooled task-run score rate is equivalent to averaging the three run-level ProgramBench score rates reported in Appendix D.2.

The task-level success/failure groupings used in the trajectory analysis are defined separately from the graded ProgramBench score. For those groupings, each task-run is first mapped to a binary status:

$$\text{success}_i = \begin{cases} 1, & \text{score}_i > 0, \\ 0, & \text{score}_i = 0. \end{cases}$$

Thus a task-run with any positive ProgramBench score is treated as successful for the binary grouping, whereas compile failures, runtime errors, timeouts, dependency failures, and other zero-score

outcomes are treated as failures. If a run reaches the step limit but still obtains a positive score, it is counted as successful for this binary grouping; if it obtains zero score, it is counted as failed.

For each method and task, the three binary run-level statuses are then aggregated by majority vote:

$$\text{task_success} = 1 \quad \text{if} \quad \text{success}_1 + \text{success}_2 + \text{success}_3 \geq 2,$$

and $\text{task_success} = 0$ otherwise. Under this definition, Raw has 59 / 200 task-level successes and EDO has 53 / 200 task-level successes. These counts are close to, but not identical with, the main ProgramBench score rates of 29.41% and 26.59%, because the former are task-level positive-score majority outcomes and the latter are pooled graded score rates over all task-runs.

A.2 Common Model and Runtime Settings

The main solver model is `Qwen/Qwen3.7-flash` as used through the OpenAI-compatible interface. The compression model used by EDO, Generic Summary, Task-Conditioned Summary, Expected + Conditional Summary, and AgentDiet is `Qwen/Qwen3.5-4B`.

The reported final implementation uses temperature-zero decoding:

Setting	Value
Main model temperature	0
Main model max output tokens	4,096
Compression model temperature	0
Non-budget model-generated compression max output tokens	1,024 for Generic Summary, Task-Conditioned Summary, Expected + Conditional Summary, EDO residual extraction, and AgentDiet
Thinking mode	disabled
Runs per task	3
Reported value	average across the 3 runs

Throughout the main text and appendix, quantities labeled "per task" are obtained by first averaging the three independent task-runs for each task and then aggregating across tasks. This convention applies to token usage, cost, latency, and other reported per-task averages. Statements explicitly referring to all 600 task-runs, run-level outcomes, or task-run totals retain task-run as their statistical unit.

The agent framework is mini-swe-agent for SWE-bench Lite and ProgramBench, with benchmark-specific wrappers for TerminalBench-style and CompileBench-style tasks. The available tools are the standard benchmark tools: shell execution, file read/write, patching or source generation, compilation, and benchmark test/evaluation commands.

The 1,024-token model-generated compression cap applies to non-budget compressed representations. The 20%, 40%, 60%, and 80% joint-budget experiments instead apply the retained-representation budgets described in Appendix B.1; they do not apply the fixed 1,024-token generation cap.

The final settings use a step limit of 30 for every benchmark together with benchmark-level timeout guards. ProgramBench wraps potentially non-terminating executable probes with a short timeout. Evaluation timeouts are applied consistently to all methods within the same benchmark.

Each benchmark uses a 120-second tool timeout and a 900-second task timeout. Model-interface errors are retried at most twice, with backoff intervals of 1 and 4 seconds. Read-only tool calls are retried once after a timeout; side-effecting actions such as file writes, installation, and compilation are not replayed automatically. The three runs use seeds 1, 2, and 3. These temperature-zero settings apply to all benchmark evaluations; the separate Figure 1 action-distribution estimate uses the sampling configuration reported in Appendix C.2.

A.3 EDO Execution Flow and Key Prompts

A.3.1 Expected Observation

The expected observation is generated in the same main-model response that contains the tool command. It is generated before executing the tool. At that moment, the model can see the task instruction, system instruction, prior trajectory, and the command it is about to run, but it cannot see the future tool output.

The key prompt asks the model to provide a concise predictive reference for interpreting the subsequent tool result:

```
Before executing the command, write a concise expected
observation that will
serve as a reference for comparison with the actual result.
Describe the
expected execution status and the key observable evidence,
values, or artifacts
likely to appear.  Keep the prediction specific enough that
the subsequent
observation can be represented through confirmations,
contradictions, and new
evidence relative to this reference.
```

For sequential tool use, the expected observation is paired with the command in the same assistant message. Parallel tool calls are not used in the reported experiments.

A.3.2 Residual Extraction

After the tool returns, the compression model receives the task context, the solver-visible compressed history, the command, the expected observation, and the raw tool output. It returns a compact residual that focuses on the delta between the expected observation and the actual tool output.

The prompt is:

```
[SYSTEM]
Given the task, command, expected observation, and raw tool
output, construct a
compact delta residual observation.  Keep the task-relevant
difference between
what was expected and what was actually observed.  Preserve
exact identifiers,
paths, errors, test names, and values when they matter.
Preserve compact
confirmation when the raw output establishes that an
expected task-relevant
outcome actually occurred; prediction alone is not evidence.
The expected observation
is a prediction rather than evidence; preserve relevant
contradictions and
unresolved expectations without adding next-action advice.

[USER]
TASK: {task}
VISIBLE_HISTORY: {solver_visible_history}
COMMAND: {command}
EXPECTED_OBSERVATION: {expected_observation}
RAW_OUTPUT: {raw_output}
MAX_TOKENS: {residual_cap}
```

The default EDO output is evidence-focused. It is designed to preserve task-relevant differences between expected and observed behavior. It does not change the task, the tool, or the benchmark evaluator.

A.3.3 Compression and History Update

Each tool observation is processed before the next solver turn. Short or simple outputs may be preserved compactly, while longer observations are converted into residual observations. The raw tool output is stored for accounting and audit, but the next solver turn receives the expected observation plus the compact residual rather than the full raw output.

The visible next-turn message contains the executed command status and the compressed evidence. The previous complete raw output is not placed back into the main model context.

A.4 Baseline Implementation and Fairness

- **Raw:** the solver receives the full tool output, subject only to the same benchmark/tool framework limits.
- **Generic Summary:** each tool output is summarized without task-specific conditioning beyond the output itself.
- **Task-Conditioned Summary:** each tool output is summarized using the task text and prior trajectory summary to rank which visible facts are most useful.
- **Expected + Conditional Summary:** the solver uses the same expectation-generation protocol as EDO, but the auxiliary model produces a task-conditioned summary rather than a residual.
- **LLMLingua-2:** the tool output is compressed by the LLMLingua-2 compressor under the target budget.
- **Head-Tail:** the output is truncated by keeping the head and tail portions under the target budget.
- **AgentDiet:** prior trajectory content is periodically reflected and reduced by the same small model family used for the other auxiliary compression methods.
- **Goal Hint + Evidence Compression:** before each tool call, the solver states the concrete information needed from the result without predicting that result. The auxiliary model retains factual evidence that answers this information need and any additional task-relevant evidence that changes the available decision.
- **Recent-Observation Masking:** the current observation is retained verbatim up to the applicable limit or reduced by deterministic 3:1 head–tail truncation. Before later solver turns, each earlier observation is replaced by a fixed mask marker; no auxiliary model is used.

Short or simple outputs are handled by a shared passthrough rule before model-based compression. Empty outputs are recorded as empty observations; short TerminalBench-style outputs below roughly 900 characters are retained directly; file-read outputs use deterministic compaction up to roughly 4,500 characters. For ProgramBench and mini-swe-agent paths, summary passthrough uses thresholds of 1,800 characters without task context and 2,200 characters with task context. ANSI/TUI-like outputs are compacted to at most 20 lines and 12 visible examples. LLMLingua-2 uses the same passthrough rule before invoking the compressor.

The Generic Summary prompt summarizes only the raw output and preserves concrete paths, symbols, errors, tests, return codes, values, and artifacts without using task context or adding advice. Task-Conditioned Summary additionally uses the task and solver-visible history to prioritize facts that may affect later decisions. Expected + Conditional Summary may use the task, visible history, command, and expected observation to decide relevance, but it is instructed to report actual evidence rather than organize the result as a residual, to correct contradicted expectations explicitly, and not to treat unverified expectations as facts.

LLMLingua-2 uses the `PromptCompressor` with `microsoft/llmlingua-2-bert-base-multilingual-cased-meetingbank` and

`use_llmlingua2=True`. In the joint representation-budget experiments, its retention rate is set to the target budget ratio, so the 20%, 40%, 60%, and 80% settings use rates 0.20, 0.40, 0.60, and 0.80, respectively. The default main experiments use the same compressor implementation but are not part of this retained-budget sweep. Very long inputs are truncated by preserving both head and tail spans before compression, and all outputs are subsequently subject to the same observation-budget hard cap used by the other budgeted baselines.

AgentDiet uses `Qwen/Qwen3.5-4B` as the reflection model with temperature 0 and thinking disabled. Its main parameters are lag $a = 2$, prefix context width $b = 1$, trigger threshold $\theta = 500$ tokens, and a 1,024-token reflection output cap. After each new assistant/tool step, it attempts to compress the complete step two steps behind the current position. Steps at or below 500 tokens are skipped; failed or structurally invalid reflections leave the original trajectory unchanged; replacements are accepted only when they reduce the trajectory by more than 500 tokens.

Across methods, we hold the task set, solver model, decoding temperature, environment, run count, and evaluator fixed. EDO and its expectation-based controls additionally modify the solver prompt to elicit a pre-execution expectation; compression prompts and budget rules vary by method.

Goal Hint and EDO differ in both their pre-execution reference and their auxiliary compression objective, so their comparison is a complete method comparison rather than a single-factor ablation. Recent-Observation Masking changes current-observation retention, historical-observation visibility, and auxiliary-model usage simultaneously and is likewise treated as a complete baseline. The non-observation history span is unchanged across these controls.

EDO adds the expected-observation requirement to the solver prompt. This is part of the method definition and is applied consistently to EDO variants.

