PACK LIGHT, CARRYON: BUILDING OBJECT-ORIENTED CONTEXT FOR LONG-HORIZON AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon agents accumulate progress in persistent task state, yet repeatedly process interaction histories describing how that progress was achieved. This raises a natural question: can long-horizon context instead be organized around reusable progress rather than the history that produced it? In this paper, we introduce CarryOn, an **object-oriented context runtime** that organizes continuation around reusable task objects. Rather than treating compressed histories or reconstructed accounts of progress as the primary basis for continuation, CarryOn builds a context view from continuation carriers, complementary execution context, and a live tail of recent interactions. A core challenge is keeping this reusable basis valid after its surrounding history is retired: how can CarryOn update the basis without reconstructing the full representation at every fold? CarryOn maintains carrier records across folds, revising affected records against bound evidence and applicable environment checks while carrying forward unaffected records and jointly updating execution context. Experiments on OSWorld2.0 and BrowseComp-Plus show improved gains of 5.8% and 7.0% while reducing total token usage by 51.9% and 22.9%, respectively.

1 INTRODUCTION

Long-horizon agents accumulate progress in two places: persistent task state, which embodies what has been achieved, and interaction history, which records how it was achieved (Yi et al., 2026). As shown in Figure 1, once images have been inserted into a presentation or measurements have been entered into a report, the resulting task state already captures that progress for later steps to inspect and modify (Yang et al., 2024; Xie et al., 2024). As execution continues, this redundancy grows: agents repeatedly process accounts of how results already available in the task state were produced. Existing methods manage growing histories by summarizing, retrieving, or pruning past interactions (Wu et al., 2026b; Packer et al., 2024; Xiao et al., 2026; Chen et al., 2026b; Kang et al., 2025; Xu et al., 2025), or by translating them into explicit progress records and structured task states (Wu et al., 2026a; Chen et al., 2026a; Wang et al., 2024). These approaches reduce context growth, but when they substitute for reusable task results, they make continuation depend on a redundant, selective account that can drift from the state the agent directly inspects and modifies.

We challenge this reliance on reconstructed accounts of progress with an **object-oriented** view of long-horizon context, in which **reusable task objects** form the basis for continuation. These objects include work products, source-grounded findings, and application states that remain usable under explicit conditions. The agent can recover relevant information through direct inspection and use of these objects, while context supplies the complementary information needed to continue. To realize this view, we introduce CarryOn, a context runtime that organizes the agent's context into three complementary components: *continuation carriers*, which bind reusable content or object references to their conditions of use; *execution context*, which preserves active goals, applicable constraints, and unresolved issues not captured by the carriers; and *live tail*, which retains interactions since the most recent successful fold.

A central challenge in building object-oriented long-horizon context is retiring interaction history once progress becomes reusable without losing the information required for continuation, which poses three challenges: *1) how to identify reusable progress beyond its surrounding history:* Some results still depend on prior interactions for interpretation or use. CarryOn captures these dependen-

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.

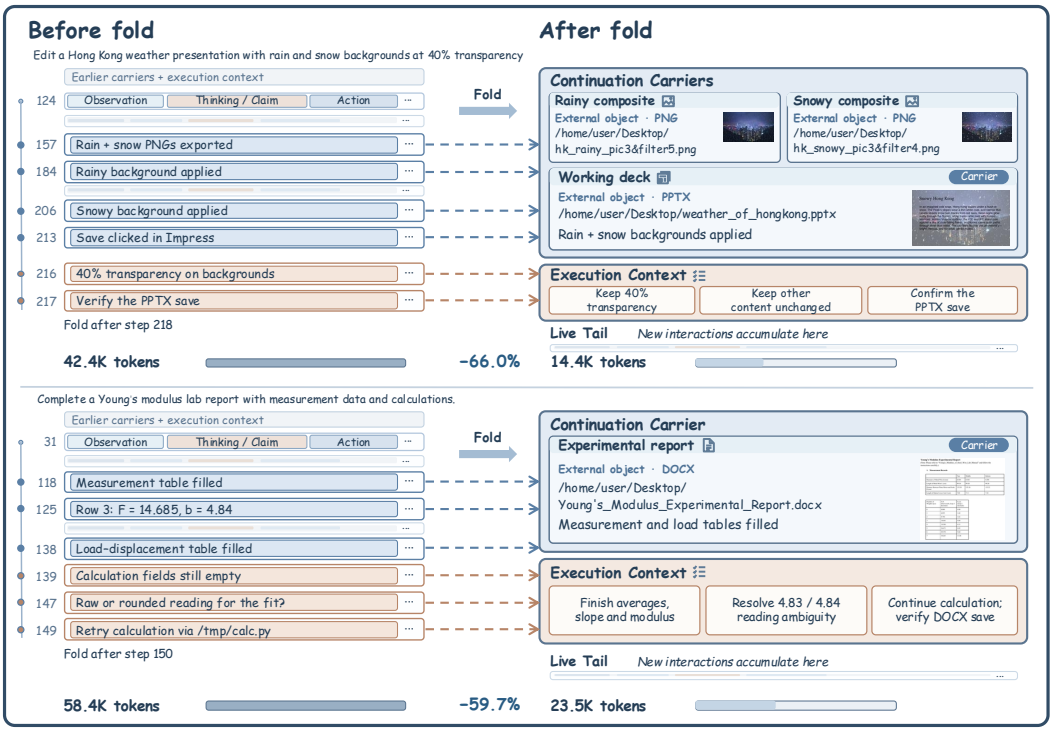


Figure 1: **Context folding with reusable task objects in long-horizon OSWorld2.0 (Yuan et al., 2026).** *Top:* In a slide-editing task, the working presentation and exported images carry completed editing progress, while remaining constraints and save verification stay explicit, reducing active context from 42.4K to 14.4K tokens (66.0%). *Bottom:* In a lab-report task, the working document carries completed edits, while unresolved calculations remain explicit, reducing active context from 58.4K to 23.5K tokens (59.7%).

cies as conditions of use explicit in continuation carriers. *2) how to maintain reusable progress as the task evolves:* Retaining a record does not by itself establish that its content remains supported or its referenced state remains available. CarryOn binds updates to supporting sources and applicable environment checks; across folds, affected records are revised while unaffected records are carried forward. Finally, *what must remain explicit when the corresponding history is retired:* Task objects do not need to capture the goals, constraints, and unresolved issues that guide subsequent execution. CarryOn jointly updates continuation carriers and execution context to preserve this complementary information, while keeping recent interactions in a live tail. Once validated, these updates are committed through a *fold*, replacing the covered interaction history with the resulting object-oriented context view. In this way, CarryOn progressively reorganizes the long-horizon context around reusable task objects while preserving the information required for the agent to continue.

In our experiments, object-oriented context management improves task performance while substantially reducing context cost. Our method, CarryOn, lifts Claude Opus 4.7 from 43.9% to 49.7% (+5.8%) on OSWorld 2.0 for long-horizon computer use and from 49% to 56% (+7%) on BrowseComp-Plus for long-horizon browsing. At the same time, it reduces total token usage by 51.9% and 22.9%, and total cost by 35.9% and 7.1%, on the two benchmarks, respectively. Compressing history alone is insufficient for continued execution: organizing context around reusable progress improves over generic compression, while the full CarryOn runtime further improves over the same principles implemented through repeated summarization. Continuation carriers and execution context play complementary roles, and reusable progress is typically maintained across folds rather than repeatedly reconstructed.

2 Related Work

Context Compression and Selection. Existing methods reduce long-horizon context by compressing, pruning, summarizing, or selectively retrieving prior interactions (Wei et al., 2026). Com-

pression and pruning range from prompt-level compression (Jiang et al., 2023; 2024) to removal of redundant or outdated interactions (Xiao et al., 2026). Summarization methods periodically replace accumulated interactions with compact reasoning states or learned summaries (Wu et al., 2026b; Kang et al., 2026; Ye et al., 2025; Hu et al., 2025), while branch-based execution can return only compact subtask results (Sun et al., 2025). External-history methods instead keep the active context bounded while preserving access to earlier interactions outside the prompt (Packer et al., 2024; Li et al., 2026), and some systems validate compressed representations against subsequent execution to reduce information loss (Chen et al., 2026b).

State Externalization and Context Reconstruction. Complementary approaches represent task progress explicitly, reducing reliance on raw interaction history alone. Some maintain an internal state by integrating prior memory with new observations (Zhou et al., 2025; Liu et al., 2025), while others use structured representations, such as execution-state trees, persistent variables, or task records (Chen et al., 2026a; Tian et al., 2026; Wu et al., 2026a; Ma et al., 2026). Recent systems further externalize state into persistent workspaces and reconstruct the model's working context through snapshots or programmatic selection (Yu et al., 2026; Lin et al., 2026; Zhu et al., 2026). Across these approaches, continuation is primarily organized around compressed interaction history or externally maintained state. Our approach instead makes reusable task objects the primary unit of continuation.

3 METHOD

3.1 FRAMEWORK

Figure 2 illustrates CarryOn, which organizes long-horizon continuation around reusable task objects rather than the interactions that produced them. Reusable objects carry established progress for subsequent inspection and use, while explicit context retains complementary continuation information. Alongside the agent's interaction loop, a *Tracker* identifies candidate reusable progress, which is bound to supporting evidence and, when applicable, checked by a *Grounder*. A *Compiler* jointly updates continuation carriers and execution context. After validation, the update replaces the covered interaction history through a *fold*, while subsequent interactions form a new live tail.

3.2 CONTEXT VIEW

We organize the agent's active context into three complementary components: *continuation carriers*, which expose reusable task objects; *execution context*, which preserves continuation information not captured by those carriers; and a *live tail*, which retains interactions accumulated since the latest fold.

Continuation Carriers. Continuation carriers expose reusable progress to subsequent execution. Each carrier has a stable identity and records the reusable content or its location, together with conditions for reuse. Revisions create successor versions under the same identity, while unaffected carriers retain their records. Depending on where the reusable progress resides, CarryOn supports three forms of carriers:

- **External Objects**: External objects are work products that exist independently in the environment and can be reopened or modified, such as code files, documents, and spreadsheets. Their carrier records reference the underlying objects and describe the progress they capture together with conditions for reuse.
- **Information Products**: Information products capture reusable findings obtained through interaction, such as synthesized information or troubleshooting conclusions. When not materialized as persistent objects, the carriers store the findings directly, with supporting sources and reuse conditions.
- **Situated State**: Situated state captures progress available only within a particular application, session, or workflow. Reusing such progress requires returning to the environment in which the state resides. Its carrier therefore records the environment anchor, the established state, and the scope within which that state remains reusable.