B Controlled Comparisons and Ablations

B.1 Joint Budget Experiment

The 20%, 40%, 60%, and 80% budgets are defined relative to the original raw tool-output token count for each observation. For a target ratio ρ and a raw observation of L tokens, the retained-representation budget is $B = \lfloor \rho L \rfloor$ tokens. These budget experiments do not use the fixed 1,024-token model-generation cap used in the non-budget experiments. The tokenizer is the same-snapshot `tokenizer.json` used for the corresponding main-model interface. After a method generates its retained representation, the result is tokenized again. If it exceeds B, the excess suffix is removed before the representation is placed in the next solver context. Thus, the prompt-level length request is followed by a deterministic budget cap, and the final retained representation is allowed to occupy the full B-token budget but cannot exceed it. At the 20% setting, for example, every over-budget representation is forcibly truncated to at most $\lfloor 0.2L \rfloor$ tokens.

For EDO, the same hard limit covers the expected-observation and residual representation jointly as they are visible to the next solver turn. Since the expected observation is generated before the actual output length is known, the residual receives only the budget remaining after the expectation is tokenized. The final concatenated expectation–residual representation is checked once more and truncated to B tokens if necessary. If the expectation alone reaches the limit, no additional residual tokens are retained.

For Expected + Conditional Summary, the same hard limit jointly covers the expected observation and summary. The summary receives only the budget remaining after the expectation is tokenized, and the final concatenated representation is checked against B before entering the next solver context.

For Goal Hint + Evidence Compression, the information need and retained evidence jointly share B. Information-need tokens are main-model generations: they are counted as main-model output when generated and as main-model input when read in later turns. For Recent-Observation Masking, B applies to the current observation; over-budget observations retain a deterministic 3:1 head–tail split, while each older observation is replaced by `[PREVIOUS OBSERVATION MASKED]`. Mask-marker tokens are counted whenever they enter a later main-model prompt.

Table 9: ProgramBench score under matched joint representation budgets. Values are means over three runs.

Budget	EDO	Goal Hint + Evidence	Recent-Observation Masking	Expected + Conditional Summary
20%	15.27%	10.65%	8.21%	2.28%
40%	19.25%	17.17%	16.52%	13.41%
60%	20.09%	19.25%	19.61%	17.72%
80%	22.79%	21.74%	19.34%	20.84%

Table 10: Paired ProgramBench differences under joint representation budgets.

Budget	Comparison	Difference	95% CI	Two-sided p
20%	EDO – Goal Hint	+4.62	[2.06, 7.21]	< 0.001
20%	EDO – Masking	+7.06	[4.28, 9.87]	< 0.001
40%	EDO – Goal Hint	+2.08	[-0.41, 4.61]	0.102
40%	EDO – Masking	+2.73	[0.18, 5.35]	0.036
60%	EDO – Goal Hint	+0.84	[-1.58, 3.27]	0.492
60%	EDO – Masking	+0.48	[-1.96, 2.91]	0.704
80%	EDO – Goal Hint	+1.05	[-1.47, 3.58]	0.414
80%	EDO – Masking	+3.45	[0.86, 6.03]	0.009

For Generic Summary and Task-Conditioned Summary, the prompt asks the model to retain approximately the target ratio of the raw tool output, and the generated summary is subsequently subject to the same B-token budget cap rather than the non-budget 1,024-token cap. For Head-Tail, the budget is split evenly between the beginning and end of the output. For LLMLingua-2, the compressor is run under the same target retention level, followed by the same post-generation budget cap.

The budget experiment uses the same task collection, scoring rule, solver model, and evaluation environment as the corresponding main ProgramBench experiment. In compact tables, Exp.+Cond. Summary abbreviates Expected + Conditional Summary, and Task-Cond. abbreviates Task-Conditioned Summary. Figure 3(c) shows the Raw score of 29.41% as a horizontal dashed reference line rather than connecting each compression method to a shared 100% point.

Expected + Conditional Summary has been run under the same ProgramBench priority-control setting. The 80% retained-budget setting is not the default Expected + Conditional Summary configuration: it follows the budget-sweep prompt and deterministic retained-representation cap rather than the non-budget 1,024-token summary setting. Therefore, its score is not expected to match the default component-ablation score.

To distinguish residual feedback from alternative reference and history-management mechanisms, we evaluate Expected + Conditional Summary, Goal Hint + Evidence Compression, and Recent-Observation Masking under the same joint representation budgets. As shown in Table 9, the clearest separation occurs at the tightest budget; at higher budgets, the point estimates are more closely grouped.

Paired inference uses the ProgramBench task as the bootstrap cluster. Each resampled task carries all three runs under every compared method, and the pooled ProgramBench score is recomputed before forming each difference. We use 10,000 resamples, percentile 95% confidence intervals, and two-sided null-centered p-values with finite-resampling correction. All p-values are unadjusted exploratory estimates.

Both comparisons are significant at the 20% budget. The 40% and 80% comparisons against Masking are nominally significant before multiple-comparison correction, while the remaining intervals include zero. Thus, the paired results localize the clearest advantage of EDO to the tightest evaluated budget rather than establishing uniform superiority across settings.

We also report trajectory-level main-agent usage and estimated cost savings for each budget setting in Table 11. Across the four budgets, EDO reduces main-agent token usage by 51.80%–80.05%. Its no-cache cost savings range from 47.58% to 73.34%, while its cache-aware cost savings range from 47.29% to 73.14%. No-cache estimates were not retained for the two additional controls and are marked by dashes; all headline cost comparisons use the cache-aware values. The retained-

representation budget and trajectory-level token saving measure different quantities: the former limits each retained observation, whereas the latter reflects the resulting complete solver trajectory.

Table 11: ProgramBench efficiency under joint representation budgets; savings are relative to Raw.

Budget	Method	Main tokens (K/task)	Token saving	No-cache saving	Cache-aware saving
20%	EDO	75.29	80.05%	73.34%	73.14%
20%	Exp.+Cond. Summary	79.98	78.81%	71.63%	70.28%
20%	Goal Hint + Evidence	77.58	79.45%	–	71.70%
20%	Recent-Obs. Masking	78.47	79.21%	–	66.80%
20%	Head-Tail	83.44	77.89%	76.90%	62.55%
20%	LLMLingua-2	91.00	75.89%	75.13%	58.39%
20%	Task-Conditioned Summary	85.58	77.33%	71.64%	33.23%
40%	EDO	110.84	70.64%	64.76%	64.52%
40%	Exp.+Cond. Summary	118.32	68.65%	61.08%	59.61%
40%	Goal Hint + Evidence	115.20	69.48%	–	62.00%
40%	Recent-Obs. Masking	115.73	69.34%	–	58.20%
40%	Head-Tail	84.86	77.52%	76.78%	66.57%
40%	LLMLingua-2	94.22	75.04%	73.95%	53.06%
40%	Task-Conditioned Summary	99.54	73.63%	62.03%	30.04%
60%	EDO	146.70	61.14%	56.02%	55.76%
60%	Exp.+Cond. Summary	156.44	58.56%	50.42%	48.70%
60%	Goal Hint + Evidence	152.32	59.65%	–	52.10%
60%	Recent-Obs. Masking	154.68	59.02%	–	49.40%
60%	Head-Tail	92.41	75.52%	74.43%	56.53%
60%	LLMLingua-2	100.77	73.30%	71.88%	40.34%
60%	Task-Conditioned Summary	111.06	70.58%	63.85%	33.23%
80%	EDO	181.95	51.80%	47.58%	47.29%
80%	Exp.+Cond. Summary	194.20	48.55%	39.71%	37.94%
80%	Goal Hint + Evidence	189.78	49.72%	–	42.20%
80%	Recent-Obs. Masking	191.82	49.18%	–	40.80%
80%	Head-Tail	113.02	70.06%	70.67%	53.54%
80%	LLMLingua-2	104.77	72.24%	71.11%	34.33%
80%	Task-Conditioned Summary	102.81	72.76%	68.03%	40.77%
100%	Raw	377.46	0.00%	0.00%	0.00%

Table 12 reports observed retention and truncation statistics for the expectation-conditioned and alternative controls. Actual retention is the observed reference-plus-feedback length divided by the corresponding raw-observation length. Dynamic budget-cap hit denotes that model generation reached the observation-specific B-token ceiling; it does not apply to deterministic Masking. Post-generation hard truncation records representations that exceeded the retained budget and were reduced before entering the solver context.

Table 12: Observed retention and truncation under ProgramBench joint representation budgets.

Method	Budget	Actual retention	Reference	Feedback	Total retained	Dynamic cap hit	Hard truncation	Reference fills budget
Exp.+Cond. Summary	20%	19.6%	63	490	553	47.5%	42.8%	2.8%
Exp.+Cond. Summary	40%	39.1%	64	1,039	1,103	39.8%	35.2%	0.5%
Exp.+Cond. Summary	60%	58.5%	64	1,586	1,650	30.6%	26.3%	0.2%
Exp.+Cond. Summary	80%	77.2%	65	2,112	2,177	21.4%	18.2%	0.0%
EDO	20%	19.8%	64	494	558	61.2%	55.7%	2.8%
EDO	40%	39.5%	64	1,050	1,114	53.9%	48.3%	0.5%
EDO	60%	59.1%	64	1,603	1,667	45.1%	39.7%	0.2%
EDO	80%	78.4%	65	2,146	2,211	36.8%	31.2%	0.0%
Goal Hint + Evidence	20%	19.5%	49	501	550	54.3%	48.7%	1.8%
Goal Hint + Evidence	40%	39.0%	49	1,051	1,100	46.8%	41.5%	0.3%
Goal Hint + Evidence	60%	58.2%	50	1,591	1,641	38.1%	33.2%	0.1%
Goal Hint + Evidence	80%	76.9%	50	2,119	2,169	29.6%	25.4%	0.0%
Recent-Obs. Masking	20%	19.9%	0	561	561	–	64.8%	0.0%
Recent-Obs. Masking	40%	39.7%	0	1,120	1,120	–	58.1%	0.0%
Recent-Obs. Masking	60%	59.5%	0	1,678	1,678	–	49.6%	0.0%
Recent-Obs. Masking	80%	79.1%	0	2,231	2,231	–	41.2%	0.0%

B.2 Component Ablation and Default Configuration

B.2.1 Component-Table Variants

- **EDO:** the solver generates an expected observation before the tool call; the auxiliary model receives the expectation and raw output and returns a residual observation.