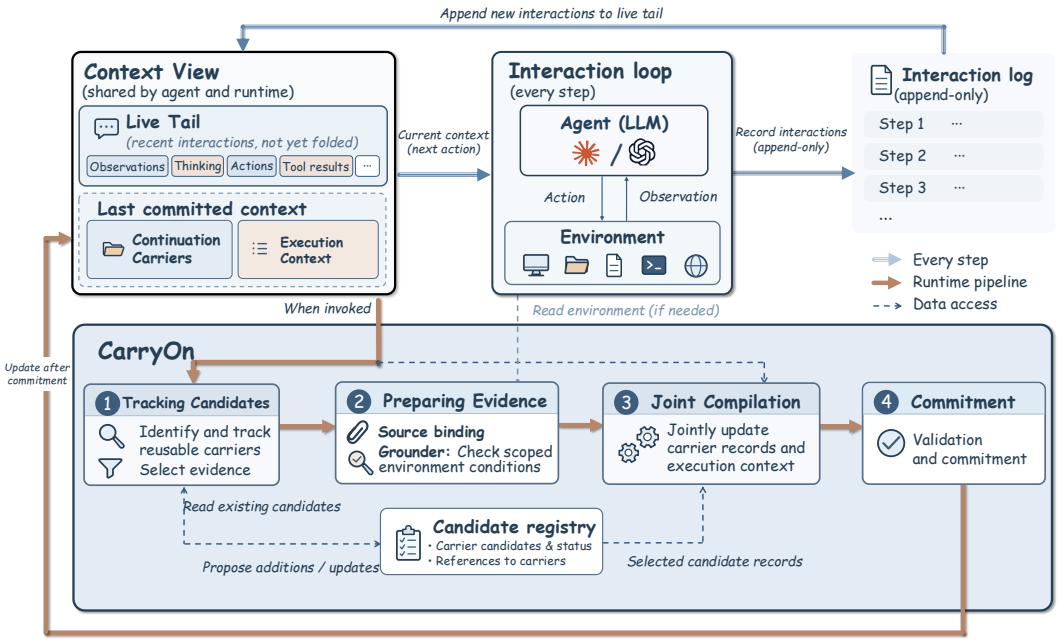


Figure 2: **Overview of the CarryOn framework.** CarryOn maintains an object-oriented context view comprising continuation carriers, execution context, and a live tail. When invoked, it tracks reusable progress, binds supporting evidence, checks relevant environment conditions, and jointly compiles carrier and execution-context updates. A successful fold commits validated updates and retires covered live-tail interactions from active context, while interaction log remains append-only.

These carrier types are complementary rather than mutually exclusive. A task may therefore involve multiple carriers, and the same carrier may be revised as its underlying progress evolves.

Execution Context. Execution context complements continuation carriers by retaining information needed for execution but not encoded in reusable task objects. It records the current focus, unresolved constraints and obligations, and pending work not materialized into a reusable carrier.

Live Tail. Live tail contains the original agent–environment interactions accumulated since the most recent fold. These interactions remain directly available until incorporated into a subsequent fold.

The context view consists of the continuation carriers and execution context produced by the most recent fold, together with the live tail accumulated afterward. At interaction step t, the accumulated interaction history is $H_t = (h_1, \ldots, h_t)$. The most recent fold k incorporates this history through step b_k and establishes the current set of carrier records $A_k = \{a_{k,1}, \ldots, a_{k,n_k}\}$, containing n_k carrier records, together with execution context E_k, while the subsequent interactions form the live tail $L_t = H_t[b_k + 1 : t]$. The context view is therefore

$$V_t = (A_k, E_k) \oplus L_t, \qquad L_t = H_t[b_k + 1 : t]. \tag{1}$$

The operator $\oplus$ is logical composition without prescribing message order. Together, these components present the available task progress, the information needed to use it, and the latest interactions, giving the agent a basis for continuing its work from the task objects.

3.3 Constructing and Updating the Context View

To maintain this context view during execution, CarryOn periodically folds accumulated interactions in the live tail into the maintained representation. Each fold identifies reusable progress, associates it with supporting evidence, and preserves the information still needed for continuation. The update proceeds in four stages: tracking candidates, preparing evidence, joint compilation, and validation and commitment.

Tracking Candidates. CarryOn first identifies progress that may become reusable in subsequent execution. The Tracker maintains candidates, each representing a potential new carrier or an update to an existing one. As execution proceeds, candidates may remain under development across multiple Tracker invocations; once a candidate has reached a state that can support continuation, the Tracker requests its compilation.

A compilation request specifies both what should be compiled and the evidence needed to support it. For external objects and situated state, this includes the relevant environment object together with the state or conditions required for its reuse. For information products, it includes the source material from which reusable information should be retained. Candidates that are not yet ready remain tracked for later folds.

Preparing Evidence. Before compilation, CarryOn first gathers the evidence authorized for each selected candidate. When compilation updates an existing carrier, it also includes the carrier's current record. For information products, CarryOn checks that the selected material is consistent with its recorded sources. For external objects and situated state, it binds the request to the corresponding environment object, and the Grounder performs read-only checks of relevant conditions where available. The resulting input package U_t contains the selected candidates, applicable carrier records, authorized source material, object references, and available check results. This package fixes the record versions and evidence against which proposed carrier edits are later validated. Additional details are given in Appendix A.2.

Joint Compilation. Once the relevant evidence has been prepared, CarryOn must decide both what progress should be captured by carriers and what information must remain explicit for continuation. These decisions are coupled: information represented by a carrier need not also remain in the execution context, whereas unresolved goals, constraints, and unfinished work must survive the fold. CarryOn therefore updates continuation carriers and execution context jointly. For the selected candidates, the Compiler receives their prior carrier records A_k^{sel}, the current execution context E_k, the prepared input package U_t, and a projection $\Pi(L_t)$ of the live tail. Here, $\Pi(L_t)$ presents the recent interactions to the Compiler in chronological text form, distinguishing agent reports, attempted actions, and available tool results. It proposes carrier edits ΔA_t together with an updated execution context $\widehat{E}_{k+1}$:

$$(\Delta A_t, \widehat{E}_{k+1}) = \mathrm{Compile}\big(A_k^{\mathrm{sel}}, E_k, U_t, \Pi(L_t)\big). \tag{2}$$

The proposed edits ΔA_t update the existing carrier set. Carriers outside the edit set keep their current records, while each updated carrier receives a successor version.

The Compiler represents progress according to where it resides. For external objects and situated state, it records what can be reused and under what conditions, while the object or state remains in the environment. For information products, it stores the reusable information in the carrier record itself. The Compiler jointly revises the execution context using the carrier edits and the live tail. Information captured by carriers can be removed from the execution context, while active goals, unresolved constraints, obligations, and unfinished work remain explicit. Using the live tail also allows the Compiler to retain information needed for continuation even when it falls outside the selected candidates.

Validation and Commitment. Before a proposed update changes the active context, CarryOn checks that its carrier edits still refer to the current candidate and carrier versions and are consistent with the evidence authorized for their compilation requests. It also verifies the required identity, reference, and output constraints. A successful fold must introduce at least one carrier change; revising the execution context alone does not retire interaction history. These checks guard the runtime against inconsistent or unsupported updates; they do not verify task correctness.

If these checks fail, the existing context view and live tail remain unchanged. Otherwise, CarryOn atomically applies the carrier edits and commits the revised execution context, advancing the fold through the covered interactions. In the current implementation, a successful fold covers the entire live tail. The resulting carrier set and execution context define the folded portion of the next context view, while subsequent interactions begin a new live tail.

Table 1: **Task performance on OSWorld 2.0.** * means we report the reproducing results using the official benchmark and API. CarryOn improves both binary (+3.7%) and partial (+5.8%) task scores over the same Claude Opus 4.7 (max) host agent while substantially reducing cost per task (-35.9%).

Model / thinking effort	Binary (%) ↑	Partial (%) ↑	Cost/task ($)
GPT-5.5 (xhigh) (Yuan et al., 2026)	13.0	49.5	25.5
Claude Opus 4.8 (max) (Yuan et al., 2026)	18.5	49.3	76.1
Claude Sonnet 4.6 (max) (Yuan et al., 2026)	8.3	41.5	22.3
Claude Sonnet 4.6 (medium) (Yuan et al., 2026)	9.3	33.9	14.4
MiniMax M3 (enabled) (Yuan et al., 2026)	4.6	22.3	2.4
Kimi 2.6 (enabled) (Yuan et al., 2026)	4.6	22.1	6.6
Qwen 3.7-Plus (thinking) (Yuan et al., 2026)	2.8	21.5	3.8
Claude Opus 4.7 (max)*	15.7	43.9	36.8
Claude Opus 4.7 (max) + CarryOn	**19.4**	**49.7**	23.6

Table 2: **Evaluation of resource usage on OSWorld 2.0 and BrowseComp-Plus.** Values are averaged per task. CarryOn reduces total token usage (by 51.9% and 22.9% respectively) and cost (by 35.9% and 7.1% respectively) on both benchmarks.

	OSWorld 2.0		**BrowseComp-Plus**	
Metric	Baseline	**CarryOn**	Baseline	**CarryOn**
Agent tokens (M)	40.10	18.61 ↓53.6%	1.56	1.17 ↓24.6%
Runtime tokens (M)	–	0.69	–	0.03
Total tokens (M)	40.10	**19.30** ↓51.9%	1.56	**1.20** ↓22.9%
Agent cost ($)	36.83	20.51 ↓44.3%	1.89	1.62 ↓14.4%
Runtime cost ($)	–	3.11	–	0.14
Total cost ($)	36.83	**23.62** ↓35.9%	1.89	**1.76** ↓7.1%

4 EXPERIMENT

4.1 EXPERIMENTAL SETUP

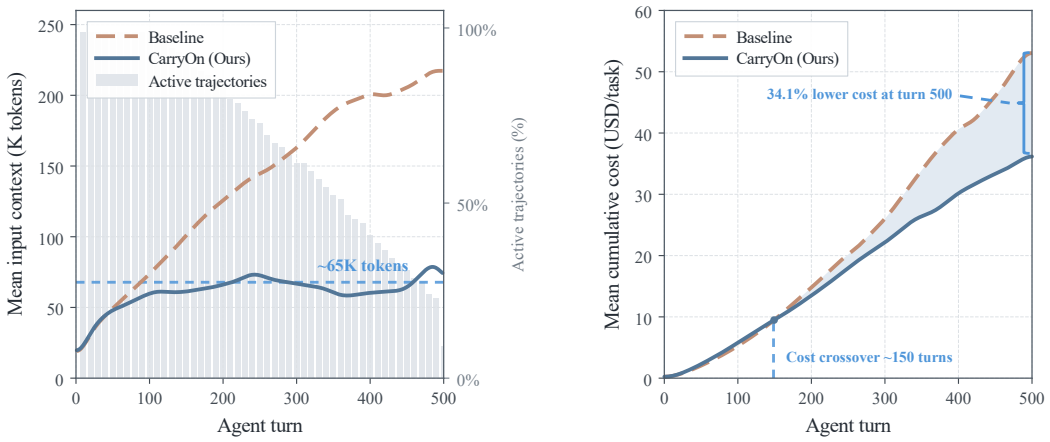
Benchmark. We evaluate CarryOn on OSWorld 2.0 (Yuan et al., 2026) and BrowseComp-Plus (Chen et al., 2025) to assess its performance across different types of long-horizon tasks. We use the full OSWorld 2.0 benchmark as our primary evaluation for its long execution horizons, complex workflows, and task diversity. We additionally evaluate our method on a randomly sampled subset of 100 BrowseComp-Plus questions, which require iterative search across multiple sources and provide a complementary setting for studying context management over long search histories. Benchmark details are provided in Appendix B.1.