Table 13: ProgramBench component accounting; savings are relative to Raw.

Method	Score	Main tokens (K/task)	Token saving	No-cache saving	Cache-aware saving
EDO	26.59%	116.93	69.02%	69.45%	68.72%
Expected + Conditional Summary (relevance-organized)	22.49%	119.45	68.35%	67.95%	67.80%
Task-Conditioned Summary	24.40%	165.34	56.20%	58.55%	58.55%
w/o expected observation	14.44%	95.02	74.83%	73.43%	70.65%
Expected + Raw	29.27%	385.46	-2.12%	-2.35%	-2.71%
Raw	29.41%	377.46	0.00%	0.00%	0.00%

Table 14: Default ProgramBench controls and retained-representation lengths.

Method	Score	Main tokens (K/task)	Token saving	Cost saving	Reference	Feedback	Joint	Mean/median steps	Tool calls	Final context
EDO	26.59%	116.93	69.02%	68.72%	64	222	286	12.3 / 11	9.4	14.1K
Goal Hint + Evidence	24.96%	118.90	68.50%	67.80%	49	338	387	12.6 / 12	9.6	14.9K
Exp.+Cond. Summary	22.49%	119.45	68.35%	67.80%	64	420	484	12.8 / 12	9.7	15.2K
Recent-Obs. Masking	24.12%	148.25	60.72%	52.40%	0	682	682	13.9 / 13	10.6	18.6K

- **Expected + Conditional Summary (relevance-organized):** the solver generates the same type of expected observation, but the auxiliary model returns a relevance-organized task-conditioned summary rather than an explicit residual.
- **Expected + Raw:** the solver generates expected observations, but raw tool outputs remain visible to the solver.
- **w/o expected observation:** the residual/summary side does not use the expected-observation signal.

EDO and the relevance-organized Expected + Conditional Summary use the same task set, solver, and run settings. Their main difference is the target representation: EDO constructs a semantic residual, while Expected + Conditional Summary constructs a direct summary.

To isolate the contributions of explicit expectations and residual-oriented compression, Table 13 reports component scores and accounting. Full EDO achieves a 26.59% ProgramBench score with 69.02% main-token saving. Replacing the residual with the relevance-organized expectation-conditioned summary reduces the score to 22.49%, while removing the explicit expectation reduces it to 14.44%. Expected + Raw reaches 29.27%, close to Raw at 29.41%, but introduces additional token and cost overhead.

B.2.2 Goal Hint and Recent-Observation Controls

Goal Hint + Evidence Compression asks the main solver to state a compact `INFORMATION_NEED` before each tool call without predicting the result. The auxiliary model receives the task, visible history, command, information need, and raw output, and retains factual evidence that answers the stated need or otherwise changes the available decision. Recent-Observation Masking instead keeps the current observation visible for the next turn, deterministically applies a 3:1 head–tail reduction above 1,024 tokens, and replaces every older observation by `[PREVIOUS OBSERVATION MASKED]`; it does not call an auxiliary model.

Information-need and expected-observation tokens are counted as main-model output when generated and as main-model input when read in later turns. Auxiliary evidence, summaries, and residuals are counted as auxiliary output when generated and as main input when placed in subsequent solver context. Fixed mask markers generate no model tokens but are included in later main-model input accounting.

Paired inference resamples ProgramBench tasks as clusters, preserving all three runs under every compared method. Each of 10,000 resamples recomputes the pooled ProgramBench score before forming the paired difference. We report percentile 95% confidence intervals and two-sided null-centered p-values with finite-resampling correction. All p-values are unadjusted.

The EDO point estimate is higher than both additional controls under the default configuration, but both EDO confidence intervals include zero. Goal Hint exceeds Expected + Conditional Summary at nominal unadjusted significance. Because Goal Hint changes both the pre-execution reference and the auxiliary compression objective, this comparison does not isolate either factor causally.

Table 15: Default-configuration paired comparisons on ProgramBench. Differences are in percentage points.

Comparison	Difference	95% CI	Two-sided p
EDO – Goal Hint + Evidence	+1.63	[-1.04, 4.45]	0.236
EDO – Recent-Obs. Masking	+2.47	[-0.38, 5.14]	0.091
Goal Hint + Evidence – Exp.+Cond. Summary	+2.47	[0.16, 4.81]	0.037

Table 16: Cache-aware ProgramBench accounting for Goal Hint and Masking controls.

Method	Main regular	Main create	Main hit	Main output	Aux. input	Aux. output	Cost
Raw	30.00	70.00	263.98	13.48	0	0	0.05574
EDO	15.00	20.44	78.12	3.37	40.91	1.01	0.01743
Goal Hint + Evidence	15.50	21.80	78.20	3.40	40.00	1.20	0.01795
Recent-Obs. Masking	19.00	29.00	93.44	6.81	0	0	0.02653

Across 600 task-runs, normal termination occurs for 90.0% of EDO runs, 88.8% of Goal Hint runs, 86.5% of Expected + Conditional Summary runs, and 86.0% of Masking runs.

Table 17: Two Expected + Conditional Summary configurations on ProgramBench.

Configuration	Score	Token saving	Cost saving	Main tokens	Exp. length	Summary length	Joint length	Mean/median steps	Tool calls	Final context	Normal end
Relevance-organized (main component table)	22.49%	68.35%	67.80%	119.45K	64	420	484	12.8 / 12	9.7	15.2K	86.5%
Evidence-only (output-constraint figure)	20.14%	59.35%	58.22%	153.44K	64	648	712	15.1 / 14	11.5	20.7K	83.3%

B.2.3 TWO EXPECTED + CONDITIONAL SUMMARY CONFIGURATIONS

The expectation-conditioned summary in Table 5 and the evidence-only summary in Figure 4 use the same task set, solver and auxiliary models, temperature-zero decoding, visible task/history/command/expectation/raw-output fields, 1,024-token non-budget output cap, 30-step limit, timeouts, three-run protocol, passthrough rules, deterministic file compaction, ANSI/TUI compaction, and failure fallback. We refer to them as the *relevance-organized* and *evidence-only* configurations, respectively. Under the documented implementation, their only intended configuration difference is the final output constraint.

Both configurations share the following core instruction:

```
You compress a tool observation for a coding agent.  Use
TASK,
VISIBLE_HISTORY, COMMAND, and EXPECTED_OBSERVATION to
determine which evidence
is relevant to the current task.  Report only information
supported by
RAW_OUTPUT. Preserve exact file paths, symbols, commands,
return codes, error
messages, test names, values, and generated artifacts when
they may affect later
decisions.  Explicitly correct EXPECTED_OBSERVATION when
RAW_OUTPUT contradicts
it.  Do not treat an unverified expectation as an observed
fact.  Do not add a
new action plan or next-step advice.
```

The relevance-organized configuration then adds:

```
Return a concise task-conditioned summary.  You may group,
prioritize, and
briefly connect evidence according to its relevance to TASK.
```

The evidence-only configuration instead adds:

```
Return factual evidence from RAW_OUTPUT only.  Do not
include rationale,
interpretation, relevance explanations, or advice.
```

Both use the same user message fields: `TASK`, `VISIBLE_HISTORY`, `COMMAND`, `EXPECTED_OBSERVATION`, `RAW_OUTPUT`, and `MAX_TOKENS`. The shared ProgramBench passthrough threshold is 2,200 characters when task context is available; file-read compaction retains at most roughly 4,500 characters; ANSI/TUI compaction retains at most 20 lines and 12 visible examples. Generated outputs are capped at 1,024 tokens by deleting any excess suffix.

The corresponding step-cap, timeout, and other-failure rates are 6.3% / 3.2% / 4.0% for the relevance-organized configuration and 8.7% / 3.7% / 4.3% for the evidence-only configuration. In this comparison, allowing relevance-based organization and brief evidence connection yields a shorter retained representation and better downstream score than requiring a direct evidence-only listing. This comparison isolates the documented prompt constraint; it should not be interpreted as a general benefit of unrestricted rationale generation.

B.2.4 OUTPUT CONSTRAINTS

The three output constraints are:

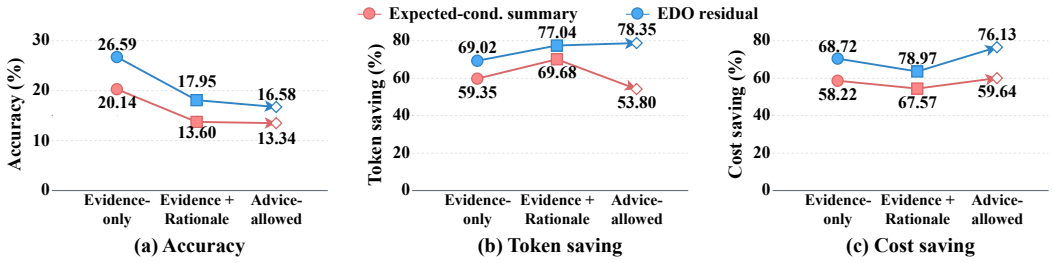

Figure 4: ProgramBench results under evidence-only, rationale, and advice-allowed output constraints.

- **evidence-only:** compressed output contains only factual evidence from the tool result.
- **evidence + rationale:** compressed output contains factual evidence plus a short explanation of why each fact matters for the task.
- **advice-allowed:** compressed output may include explicit next-step advice in addition to factual evidence.

Summary and residual variants are compared under the same task set, solver model, auxiliary model, run count, and evaluation rule. The comparison is therefore intended to isolate the effect of output constraint rather than task or evaluator differences.

The default EDO configuration is closest to evidence-focused residual construction: it is designed to expose the solver to the expected-vs-actual delta while avoiding unnecessary raw-output verbatim context.

Table 18 reports the three runs underlying Figure 4. The relevance-organized Expected + Conditional Summary variant in Table 5 may reorganize evidence by task relevance, whereas the evidence-only summary below forbids rationale and advice. All compression outputs use a maximum length of 1,024 tokens. The corrected EDO evidence-only cost saving is available as an aggregate over all runs; per-run corrected cost values are therefore not reported.