Implementation. We use Claude Opus 4.7 for the host agent and CarryOn's Tracker and Compiler. We use a 500-step budget per task on OSWorld 2.0, and the agent retrieves documents using BM25 (Robertson et al., 1995) on BrowseComp-Plus. Detailed experimental configurations and implementation settings are provided in Appendix A.3.

Evaluation Metrics. On OSWorld 2.0, we report partial score and binary score, following the official evaluation protocol. On BrowseComp-Plus, we report answer accuracy, with final responses assessed by Qwen3-32B using the benchmark's official grading prompt. We measure efficiency through token usage per task and cost per task, including both agent and context-management calls.

Table 3: **Task performance on the 100-question subset of BrowseComp-Plus.** CarryOn improves the accuracy over the same Claude Opus 4.7 by 7%.

Model	Retriever	Accuracy (%) ↑
OpenAI GPT-5 (Chen et al., 2025)	BM25	52.00
OpenAI o3 (Chen et al., 2025)	BM25	50.00
OpenAI GPT-OSS-120B (Zhang et al., 2026)	BM25	41.00
Claude Opus 4.7	BM25	49.00
Claude Opus 4.7 + CarryOn	BM25	**56.00**



(a) Input context **(b) Cumulative model cost**

Figure 3: **Context and cost dynamics on OSWorld 2.0.** (a) Mean input context size and (b) mean cumulative model cost over agent turns. CarryOn keeps input context near 65K tokens while the baseline continues to grow, and its cost advantage widens with trajectory length, reaching a 34.1% reduction by turn 500. Gray bars indicate the fraction of baseline trajectories remaining active; costs include context-management overhead.

4.2 Main Result

As shown in Table 1, CarryOn boosts the partial score from 43.9% to 49.7% and binary score from 15.7% to 19.4%, exceeding other frontier models. At the same time, CarryOn reduces total token usage by 51.9% and cost per task by 35.9% as demonstrated in Table 2. These results show that CarryOn reduces long-horizon execution cost without sacrificing task performance.

We further evaluate CarryOn on a random 100-question subset of BrowseComp-Plus. CarryOn raises accuracy from 49.0% to 56.0% over the Claude Opus 4.7 baseline, while the score is also above other frontier models shown in Table 3. At the same time, total token usage decreases by 22.9% and total cost by 7.1%. Together with the OSWorld results, this shows a consistent pattern across computer-use and information-seeking tasks: CarryOn reduces context-processing overhead while maintaining or improving observed task performance.

4.3 Context and Cost Dynamics

We analyze how CarryOn's input context and cumulative cost evolve over the course of long-horizon execution on OSWorld 2.0. As shown in Figure 3(a), the baseline context grows steadily with trajectory length, eventually exceeding 200K tokens, whereas CarryOn's mean input context stabilizes at approximately 65K tokens after the initial growth phase. This indicates that our method maintains a consistently compact context for the agent, regardless of how long execution continues.

Table 4: **Comparison of different methods on a shared 40-task subset of OSWorld 2.0.** Post-fold text length measures the context retained after each compression or fold. CarryOn achieves the highest partial score (55.93%) without retaining the most text.

Method	Text Length (k chars)	Binary (%) ↑	Partial (%) ↑
Baseline	–	22.50	50.32
Sliding Window	12.08	15.00	37.41
Generic Summary (GS)	17.60	15.00	44.94
CarryOn-Guided Summary (CGS)	23.94	20.00	50.93
CarryOn (ours)	21.54	**22.50**	**55.93**

The corresponding cost dynamics are shown in Figure 3(b). CarryOn initially incurs additional cost from context-management calls, but this overhead is offset as repeated agent calls operate on a smaller context. The cumulative-cost curves cross before turn 200, after which the gap continues to widen; by turn 500, CarryOn reduces cumulative cost by 34.1%, including context-management overhead. Thus, the savings arise progressively as longer trajectories avoid repeatedly processing an ever-growing interaction history.

4.4 Comparison with Context Management Strategies

We examine the proposed object-oriented organization against conventional history compression, and compare the full CarryOn runtime with a summary-based implementation of the same principles. On the same random 40-task subset of OSWorld 2.0, we compare: (i) **Baseline**, the default OSWorld 2.0 agent setting without additional context compression; (ii) **Sliding Window**, which retains only recent messages; (iii) **Generic Summary (GS)**, which summarizes earlier history into a continuation note; and (iv) **CarryOn-Guided Summary (CGS)**, which applies our object-oriented organizing principles within a textual summary. Sliding Window and GS follow the FIFO truncation and prompting-based summarization baselines of Kang et al. (2026). All methods use the same host model (Claude Opus 4.7), the same trigger condition, and the same step budget.

As shown in Table 4, aggressive history compression does not improve performance by itself: Sliding Window and GS retain the shortest contexts but reduce both binary and partial scores relative to the baseline. Applying the same organizing principles through CGS recovers much of this loss, reaching a 50.93% partial score. Full CarryOn further increases the partial score to 55.93% while matching the baseline binary score of 22.50%. Compared with CGS, CarryOn achieves 5.00% higher partial score with about 10% shorter post-fold text. Together, these results show that our object-oriented organization principle is effective even when applied within a textual summary, while the full CarryOn runtime yields further improvements, demonstrating the additional benefit of maintaining and managing this structured state explicitly during execution.