Table 18: Per-run ProgramBench results for the output-constraint study.

Representation / constraint	Metric	Run 1	Run 2	Run 3
Expected + Conditional Summary / evidence-only	Score	20.62	19.73	20.07
	Token saving	59.88	58.96	59.21
	Cost saving	57.91	58.74	58.01
EDO Residual / evidence-only	Score	27.03	26.12	26.62
	Token saving	68.71	69.54	68.81
	Cost saving	68.72 (aggregate; corrected usage accounting)		
Expected + Conditional Summary / evidence + rationale	Score	14.04	13.21	13.55
	Token saving	69.12	70.03	69.89
	Cost saving	67.88	67.09	67.74
EDO Residual / evidence + rationale	Score	17.43	18.31	18.11
	Token saving	77.39	76.62	77.11
	Cost saving	78.51	79.42	78.98
Expected + Conditional Summary / advice-allowed	Score	13.77	12.96	13.29
	Token saving	53.28	54.19	53.93
	Cost saving	60.02	59.17	59.73
EDO Residual / advice-allowed	Score	16.09	16.94	16.71
	Token saving	78.72	77.91	78.42
	Cost saving	75.69	76.58	76.12

The higher trajectory-level token savings observed under the rationale and advice-allowed constraints should not be interpreted as better compression quality. These savings are measured over complete agent trajectories and can increase when degraded feedback leads to shorter or earlier-terminating runs.

C Evidence Diagnostics and Qualitative Analysis

C.1 Fact Retention and Expectation Diagnostics

The diagnostics separate retention of factual content from explicit handling of expectations. Overall fact recall measures how much observed content survives compression. Correction recall measures retention of corrective facts, whereas contradicted-expectation correction measures whether the feedback explicitly identifies a false prediction as contradicted or corrected. Unsupported-expectation assertions capture a different error: presenting an unverified prediction as observed evidence. These metrics characterize observable properties of the retained text; they do not directly measure the agent's internal belief state or establish a causal explanation of task performance.

The source pool covers all three runs of the complete ProgramBench evaluation set: 200 tasks and 600 task-run trajectories in total. These trajectories are generated under the EDO configuration, with the raw observation, pre-execution expected observation, and solver-visible history saved at every step. From the saved steps, we construct a fixed diagnostic set of 600 tool observations. Task-Conditioned Summary, Expected + Conditional Summary, and EDO Residual are evaluated by offline replay on exactly these observations. All three receive the same task, command, tool state, and history cutoff; Task-Conditioned Summary does not receive the expected observation, whereas the other two methods receive the same saved expectation. The diagnostic compares evidence retention only and does not continue agent execution after compression.

All three auxiliary-model generations use a maximum of 1,024 tokens and are followed by the same 1,024-token hard output cap. This diagnostic does not use the 20%, 40%, 60%, or 80% joint representation budgets, and the expected observation is conditioning input rather than part of the 1,024-token compressed-output cap.

Empty outputs remain in the trajectory manifest but contribute no fact units. Timeouts and tool failures contribute fact units when they return an explicit status, error type, or error text; a completely empty return is treated as an empty output. Passthrough observations remain eligible and are scored using the exact text shown to the next solver turn. All methods use the same observation-ID set, without dropping examples because a method produced empty, malformed, or truncated output. Each record stores its run, task, step, and observation identifiers together with empty-output, timeout, tool-failure, passthrough, and hard-truncation flags.

Definitions:

- **fact:** a concrete visible statement from the raw output, such as a file path, error name, test result, return code, value, or command output.
- **correction:** evidence showing that the expected observation was wrong.
- **new evidence:** task-relevant evidence not predicted by the expected observation.
- **contradicted expectation:** an expected fact that is explicitly denied by the actual output.
- **unverified expectation:** an expected fact that is not confirmed by the raw output.

For each metric, the scoring procedure first identifies the eligible units in the raw observation and expectation, then checks whether the compressed feedback satisfies the corresponding criterion. `Qwen/Qwen3.5-4B` first proposes atomic fact units under a fixed JSON schema; a single human annotator then inspects and corrects every proposed unit and label. All reported counts use the corrected annotations and are aggregated over the same 600 observations before taking the ratio. Specifically,

$$\text{Correction Recall} = \frac{\#\{\text{raw corrective facts retained in compressed feedback}\}}{\#\{\text{raw corrective facts}\}},$$

$$\text{Contradicted Expectations Corrected} = \frac{\#\{\text{contradicted expected facts explicitly corrected}\}}{\#\{\text{contradicted expected facts}\}},$$

and

Table 19: Expectation-quality stress-test diagnostics on 600 fixed ProgramBench observations.

Condition	Overall fact recall	Contradicted exp. corrected	Unsupported as fact	Reference	Feedback	Joint
Mostly accurate expectations	3,901 / 4,920 (79.29%)	142 / 156 (91.03%)	10 / 188 (5.32%)	58	198	256
Partially incorrect expectations	3,773 / 4,920 (76.69%)	448 / 486 (92.18%)	35 / 286 (12.24%)	63	226	289
Incorrect expectations	3,639 / 4,920 (73.96%)	711 / 792 (89.77%)	53 / 684 (7.75%)	71	314	385
Irrelevant expectations	3,489 / 4,920 (70.91%)	305 / 514 (59.34%)	89 / 802 (11.10%)	67	278	345
Goal Hint	3,606 / 4,920 (73.29%)	N/A	N/A	50	334	384

Table 20: Fact-type recall in the expectation-quality stress test.

Condition	Core task facts	Error/status facts	Path/symbol facts	New evidence
Mostly accurate expectations	1,438 / 1,760 (81.70%)	724 / 820 (88.29%)	897 / 1,080 (83.06%)	842 / 1,260 (66.83%)
Partially incorrect expectations	1,376 / 1,760 (78.18%)	760 / 820 (92.68%)	864 / 1,080 (80.00%)	773 / 1,260 (61.35%)
Incorrect expectations	1,257 / 1,760 (71.42%)	777 / 820 (94.76%)	798 / 1,080 (73.89%)	807 / 1,260 (64.05%)
Irrelevant expectations	1,246 / 1,760 (70.80%)	621 / 820 (75.73%)	787 / 1,080 (72.87%)	835 / 1,260 (66.27%)
Goal Hint	1,338 / 1,760 (76.02%)	615 / 820 (75.00%)	821 / 1,080 (76.02%)	832 / 1,260 (66.03%)

$$\text{Unsupported Expectations as Facts} = \frac{\#\{\text{unverified expected facts asserted as observed facts}\}}{\#\{\text{unverified expected facts}\}}.$$

The denominator for each metric therefore consists of all corresponding eligible units found across the complete ProgramBench task-run trajectories. Every method is scored against the same denominator for that metric. All metrics are better when higher except Unsupported Expectations as Facts, where lower is better.

Metric	Task-cond. Summary	Expected + Cond. Summary	EDO Residual
Correction recall	1,420 / 2,200 (64.55%)	1,540 / 2,200 (70.00%)	2,080 / 2,200 (94.55%)
Overall fact recall	4,360 / 5,960 (73.15%)	4,720 / 5,960 (79.19%)	4,600 / 5,960 (77.18%)
New-evidence recall	1,100 / 1,600 (68.75%)	1,500 / 1,600 (93.75%)	1,561 / 1,600 (97.56%)
Contradicted expectations corrected	N/A	135 / 571 (23.64%)	536 / 571 (93.87%)
Unsupported expectations as facts	N/A	70 / 320 (21.88%)	40 / 320 (12.50%)

Figure 3(a)'s correction recall and Figure 3(b)'s contradicted-expectation correction are related but not identical: correction recall measures whether the compressed observation retains corrective facts, while contradicted expectations corrected measures whether expected facts that were false are explicitly marked as contradicted or corrected. The repeated evaluation gives 94.55% correction recall and 93.87% contradicted-expectation correction for EDO.

C.1.1 Expectation Quality Stress Test

We further test how residual feedback changes when the pre-execution expectation is more or less aligned with the subsequent observation. For fact-retention analysis, this stress test uses a separately sampled fixed set of 600 ProgramBench raw observations with its own 4,920-unit raw-fact scoring pool; this pool is distinct from the 5,960-fact default diagnostic above. For downstream evaluation, the stress test uses a controlled subset of 80 ProgramBench tasks with three runs per task. It is a stress test of expectation quality and does not replace the full 200-task ProgramBench evaluation reported in the main tables.

The partially incorrect expectation setting serves as a controlled proxy for the typical expectation quality encountered by EDO: the expectation is task-relevant but may be contradicted or left unsupported by the raw output. It is not the same scoring sample as the default EDO diagnostic above. We compare it with mostly accurate expectations, incorrect expectations, irrelevant expectations, and a Goal Hint condition that states the information need without predicting the observation.

The fact-retention results show that relevant expectations are useful, but they do not need to be perfect. Moving from mostly accurate to partially incorrect expectations reduces overall fact recall only modestly, while the contradicted-expectation correction rate remains high. Incorrect expectations preserve less overall context but still trigger many explicit corrections, whereas irrelevant expectations reduce both overall fact recall and the correction rate.

Table 21: Downstream results for the 80-task expectation-quality subset.

Condition	Correct next action	Task score	Normal end	Step cap	Timeout / other failure
Mostly accurate expectations	171 / 240 (71.25%)	68.66 / 240 (28.61%)	218 / 240 (90.83%)	13 / 240 (5.42%)	9 / 240 (3.75%)
Partially incorrect expectations	164 / 240 (68.33%)	65.95 / 240 (27.48%)	216 / 240 (90.00%)	14 / 240 (5.83%)	10 / 240 (4.17%)
Incorrect expectations	153 / 240 (63.75%)	60.86 / 240 (25.36%)	207 / 240 (86.25%)	20 / 240 (8.33%)	13 / 240 (5.42%)
Irrelevant expectations	146 / 240 (60.83%)	57.41 / 240 (23.92%)	201 / 240 (83.75%)	24 / 240 (10.00%)	15 / 240 (6.25%)
Goal Hint	157 / 240 (65.42%)	61.89 / 240 (25.79%)	211 / 240 (87.92%)	17 / 240 (7.08%)	12 / 240 (5.00%)

Table 22: Paired differences on the 80-task expectation-quality subset.

Comparison	Downstream score difference	95% CI	Two-sided p
Partially incorrect – mostly accurate expectations	-1.13	[-3.08, 0.77]	0.248
Incorrect – partially incorrect expectations	-2.12	[-4.51, 0.23]	0.079
Irrelevant – partially incorrect expectations	-3.56	[-6.01, -1.03]	0.006
Goal Hint – partially incorrect expectations	-1.69	[-4.03, 0.56]	0.142
Partially incorrect expectations – Goal Hint	+1.69	[-0.56, 4.03]	0.142

Table 23: Evidence retention under 128- and 96-token joint budgets.

Scenario	Allocation	Budget	Expectation fills budget	Residual evidence tokens	Execution status retained	Key contradiction retained	Unsupported as fact
Short output	Chronological	128	0 / 120 (0.00%)	82	119 / 120 (99.17%)	72 / 76 (94.74%)	5 / 88 (5.68%)
Short output	Status-first	128	0 / 120 (0.00%)	84	120 / 120 (100.00%)	73 / 76 (96.05%)	4 / 88 (4.55%)
Strong contradiction	Chronological	128	4 / 160 (2.50%)	95	154 / 160 (96.25%)	286 / 322 (88.82%)	24 / 180 (13.33%)
Strong contradiction	Status-first	128	1 / 160 (0.63%)	102	159 / 160 (99.38%)	309 / 322 (95.96%)	18 / 180 (10.00%)
Extreme budget	Chronological	96	31 / 140 (22.14%)	28	112 / 140 (80.00%)	112 / 196 (57.14%)	41 / 260 (15.77%)
Extreme budget	Status-first	96	2 / 140 (1.43%)	62	137 / 140 (97.86%)	174 / 196 (88.78%)	21 / 260 (8.08%)

For downstream paired inference, ProgramBench tasks are resampled as clusters, and each selected task carries all three evaluation runs with seeds 1, 2, and 3 under every compared condition. We use 10,000 bootstrap samples, percentile 95% confidence intervals computed with linear quantiles, and two-sided null-centered paired-bootstrap p-values with the finite-resampling correction. These exploratory p-values are not adjusted for multiple comparisons.

The stress test supports the usefulness of relevant expectations while showing that EDO is not strictly dependent on perfect predictions. Irrelevant expectations degrade both fact retention and downstream score. However, the differences between mostly accurate and partially incorrect expectations, and between partially incorrect expectations and Goal Hint, are not statistically reliable on this controlled subset.

C.1.2 Extremely Tight Budget Boundary Test

We next test whether the order in which the retained representation is allocated matters when the expectation itself can consume most of the available joint retained-representation budget. This boundary test is separate from the 20%–80% joint-budget sweep in Appendix B.1: instead of setting $B = \lfloor \rho L \rfloor$ relative to each raw observation length, it uses fixed absolute joint budgets of 128 or 96 tokens, shared by the retained expectation and residual evidence, to stress allocation under extremely tight contexts. The expectation-quality stress test above uses a separately sampled 600-observation pool with 4,920 raw fact units; the boundary analysis below uses a mutually exclusive 420-observation subset selected from that pool, consisting of 120 short-output observations, 160 strong-contradiction observations, and 140 extreme-budget observations.

We compare two allocation rules. *Chronological* retains generated content in its original order until the budget is exhausted. *Status-first* instead prioritizes tool exit status, timeout status, exception type, return code, direct contradictions to the expectation, test failures, assertion mismatches, missing symbols, paths, versions, generated artifacts, and other new task-relevant evidence before spending any remaining budget on expectation text.

For downstream evaluation, we use 60 ProgramBench tasks in total, stratified into 20 short-output, 20 strong-contradiction, and 20 extreme-budget tasks. Each task is run three times, giving 60 task-runs per scenario and 180 task-runs per allocation rule. These controlled-subset scores are not the full 200-task ProgramBench main results.

Paired inference for this boundary test uses the ProgramBench task as the bootstrap cluster, carrying all three runs for both allocation rules together within each resampled task. We use 10,000 bootstrap

Table 24: Downstream results under extremely tight joint budgets.

Scenario	Allocation	Correct next action	Task score	Normal end	Step cap	Timeout / other failure
Short output	Chronological	50 / 60 (83.33%)	17.00 / 60 (28.33%)	55 / 60 (91.67%)	3 / 60 (5.00%)	2 / 60 (3.33%)
Short output	Status-first	51 / 60 (85.00%)	17.30 / 60 (28.83%)	56 / 60 (93.33%)	2 / 60 (3.33%)	2 / 60 (3.33%)
Strong contradiction	Chronological	45 / 60 (75.00%)	16.20 / 60 (27.00%)	54 / 60 (90.00%)	4 / 60 (6.67%)	2 / 60 (3.33%)
Strong contradiction	Status-first	50 / 60 (83.33%)	17.40 / 60 (29.00%)	55 / 60 (91.67%)	3 / 60 (5.00%)	2 / 60 (3.33%)
Extreme budget	Chronological	31 / 60 (51.67%)	11.60 / 60 (19.33%)	47 / 60 (78.33%)	9 / 60 (15.00%)	4 / 60 (6.67%)
Extreme budget	Status-first	44 / 60 (73.33%)	15.70 / 60 (26.17%)	52 / 60 (86.67%)	5 / 60 (8.33%)	3 / 60 (5.00%)

Table 25: Status-first versus chronological allocation on the 60-task subset.

Metric	Chronological	Status-first	Difference	95% CI / two-sided p
Correct next action	126 / 180 (70.00%)	145 / 180 (80.56%)	+10.56	[3.89, 17.22] / 0.002
Task score	44.80 / 180 (24.89%)	50.40 / 180 (28.00%)	+3.11	[0.61, 5.74] / 0.016
Step cap	16 / 180 (8.89%)	10 / 180 (5.56%)	-3.33	[-7.78, 0.56] / 0.096
Normal end	156 / 180 (86.67%)	163 / 180 (90.56%)	+3.89	–
Timeout / other failure	8 / 180 (4.44%)	7 / 180 (3.89%)	-0.56	–

samples, percentile 95% confidence intervals, and two-sided null-centered paired-bootstrap p-values with finite-resampling correction. The reported p-values are unadjusted exploratory estimates.

The boundary results show that allocation order matters most when the budget is extremely tight. Under short outputs, both rules almost always retain status and contradictions, so downstream differences are small. Under the 96-token extreme-budget setting, chronological retention often allows expectation text to fill the budget before residual evidence is preserved, whereas Status-first sharply reduces expectation-only budget exhaustion and retains more execution status and contradiction evidence. The paired downstream results are exploratory and unadjusted for multiple comparisons, but they support using a status- and contradiction-prioritized ordering when the joint budget is small enough for expectation text to crowd out residual evidence.

C.2 Figure 1 ProgramBench Trajectory Excerpt

Figure 1 uses a normalized excerpt from ProgramBench task `elkowar__pipr.fae0b17`, EDO run 1, solver step 6, tool call 5. ProgramBench begins from a reference executable and usage documentation rather than the original repository source tree or official test sources (Yang et al., 2026). In the interaction illustrated here, `README.md` and `docs/setup.md` are initial documentation, whereas `src/config.py` and `tests/test_config.py` are candidate workspace files created by the agent during preceding reconstruction steps. The latter test file is an agent-generated local probe, not part of the hidden ProgramBench evaluation suite.

The search invocation, line-wrapped below without changing its arguments, was:

```
rg -n -i ''parse[_ -]?config|load[_ -]?config|
configuration loading'' \
  src tests README.md docs
```

It exited with status 0 and returned an empty standard-error stream. For legibility, the figure removes line numbers, folds repeated matches into one representative entry per source, and shortens signatures. Its displayed search excerpt is:

```
src/config.py:  def load_config(path)
tests/test_config.py:  test_load_config()
README.md:  use load_config(...)
docs/setup.md:  configuration loading
```

The sentence `No parse_config; found load_config.` is a derived correction from the absence of `parse_config` matches and the presence of `load_config` matches; it is not a verbatim line emitted by `rg`. The figure includes it in the displayed search block to make the expectation–observation mismatch explicit.

The figure's *Context* line is a normalized description of the agent's intermediate working assumption after earlier reconstruction steps, not the original ProgramBench task instruction. The expected observation is the prediction emitted immediately before this search:

```
Context:
Fix configuration loading by editing parse_config, which is
expected to exist
in the codebase.

Search query:
Search for parse_config and related configuration-loading
symbols across the
source, tests, README, and setup documentation.

Expected observation:
A parse_config definition and its relevant usages will be
found.

Raw search output:
src/config.py:  def load_config(path)
tests/test_config.py:  test_load_config()
README.md:  use load_config(...)
docs/setup.md:  configuration loading
No parse_config; found load_config.

Ordinary summary:
Relevant configuration-loading code was found in
src/config.py, with
corresponding tests and usage documentation.

EDO residual:
src/config.py and tests/test_config.py matched.
No parse_config; found load_config.
```

The result contradicts the expectation that `parse_config` exists and identifies `load_config` as the relevant candidate entry. The summary displayed in the figure is an offline summary generated from the same displayed evidence for the controlled visualization; it was not part of the retained EDO trajectory and was not shown to the EDO solver at this step. It retains the broad fact that configuration-loading code was found but omits the correction. The EDO residual was the feedback retained in the online trajectory; it preserves both the matched files and the explicit contradiction. The next recorded action loaded the agent-created `src/config.py` and `tests/test_config.py` workspace files and did not continue searching for `parse_config`.

The three action labels in the figure denote the following candidate next steps:

- `load`: inspect or edit the existing `load_config` implementation;
- `parse`: continue acting on the unsupported assumption that `parse_config` exists;
- `again`: repeat the search without using the correction already returned.