4.5 Analysis

Carriers and the execution context are complementary. We conduct an ablation study on the shared 40-task OSWorld 2.0 subset by hiding either continuation carriers or execution context from the host agent. As shown in Table 5, removing execution context reduces binary score from 22.50% to 17.50% and partial score from 55.93% to 44.00%, while the number of agent calls changes little. The carriers still expose completed progress, but the agent loses complementary goals, constraints, and unresolved requirements needed to finish the task. Removing continuation carriers reduces partial score to 44.62%, while increasing agent calls by 23.8% and cost by 22.3%. Without direct access to established progress, the agent must recover or reproduce earlier results before continuing. This behavior is visible in trajectory-level examples in Appendix C.2. Together, the two ablations show that carriers and execution context preserve distinct, complementary information for continuation.

Continuation carriers preserve and update progress across folds. To examine how carrier records evolve as execution progresses, we compare them before and after each committed fold in CarryOn's OSWorld 2.0 trajectories. Figure 4(a) shows that 80% of carrier records are retained or updated. This pattern reflects an evolving representation of task progress that existing records

Table 5: **Context-view ablation on the shared 40-task subset of OSWorld 2.0.** Each ablation hides one component from the host agent while preserving CarryOn's internal runtime state. Removing either component substantially lowers partial score, while removing continuation carriers also increases agent calls and cost.

Method	Agent Calls	Cost ($) ↓	Binary (%) ↑	Partial (%) ↑
w/o Execution Context	274.33	18.33	17.50	44.00
w/o Continuation Carriers	349.45	25.14	**25.00**	44.62
CarryOn (ours)	282.25	20.55	22.50	**55.93**

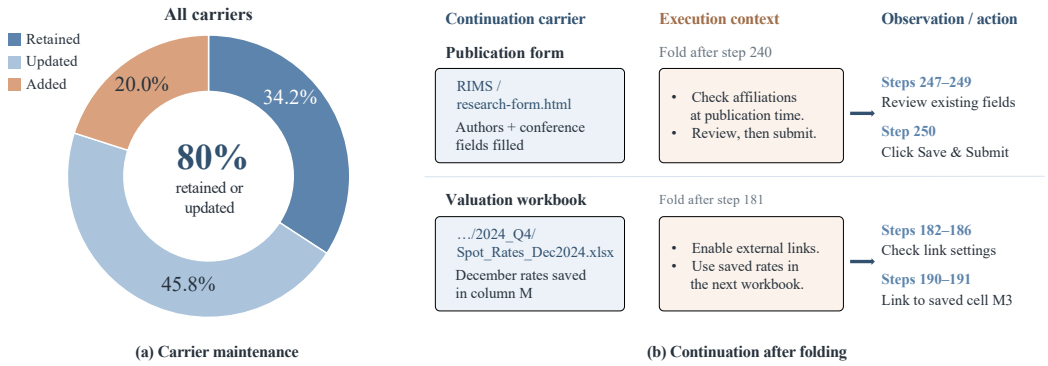


Figure 4: **Carrier persistence and continuation after folding.** (a) Across committed folds on OSWorld 2.0, 80% of carrier observations are retained or updated rather than newly added. (b) Representative trajectories show the agent continuing directly from reusable task state after earlier interactions have been folded, without repeating the work that produced it.

remain available across folds, while updates incorporate subsequent work on the same task objects. CGS follows the same organizing principles but maintains this information by generating a complete replacement summary at each fold. Despite retaining more post-fold text, CGS achieves a lower partial score than CarryOn (50.93% vs. 55.93%) as shown in Table 4. Explicitly maintaining records as task objects evolve may help the agent build on prior progress, providing one possible explanation for CarryOn's higher score.

Context views enable continuation from current object state. We inspect whether the agent continues from progress whose producing interactions have already been folded. Figure 4(b) shows cases where the agent directly reuses the resulting task state, such as submitting a populated form or using previously saved exchange rates without repeating the earlier work. The 23.8% increase in agent calls shown in Table 5 when carriers are hidden further indicates that access to established progress reduces recovery work.

5 Conclusion

We introduce CarryOn, a context runtime that organizes long-horizon agent context around reusable task objects. Our method combines continuation carriers that expose reusable progress, execution context that preserves complementary goals, constraints, and unresolved issues, and a live tail of recent interactions. As the task progresses, CarryOn jointly updates continuation carriers and execution context to reflect evolving task state and remaining requirements, progressively replacing covered interaction history with an updated context view. On OSWorld 2.0, CarryOn raises partial score from 43.9% to 49.7% while reducing total token usage by 51.9% and nominal model cost by 35.9%, including runtime overhead. On BrowseComp-Plus, it increases accuracy from 49% to 56% with lower token usage and cost. Together, these results support object-oriented context management for long-horizon agents, where task objects carry established progress and explicit context evolves with what is still needed to move forward.

AI USE STATEMENT

We used generative AI tools to improve the clarity, fluency, and overall coherence of the manuscript, as well as to assist with minor code editing and debugging. These tools were used solely for support and did not replace the authors' scientific judgment or decision-making. All AI-assisted text was carefully reviewed and revised by the authors, and any AI-assisted code was independently inspected and tested for correctness before being used in our experiments.