The action distributions in Figure 1 are a post-hoc controlled analysis rather than ProgramBench task scores. They are estimated from 200 samples for each fixed context using temperature 0.7 and `top_p`=0.95. The figure's three contexts contain the normalized raw search feedback, the offline summary, and the EDO residual feedback, respectively. This nonzero temperature is used only for the Figure 1 distribution estimate; all benchmark experiments use temperature 0. The observed counts are:

Context	`load`	`parse`	`again`	Total
Raw	144 (72%)	36 (18%)	20 (10%)	200
Summary	44 (22%)	126 (63%)	30 (15%)	200
EDO residual	138 (69%)	40 (20%)	22 (11%)	200

C.3 Qualitative Failure Case

We also observe cases where residualization preserves the most explicit expectation–observation mismatch but under-preserves contextual evidence needed for root-cause localization. In one ProgramBench configuration-loading task, the raw trajectory retained surrounding evidence indicating that an environment-specific path should override the default path in `load_config`. The corresponding test, `test_env_path_overrides_default`, therefore required reasoning about path-priority behavior rather than only identifying the immediate failing assertion.

The EDO residual in this case focused on the direct mismatch visible in the test output and compressed away part of the surrounding path-priority context. The downstream solver consequently attempted an incorrect fix, whereas the Raw agent retained enough context to solve the task. This example illustrates a limitation of expectation-conditioned residuals: they are most effective when decision-relevant evidence is explicit in the observation, but they can under-preserve implicit causal context when the residual extractor focuses too narrowly on explicit differences.

This case delineates the scope of expectation-conditioned feedback: identifying what was wrong is not equivalent to retaining all the evidence needed to determine the correct repair. A useful residual must preserve not only the immediate mismatch but also the contextual constraints required to interpret it. Explicit correction is therefore one requirement for effective feedback rather than a substitute for broader evidence retention.

D Efficiency and Trajectory Analysis

D.1 Token, Cost, and Latency

D.1.1 Token Accounting

Main input and main output tokens are taken from the solver-model API usage. Compressor input and output tokens are taken from the auxiliary-model API usage. The expected observation is generated by the main model, so its tokens are counted as main-model output at the turn where it is written. When that expectation is read again in later turns, it is counted as main-model input.

Total token saving is computed by summing usage over the whole benchmark before taking the ratio:

```
Total Token Saving =
(Raw main input + Raw main output − Method main input −
Method main output)
/
(Raw main input + Raw main output)
```

Auxiliary-model tokens are not included in Total Token Saving. They are included in cost.

D.1.2 Cache-Aware Cost

Cost is reported in Chinese yuan (CNY) using accounting prices per one million tokens. The main-model price schedule is taken from the Alibaba Cloud Bailian model-service pricing table used at the time of the experiments. Access to `Qwen/Qwen3.5-4B` was free in the experimental environment. To avoid giving methods that use this auxiliary model an artificial zero-cost advantage, we conservatively impute its accounting price as one tenth of the corresponding `Qwen/Qwen3.7-flash` price for regular input, cache creation, cache hit, and output tokens. This one-tenth rate is an evaluation assumption rather than an observed platform charge. The same accounting rule is applied to every method using the auxiliary model, and the resulting costs are intended for controlled relative comparison rather than as a guarantee of future platform prices.

Accounting model	Regular input	Cache creation	Cache hit	Output
`Qwen/Qwen3.7-flash`	0.20	0.25	0.02	2.00
`Qwen/Qwen3.5-4B`	0.02	0.025	0.002	0.20

Table 26: ProgramBench cache-category and auxiliary-model accounting. Absolute cost is in CNY per task.

Method	Main regular	Main create	Main hit	Main output	Aux. input	Aux. output	Cost
Raw	30.00	70.00	263.98	13.48	0	0	0.05574
Generic Summary	25.40	30.94	115.56	4.43	11.51	1.57	0.02453
Task-Conditioned Summary	22.79	28.99	109.27	4.29	11.29	1.54	0.02310
LLMLingua-2	41.07	63.33	247.43	11.70	0	0	0.05240
Head-Tail	27.65	34.96	131.61	5.45	0	0	0.02780
AgentDiet	9.52	23.30	96.65	5.09	10.87	1.48	0.02036
EDO	15.00	20.44	78.12	3.37	40.91	1.01	0.01743

Regular-input, cache-creation, and cache-hit tokens are mutually exclusive input categories. For model m, its cache-aware cost is

$$C_m = \frac{p_m^{\text{in}} N_m^{\text{in}} + p_m^{\text{create}} N_m^{\text{create}} + p_m^{\text{hit}} N_m^{\text{hit}} + p_m^{\text{out}} N_m^{\text{out}}}{10^6},$$

where N_m^{in}, N_m^{create}, N_m^{hit}, and N_m^{out} denote regular input, cache-creation input, cache-hit input, and output tokens, respectively. The total method cost sums the solver and all auxiliary-model costs:

$$C_{\text{method}} = C_{\text{solver}} + \sum_{a \in \text{aux}} C_a.$$

For the no-cache estimate, all solver input tokens (regular, cache-creation, and cache-hit categories) are charged at the regular-input rate; auxiliary-model input and output use their ordinary input and output rates.

Cost saving is then computed from benchmark-level totals:

$$\text{Cost Saving} = 1 - \frac{C_{\text{method}}}{C_{\text{Raw}}}.$$

D.1.3 ProgramBench Token and Cost Breakdown

Table 26 reports average token usage and absolute cache-aware cost per task. Token columns are in thousands. The 0.60K expected-observation tokens for EDO are main-model generations: they are counted as main-model output when generated and as main-model input when read in later turns, rather than added as a separate token category. The 9.40 tool calls per task comprise 3.20 passthrough calls, 2.00 deterministic file-compaction calls, and 4.20 residual-model calls. A residual call averages 9.74K input tokens and 0.24K output tokens, giving 40.91K auxiliary input and 1.01K auxiliary output tokens per task.

Across all 600 ProgramBench task-runs, the corresponding total costs are 33.44 CNY for Raw, 14.72 CNY for Generic Summary, 13.86 CNY for Task-Conditioned Summary, 31.44 CNY for LLMLingua-2, 16.68 CNY for Head-Tail, 12.22 CNY for AgentDiet, and 10.46 CNY for EDO.

The ProgramBench conclusion is robust to the imputed auxiliary-model price. Let (k) scale the auxiliary schedule above, with (k=1) denoting the reported one-tenth schedule and (k=10) pricing auxiliary tokens at the corresponding solver rates. Using the observed token categories, EDO's per-task cost is (0.0164098+0.0010202k) CNY. At (k=10), its estimated cost is 0.02661 CNY per task, still 52.3% below Raw; the estimate reaches Raw only near (k=38.6). This sensitivity calculation changes prices while holding the observed trajectories fixed.

D.1.4 Latency

Latency is measured from the start of a task run to the end of the task run, including solver calls, auxiliary compression calls, tool execution, network waiting, retries, and benchmark environment interaction. Environment initialization and evaluation are reported separately when needed.

Reported latency values use the same task set and run count as the corresponding score and token/cost measurements. Each task is run 3 times, and tables report the average.

D.2 PROGRAMBENCH RUN VARIATION AND TRAJECTORY ANALYSIS

D.2.1 PER-RUN SCORES AND PAIRED STATISTICS

Table 27: ProgramBench score (%) in each of the three runs.

Method	Run 1	Run 2	Run 3
Raw	29.74	28.96	29.53
EDO	27.03	26.12	26.62
Generic Summary	25.12	26.03	25.62
Expected + Conditional Summary	22.91	22.06	22.50

For paired inference, the 200 task clusters are resampled with replacement 10,000 times. For each bootstrap sample and method, the ProgramBench score is recomputed as the pooled achieved score divided by the pooled maximum score over all selected task-runs; paired differences are then formed within the same resample. Table 28 reports two-sided paired bootstrap results. Small variation under temperature-zero decoding can still arise from service-side batching and numerical nondeterminism, tool-output ordering, network or process timeouts, dependency downloads, and test timing.

Table 28: Paired bootstrap comparisons on ProgramBench. Differences are in percentage points.

Comparison	Paired difference	95% CI	Two-sided p
EDO – Raw	-2.82	[-5.62, 0.08]	0.057
EDO – Generic Summary	+1.00	[-1.42, 3.49]	0.421
EDO – Expected + Conditional Summary	+4.10	[1.44, 6.83]	0.003

D.2.2 TRAJECTORY LENGTH AND TERMINATION

All statistics in this subsection use the 200 ProgramBench tasks and three independent runs. Table 29 uses all 600 task-runs as its statistical units.

Table 29: ProgramBench trajectory length and termination statistics.

Method	Mean/median steps	Tool calls	Step cap	Timeout	Other failure	Normal end	Raw obs.	Retained repr.	Final context
Raw	17.8 / 18	13.6	10.5%	4.0%	3.5%	82.0%	2,840	2,840	42.6K
Generic Summary	15.2 / 14	11.6	8.3%	3.5%	4.0%	84.2%	2,790	410	22.8K
Task-Conditioned Summary	14.8 / 14	11.3	7.8%	3.3%	4.2%	84.7%	2,805	385	21.5K
LLMLingua-2	17.4 / 17	13.2	10.0%	4.2%	3.8%	82.0%	2,830	2,735	40.8K
Head-Tail	13.4 / 12	10.1	6.5%	3.0%	4.0%	86.5%	2,775	740	25.2K
AgentDiet	12.9 / 12	9.8	5.8%	3.2%	4.0%	87.0%	2,815	520	18.9K
EDO	12.3 / 11	9.4	4.2%	2.5%	3.3%	90.0%	2,820	286	14.1K

The EDO retained representation averages 64 expectation tokens and 222 residual tokens. Here, normal completion denotes a task-run that ended without hitting the step limit, timeout, or other runtime-failure category. It is not the benchmark success rate; task success is still determined by the official evaluator and, for task-level groupings, by majority vote over the three runs. For Tables 30 and 31, the three runs are first aggregated within each task and success status is assigned by majority outcome. The successful and failed task groups in Table 30 are defined separately for each method, so the successful-task set is not necessarily identical across methods.