ETHICS STATEMENT

This work does not involve human subjects, personally identifiable information, or sensitive private data. The datasets and evaluation protocols used in our experiments follow their respective licenses and intended research purposes. We are not aware of any immediate ethical concerns arising from the methodology or experiments presented in this work. Nevertheless, as with many machine learning methods, potential misuse or unintended applications may arise in downstream settings, and we encourage responsible use with appropriate consideration of privacy, fairness, security, and other application-specific risks.

REPRODUCIBILITY STATEMENT

We use publicly available benchmarks and language models accessible through APIs or released weights. The runtime pipeline and implementation details are described in Appendix A.1 and A.3, with model configurations and runtime settings in Appendix A.3. Benchmark and task selection, evaluation protocols, and resource accounting are detailed in Appendix B.1. The runtime and baseline prompt designs are presented in Appendix D. Together, these descriptions support the reproducibility of our method and experimental evaluation.

REFERENCES

Yaoqi Chen, Haibin Lai, Yuru Feng, Chuyu Han, Qianxi Zhang, Baotong Lu, Menghao Li, Xinjiang Wang, Zhirui Wang, Shusen Xu, Zengzhong Li, Zewen Jin, Hao Wu, Cheng Li, and Qi Chen. Beyond semantic organization: Memory as execution state management for long-horizon agents, 2026a. URL https://arxiv.org/abs/2606.06090.

Zhuofu Chen, Rui Pan, Yinwei Dai, and Ravi Netravali. Slipstream: Trajectory-grounded compaction validation for long-horizon agents, 2026b. URL https://arxiv.org/abs/2605.08580.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent, 2025. URL https://arxiv.org/abs/2508.06600.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. HiAgent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32779–32798, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.1575. URL https://aclanthology.org/2025.acl-long.1575/.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMLingua: Compressing prompts for accelerated inference of large language models. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 13358–13376, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.825. URL https://aclanthology.org/2023.emnlp-main.825/.

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 Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1658–1677, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.91. URL https://aclanthology.org/2024.acl-long.91/.

Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. Memory OS of AI agent. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 25961–25970, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1318. URL https://aclanthology.org/2025.emnlp-main.1318/.

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, 2026. URL https://arxiv.org/abs/2510.00615.

Xiaochuan Li, Ryan Ming, Meng Chu, Shuai Shao, Rong Jin, and Chenyan Xiong. Acm: Agentic context management for long horizon tasks, 2026. URL https://arxiv.org/abs/2607.23809.

Yin Lin, Elaine Ang, Erkang Zhu, Bolin Ding, and Jingren Zhou. Context as an environment: Programmatic context management for long-horizon agents, 2026. URL https://arxiv.org/abs/2608.21690.

Shukai Liu, Jian Yang, Bo Jiang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents, 2025. URL https://arxiv.org/abs/2512.22087.

Ziyu Ma, Hailang Huang, Shun Zou, Yong Wang, Shidong Yang, Yiming Hu, Fei Wei, and XiangXiang Chu. Longhorizon-harness: Advancing long-horizon agents for real-world tasks, 2026. URL https://arxiv.org/abs/2608.01964.

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

Stephen Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford. Okapi at trec-3. In *Overview of the Third Text REtrieval Conference (TREC-3)*, pp. 109–126. Gaithersburg, MD: NIST, January 1995. URL https://www.microsoft.com/en-us/research/publication/okapi-at-trec-3/.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon llm agent via context-folding, 2025. URL https://arxiv.org/abs/2510.11967.

Shizuo Tian, Hao Wen, Yuxuan Chen, Jiacheng Liu, Shanhui Zhao, Guohong Liu, Ju Ren, Yunxin Liu, and Yuanchun Li. Agentprog: Empowering long-horizon gui agents with program-guided context management, 2026. URL https://arxiv.org/abs/2512.10371.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory, 2024. URL https://arxiv.org/abs/2409.07429.

Lei Wei, Xiao Peng, Tt, Guannan Zhang, Chenhao Jiang, Hongyu Li, Lanbo Lin, Yuanwu Xu, Jiayao Liu, Kesu Wang, and Bin Wang. PACE: Predictive adaptive context extraction for long-horizon LLM agents. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 27184–27199, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.1252. URL https://aclanthology.org/2026.acl-long.1252/.

Wenyi Wu, Sibo Zhu, Kun Zhou, Aayush Salvi, Zixuan Song, and Biwei Huang. Structagent: Harness long-horizon digital agents with unified causal structure, 2026a. URL https://arxiv.org/abs/2607.11388.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, Minhao Cheng, Shuai Wang, Hong Cheng, and Jingren Zhou. Resum: Unlocking long-horizon search intelligence via context summarization, 2026b. URL https://arxiv.org/abs/2509.13313.

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), July 2026. doi: 10.1145/3797084.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024. URL https://arxiv.org/abs/2404.07972.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents, 2025. URL https://arxiv.org/abs/2502.12110.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-Agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Neural Information Processing Systems*, 2024. doi: 10.52202/079017-1601. URL https://mlanthology.org/neurips/2024/yang2024neurips-sweagent/.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Siheng Chen, Jingren Zhou, and Yong Jiang. Agentfold: Long-horizon web agents with proactive context management, 2025. URL https://arxiv.org/abs/2510.24699.

Lu Yi, Runlin Lei, Liuyi Yao, Yuexiang Xie, Yuyang Li, Wenhao Zhang, Zhewei Wei, Yaliang Li, and Jian-Yun Nie. Learning agent-compatible context management for long-horizon tasks, 2026. URL https://arxiv.org/abs/2605.30785.

Chenglin Yu, Yuchen Wang, Songmiao Wang, Hongxia Yang, and Ming Li. Infiagent: An infinite-horizon framework for general-purpose autonomous agents, 2026. URL https://arxiv.org/abs/2601.03204.