Table 30: Average ProgramBench main-agent tokens and latency for successful and failed tasks.

Method	Successful tasks Main tokens / latency	Failed tasks Main tokens / latency
Raw	421.0K / 139.8 s	359.2K / 119.5 s
Generic Summary	198.4K / 103.4 s	168.7K / 87.8 s
Task-Conditioned Summary	186.7K / 106.2 s	158.4K / 91.0 s
LLMLingua-2	401.8K / 137.0 s	351.2K / 120.4 s
Head-Tail	224.3K / 70.4 s	195.5K / 61.1 s
AgentDiet	151.7K / 72.2 s	129.7K / 61.0 s
EDO	131.0K / 52.1 s	111.8K / 41.6 s

Table 31: Task-level success-status groups for EDO and Raw on ProgramBench.

Group	Tasks	EDO token saving	Step change
Both successful	49 / 200 (24.5%)	65.4%	-4.10
Both failed	137 / 200 (68.5%)	70.8%	-5.92
EDO successful, Raw failed	4 / 200 (2.0%)	61.7%	-3.50
Raw successful, EDO failed	10 / 200 (5.0%)	73.1%	-7.40

D.2.3 Step-Count-Matched Prefix Token Saving

We additionally compare Raw and EDO over prefixes with the same number of completed solver steps. Trajectories are paired by benchmark, task ID, and run ID. For pair i, the prefix endpoint is

$$K_i = \min(S_{\mathrm{Raw},i}, S_{\mathrm{EDO},i}),$$

where $S_{m,i}$ is the total number of completed solver steps for method m. Both prefixes start from the initial task input and end after step K_i. Matching does not require the agents to execute the same commands, stop at their first action divergence, or receive fixed observations through offline replay. This is therefore a descriptive control for solver-step count rather than a counterfactual same-action comparison.

For method m, we first accumulate main-model input and output tokens through step K_i, then compute

$$\bar{T}_m^{\mathrm{prefix}} = \frac{1}{600}\sum_{i=1}^{600}\left(T_{m,i,\mathrm{input}}^{(1:K_i)} + T_{m,i,\mathrm{output}}^{(1:K_i)}\right), \qquad \mathrm{Saving}_{\mathrm{prefix}} = 1 - \frac{\bar{T}_{\mathrm{EDO}}^{\mathrm{prefix}}}{\bar{T}_{\mathrm{Raw}}^{\mathrm{prefix}}}.$$

Each of the 200 tasks contributes exactly three paired runs, so averaging over the 600 pairs is equivalent to first averaging the three runs within each task and then averaging over tasks. We therefore report token quantities per task.

Table 32: Step-count-matched prefix statistics on ProgramBench. Token quantities are averages per task.

Statistic	Raw	EDO
Paired trajectories	600	600
Matched prefix, mean / median steps	11.4 / 11	11.4 / 11
Matched prefix, Q1 / Q3 steps	7 / 15	7 / 15
Mean tool calls in prefix	8.7	8.7
Main input tokens	262.50K	100.91K
Main output tokens	9.75K	3.09K
Main total tokens	272.25K	104.00K

The resulting step-count-matched prefix saving is

$$1 - \frac{104.00\mathrm{K}}{272.25\mathrm{K}} = 61.80\%.$$

EDO therefore retains substantial main-agent token savings after matching the number of solver steps. The reduction from the 69.02% complete-trajectory saving to 61.80% is consistent with part of the complete-trajectory saving being associated with EDO's shorter trajectories. Because the paired agents may execute different commands and receive different observations inside the matched prefixes, the comparison does not isolate trajectory length causally.

E Robustness and Additional Benchmarks

E.1 Additional Solver Robustness Check

All main results in the paper use `Qwen/Qwen3.7-flash` as the solver. This section reports an additional ProgramBench robustness check with `Qwen3.8-27B` as the solver while keeping the

compressor fixed as `Qwen/Qwen3.5-4B`. This experiment does not change the main experimental configuration; it tests whether the advantage of residual feedback over an expectation-conditioned summary remains under a stronger solver.

In this section, Qwen3.8-matched Exp.+Cond. Summary denotes the Expected + Conditional Summary baseline evaluated with the same `Qwen3.8-27B` solver, task set, run count, decoding setting, and evaluation environment as EDO.

E.1.1 Score and Efficiency

Table 33: ProgramBench robustness results with `Qwen3.8-27B`.

Method	Run 1	Run 2	Run 3	Mean score	Main tokens
Raw	33.72%	32.89%	33.41%	33.34%	348.40K
Qwen3.8-matched Exp.+Cond. Summary	27.86%	27.08%	27.42%	27.45%	105.05K
EDO	31.62%	30.79%	31.21%	31.21%	100.06K

Table 34: Cache-aware token and cost accounting with `Qwen3.8-27B`.

Method	Main regular	Main create	Main hit	Main output	Aux. input	Aux. output	Cost	Cost saving
Raw	24.50	66.00	246.00	11.90	0	0	0.61140	0.00%
Qwen3.8-matched Exp.+Cond. Summary	12.70	19.60	69.70	3.05	39.40	1.14	0.19104	68.75%
EDO	12.30	18.70	66.20	2.86	40.20	0.98	0.18207	70.22%

The `Qwen3.8-27B` accounting prices are 3.00 CNY per million regular-input tokens, 3.75 for cache creation, 0.60 for cache hits, and 12.00 for output. The auxiliary `Qwen/Qwen3.5-4B` compressor is charged using the same conservative accounting rates used elsewhere in the paper. Under this stronger solver, all absolute ProgramBench scores are higher than in the main `Qwen/Qwen3.7-flash` configuration. EDO scores 31.21%, compared with 27.45% for the matched expectation-conditioned summary baseline, while reducing main-agent tokens by 71.28% and estimated cost by 70.22% relative to Raw.

E.1.2 Paired Comparison

For paired inference, we use the same ProgramBench task-clustered bootstrap procedure as in Appendix D.2: 200 task clusters are resampled with replacement 10,000 times, each selected task carries all three runs, ProgramBench score is recomputed as the pooled achieved score divided by the pooled maximum score, and paired differences are formed within the same resample. The confidence interval is the percentile 95% interval, and the p-value is a two-sided null-centered paired-bootstrap value with the finite-resampling correction. These p-values are not adjusted for multiple comparisons.

Table 35: Paired ProgramBench comparison under the `Qwen3.8-27B` solver. Differences are in percentage points.

Comparison	Paired difference	95% CI	Two-sided p
EDO – Qwen3.8-matched Exp.+Cond. Summary	+3.76	[1.02, 6.48]	0.007

E.1.3 Prompt-Paraphrase Check

We also paraphrase the expectation-conditioned summary and EDO prompt wording while keeping the solver, compressor, task set, and evaluator fixed. The three paraphrases are intended as a sensitivity check for prompt wording rather than as new methods.

Table 36: Prompt-paraphrase results under the `Qwen3.8-27B` solver. Main tokens are averages per task.

Variant	Method	Run 1	Run 2	Run 3	Mean score
A	Qwen3.8-matched Exp.+Cond. Summary	27.91%	27.18%	27.16%	27.42%
A	EDO	31.26%	30.72%	30.99%	30.99%
B	Qwen3.8-matched Exp.+Cond. Summary	27.34%	26.58%	27.14%	27.02%
B	EDO	31.75%	30.86%	31.00%	31.20%
C	Qwen3.8-matched Exp.+Cond. Summary	27.08%	27.86%	27.47%	27.47%
C	EDO	30.31%	31.42%	30.75%	30.83%

Across the three paraphrases, the matched expectation-conditioned summary baseline averages a 27.30% score with a sample standard deviation of 0.25 percentage points, while EDO averages 31.01% with a sample standard deviation of 0.19 points. The EDO advantage ranges from +3.36 to +4.18 points, with a mean difference of +3.70 points and sample standard deviation of 0.43 points. This indicates small variation in this robustness check, not invariance to prompt wording in general.

Table 37: Prompt-paraphrase paired comparisons under the `Qwen3.8-27B` solver. Differences are in percentage points.

Variant	Paired difference	95% CI	Two-sided p
A	+3.57	[0.62, 6.31]	0.018
B	+4.18	[1.37, 6.86]	0.004
C	+3.36	[0.41, 6.04]	0.025
Pooled variants	+3.70	[1.02, 6.23]	0.006

E.1.4 Trajectory Statistics

Table 38: Trajectory and retained-representation statistics with `Qwen3.8-27B`.

Setting	Method	Reference length	Feedback length	Joint length	Mean/median steps	Tool calls	Normal end
Default	Qwen3.8-matched Exp.+Cond. Summary	64	396	460	11.9 / 11	9.1	88.7%
Default	EDO	64	214	278	11.3 / 10	8.7	91.2%
Prompt paraphrase mean	Qwen3.8-matched Exp.+Cond. Summary	64	405	469	12.1 / 11	9.2	88.3%
Prompt paraphrase mean	EDO	64	219	283	11.5 / 10	8.8	90.7%

Under the stronger solver, EDO retains shorter feedback than the matched expectation-conditioned summary baseline and shows slightly shorter trajectories in this additional check. These results support the robustness of the residual-feedback comparison on ProgramBench, but they should not be read as a claim that the same magnitude of improvement holds for every solver or benchmark.

E.2 Additional TerminalBench and DevEval Results

We report two representative additional checks beyond ProgramBench: TerminalBench, where EDO improves over Raw in the main results, and DevEval, where EDO trails Raw. These results illustrate whether the same budget trends appear outside ProgramBench; they are not a complete six-benchmark budget expansion, and the main conclusions remain based on the six-benchmark main tables and the systematic ProgramBench analyses.

E.2.1 Additional Controls Under the Default Configuration

Table 39: Default matched controls on TerminalBench and DevEval.

Benchmark	Method	Run 1	Run 2	Run 3	Mean	Token saving	Cost saving
TerminalBench	Exp.+Cond. Summary	39/86 (45.35%)	41/86 (47.67%)	39/86 (45.35%)	119/258 (46.12%)	29.80%	24.50%
TerminalBench	Goal Hint + Evidence	44/86 (51.16%)	41/86 (47.67%)	47/86 (54.65%)	132/258 (51.16%)	35.60%	29.30%
TerminalBench	Recent-Obs. Masking	41/86 (47.67%)	45/86 (52.33%)	43/86 (50.00%)	129/258 (50.00%)	32.40%	26.10%
DevEval	Exp.+Cond. Summary	10/66 (15.15%)	11/66 (16.67%)	12/66 (18.18%)	33/198 (16.67%)	23.60%	20.90%

Table 40: Task-clustered paired differences on TerminalBench and DevEval.

Benchmark	Comparison	Difference	95% CI	Two-sided p
TerminalBench	EDO – Raw	+2.33	[-5.04, 9.69]	0.577
TerminalBench	EDO – Expected + Conditional Summary	+8.53	[0.78, 15.89]	0.031
TerminalBench	EDO – Task-Conditioned Summary	+27.91	[20.54, 34.88]	< 0.001
TerminalBench	EDO – AgentDiet	+32.56	[26.36, 38.76]	< 0.001
DevEval	EDO – Raw	-3.03	[-9.60, 3.54]	0.399
DevEval	EDO – Expected + Conditional Summary	+3.03	[-2.53, 9.09]	0.339
DevEval	EDO – AgentDiet	+1.52	[-5.05, 8.59]	0.716

Table 41: Per-task DevEval differences in successful runs.

Comparison	Per-task difference	Number of tasks	Contribution
EDO – Expected + Conditional Summary	-1	9	-9
	0	46	0
	+1	8	+8
	+2	2	+4
	+3	1	+3
	Total	66	+6
EDO – AgentDiet	-2	1	-2
	-1	14	-14
	0	37	0
	+1	9	+9
	+2	5	+10
	Total	66	+3

E.2.2 Task-Clustered Paired Comparisons

TerminalBench and DevEval contain 86 and 66 tasks, respectively, with three runs per task. The reported binary accuracy is pooled over all task-runs. For paired inference, tasks are resampled with replacement 10,000 times, and each selected task carries all three runs under every compared method. Each bootstrap sample recomputes pooled run-level accuracy before forming the paired difference. We report percentile 95% confidence intervals and two-sided null-centered paired-bootstrap p-values with the finite-resampling correction. The task is the resampling cluster; the reported metric is not a majority-vote task-level success rate. These comparisons are exploratory, and the p-values are not adjusted for multiple comparisons.

For an anonymized view of the DevEval pairing structure, Table 41 reports the distribution of per-task differences in the number of successful runs. A value of $+1$, for example, means that EDO succeeds in one more of the three runs than the comparison method on that task.

On TerminalBench, EDO does not differ significantly from Raw, while its comparisons with Expected + Conditional Summary, Task-Conditioned Summary, and AgentDiet have confidence intervals above zero; the $p = 0.031$ comparison is interpreted as nominal significance under the unadjusted exploratory analysis. On DevEval, all three intervals include zero. The higher EDO point estimates relative to Expected + Conditional Summary and AgentDiet on DevEval therefore do not establish a reliable improvement, and the lower point estimate relative to Raw does not establish a reliable degradation.

E.2.3 Joint Representation Budgets

The budget definition is identical to the ProgramBench joint-budget experiment: each ratio is relative to the raw token count of the corresponding tool observation, and the fixed 1,024-token non-budget generation cap is not applied. In this compact table, Exp.+Cond. Summary abbreviates Expected + Conditional Summary. The EDO budget is shared by the expected observation and residual, and the Expected + Conditional Summary budget is shared by the expected observation and summary. The 80% setting is not the default EDO configuration: it applies a per-observation retained-representation budget shared by the expectation and residual, whereas the default EDO run uses the fixed 1,024-token model-generated compression cap. Therefore, the 80% budget results are not expected to match the default main-table scores.

Table 42: Per-run success rates under joint representation budgets on TerminalBench and DevEval.

Benchmark	Method	Budget	Run 1	Run 2	Run 3	Mean
TerminalBench	Exp.+Cond. Summary	20%	22/86 (25.58%)	24/86 (27.91%)	19/86 (22.09%)	65/258 (25.19%)
TerminalBench	Exp.+Cond. Summary	40%	31/86 (36.05%)	33/86 (38.37%)	32/86 (37.21%)	96/258 (37.21%)
TerminalBench	Exp.+Cond. Summary	60%	42/86 (48.84%)	38/86 (44.19%)	41/86 (47.67%)	121/258 (46.90%)
TerminalBench	Exp.+Cond. Summary	80%	44/86 (51.16%)	42/86 (48.84%)	43/86 (50.00%)	129/258 (50.00%)
TerminalBench	Goal Hint + Evidence	20%	29/86 (33.72%)	31/86 (36.05%)	28/86 (32.56%)	88/258 (34.11%)
TerminalBench	Goal Hint + Evidence	40%	38/86 (44.19%)	35/86 (40.70%)	39/86 (45.35%)	112/258 (43.41%)
TerminalBench	Goal Hint + Evidence	60%	41/86 (47.67%)	44/86 (51.16%)	42/86 (48.84%)	127/258 (49.22%)
TerminalBench	Goal Hint + Evidence	80%	45/86 (52.33%)	43/86 (50.00%)	44/86 (51.16%)	132/258 (51.16%)
TerminalBench	Recent-Obs. Masking	20%	25/86 (29.07%)	27/86 (31.40%)	24/86 (27.91%)	76/258 (29.46%)
TerminalBench	Recent-Obs. Masking	40%	34/86 (39.53%)	36/86 (41.86%)	35/86 (40.70%)	105/258 (40.70%)
TerminalBench	Recent-Obs. Masking	60%	42/86 (48.84%)	39/86 (45.35%)	43/86 (50.00%)	124/258 (48.06%)
TerminalBench	Recent-Obs. Masking	80%	44/86 (51.16%)	46/86 (53.49%)	43/86 (50.00%)	133/258 (51.55%)
TerminalBench	EDO	20%	33/86 (38.37%)	36/86 (41.86%)	31/86 (36.05%)	100/258 (38.76%)
TerminalBench	EDO	40%	43/86 (50.00%)	38/86 (44.19%)	41/86 (47.67%)	122/258 (47.29%)
TerminalBench	EDO	60%	43/86 (50.00%)	47/86 (54.65%)	44/86 (51.16%)	134/258 (51.94%)
TerminalBench	EDO	80%	47/86 (54.65%)	44/86 (51.16%)	45/86 (52.33%)	136/258 (52.71%)
DevEval	Exp.+Cond. Summary	20%	4/66 (6.06%)	5/66 (7.58%)	3/66 (4.55%)	12/198 (6.06%)
DevEval	Exp.+Cond. Summary	40%	8/66 (12.12%)	7/66 (10.61%)	8/66 (12.12%)	23/198 (11.62%)
DevEval	Exp.+Cond. Summary	60%	10/66 (15.15%)	12/66 (18.18%)	11/66 (16.67%)	33/198 (16.67%)
DevEval	Exp.+Cond. Summary	80%	13/66 (19.70%)	11/66 (16.67%)	12/66 (18.18%)	36/198 (18.18%)
DevEval	EDO	20%	5/66 (7.58%)	7/66 (10.61%)	6/66 (9.09%)	18/198 (9.09%)
DevEval	EDO	40%	11/66 (16.67%)	9/66 (13.64%)	10/66 (15.15%)	30/198 (15.15%)
DevEval	EDO	60%	11/66 (16.67%)	13/66 (19.70%)	12/66 (18.18%)	36/198 (18.18%)
DevEval	EDO	80%	14/66 (21.21%)	12/66 (18.18%)	13/66 (19.70%)	39/198 (19.70%)

The residual representation generally shows clearer separation at lower budgets. At higher retained budgets, Expected + Conditional Summary preserves more of the raw observation, so the gap becomes smaller.

F A Minimal Analysis of Feedback Compression

This analysis shows why fact coverage need not preserve decision-relevant distinctions. It neither proves that EDO preserves every such distinction nor establishes residual optimality.

F.1 High Fact Coverage Does Not Guarantee Decision Fidelity

Let an observation contain n background facts shared by two outcomes and an equiprobable binary outcome $Y \in \{0, 1\}$ that determines the correct next action. A compressor retains all n background facts but removes Y. Factual precision is one, while fact recall is

$$\frac{n}{n+1} \longrightarrow 1 \qquad \text{as } n \longrightarrow \infty.$$

The compressed representation is identical under $Y = 0$ and $Y = 1$, so any decision rule using it has minimum error probability $1/2$, whereas the raw observation permits zero error. Fact recall can therefore approach one while erasing the distinction that determines the next action. Given a fixed expectation, the omitted outcome also determines whether the observation confirms or contradicts it.

F.2 A Bound for Indistinguishable Feedback

Fix a pre-execution context $c = (\bar{\tau}_t, a_t, \hat{o}_t)$ and two raw observations o_1, o_2 that induce next-action distributions

$$p_1 = \pi_\theta(\cdot \mid c, o_1), \qquad p_2 = \pi_\theta(\cdot \mid c, o_2).$$

If a compressor maps both observations to the same feedback, $C_c(o_1) = C_c(o_2) = z$, the solver produces a single distribution $q = \pi_\theta(\cdot \mid c, z)$.

Proposition. For total variation distance,

$$\max\{\mathrm{TV}(p_1, q), \mathrm{TV}(p_2, q)\} \geq \frac{1}{2}\,\mathrm{TV}(p_1, p_2).$$

Proof. The triangle inequality gives

$$\mathrm{TV}(p_1, p_2) \leq \mathrm{TV}(p_1, q) + \mathrm{TV}(q, p_2).$$

At least one term on the right must therefore be no smaller than half of the left-hand side.

If two outcomes call for different responses but become indistinguishable after compression, at least one response cannot be reproduced exactly. The result concerns erased distinctions, not residual wording: a conventional summary can preserve them, and distinguishability alone does not ensure correct use. The matched controls and diagnostics supply the empirical evidence for EDO's specific representation.