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, Dunjie Lu, Haikong Lu, Qi Zhen, Xinyuan Wang, Jiaqi Deng, Yuhao Yang, Cheng Chen, Boyuan Zheng, Alex Su, Xiao Yu, Hao Zou, Saaket Agashe, Xing Han Lu, Manpreet Kaur, Zhengyang Qi, Vincent Sunn Chen, Frederic Sala, Dayiheng Liu, Junyang Lin, Zhou Yu, Yu Su, Siva Reddy, Xin Eric Wang, Peng Qi, Tianbao Xie, and Tao Yu. Osworld 2.0: Benchmarking computer use agents on long-horizon real-world tasks, 2026. URL https://arxiv.org/abs/2606.29537.

Haoxiang Zhang, Qixin Xu, Zhuofeng Li, Lei Zhang, Pengcheng Jiang, Yu Zhang, and Julian McAuley. Masking stale observations helps search agents – until it doesn't: A regime map and its mechanism, 2026. URL https://arxiv.org/abs/2606.00408.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Jinhua Zhao, Bryan Kian Hsiang Low, and Paul Pu Liang. Mem1: Learning to synergize memory and reasoning for efficient long-horizon agents, 2025. URL https://arxiv.org/abs/2506.15841.

Chiwei Zhu, Benfeng Xu, Mingxuan Du, Shaohan Wang, Xiaorui Wang, Zhendong Mao, and Yongdong Zhang. FS-researcher: Test-time scaling for long-horizon research tasks with file-system-based agents. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6353–6373, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.288. URL https://aclanthology.org/2026.acl-long.288/.

A METHOD AND IMPLEMENTATION DETAILS

A.1 CONTEXT REPRESENTATION AND UPDATE PROCEDURE

CarryOn maintains an append-only interaction log, a candidate registry, versioned carrier records, and execution-context revisions. The committed carrier records and execution context define the folded portion of the context view; subsequent interactions form the live tail.

Candidate Records. Each candidate has a runtime-assigned identifier, a provisional carrier type, a short description, and references to the interactions supporting it. Its identity fields specify the associated object or subject, and it may refer to an existing carrier whose progress is being updated. The registry records when the candidate was first observed and last revised. Candidates can persist across Tracker invocations, allowing progress to accumulate before compilation. Registering or revising a candidate does not itself trigger a fold.

Carrier Records and Versions. A committed carrier record contains its identity, carrier type, reusable content or object reference, conditions for continuation, applicability scope, provenance, and version information. External-object identities are derived from their normalized locations. Information-product identities combine a representation type with a structured subject, while situated-state identities combine a state domain with the subject of an environment anchor. These identity fields distinguish the object being tracked from its changing contents.

Updating an existing carrier creates a successor record linked to the previous version. The successor becomes current, while earlier versions remain stored for provenance. Carriers not changed by a fold retain their current records. For information products, reusable content is stored as claims with identifiers, propositions, optional values, and supporting source references. Revisions can add, modify, or retract claims while carrying forward unaffected claims.

Execution Context. Execution context is stored as a revision associated with a committed fold. It contains the current focus, continuity information needed to resume work, open obligations, active constraints, and immediate continuation instructions. Entries retain source attribution. The Compiler revises this information against the proposed carrier updates, preserving unresolved requirements and removing items that have been resolved or are now represented by carriers.

Update Procedure. At each invocation, the runtime performs the following operations:

1. The Tracker reads recent interactions and the candidate registry. The runtime applies admissible candidate updates and resolves any compilation requests to their candidate records.
2. For each admitted request, the runtime binds the candidate snapshot, authorized prior carrier records, and supporting sources. Applicable environment checks contribute additional evidence. The resulting package fixes the records and evidence used for compilation. If no request is admitted, the invocation ends without a fold.
3. The Compiler proposes carrier additions or revisions together with a new execution context. Its input also includes the current execution context and a chronological text projection of the live tail, so continuation information outside the selected candidates can be retained.
4. The runtime checks identity and reference consistency, source authorization, output constraints, and whether the compilation input records remain current. A valid proposal must contain an actual carrier change. Acceptance atomically commits the carrier updates and execution-context revision and advances the fold boundary through the entire live tail. Rejection leaves these committed components and the live tail unchanged.

Previously accepted candidate updates remain available after a rejected compilation proposal. The original interactions remain in the log after a successful fold. Grounder rules and installation of the committed view are described in Appendices A.2 and A.3, respectively.

A.2 GROUNDER CHECKS AND VERDICT RULES

CarryOn aligns compilation requests with their supporting evidence. Information products undergo source-reference and excerpt checks. For object-bound requests, the Grounder performs scoped,

Table 6: Read-only environment checks in the OSWorld 2.0 integration.

Check	Observation
File existence	Whether the file is present at its bound path.
Change from initial state	Whether the file-content hash differs from its registered initial snapshot.
File size	Whether the file is empty or nonempty.
Document structure	Whether a `docx`, `pptx`, or `xlsx` file has a valid document container.
Document differences	Changes in document text units, added or removed parts, and placeholder-like content.
Instruction text matching	Presence or absence of a text span copied from the public task instruction.

read-only environment checks. Table 6 summarizes the implemented checks in the OSWorld 2.0 integration.

Request binding and evidence records. Each check is bound to a candidate identity and revision, an object reference, a predicate, and an expected value before execution. The runtime records the observation and its provenance, and validates the binding against the current candidate before using the result.

Verdicts and compilation. A condition is `supported` when the observation establishes the expected value, `contradicted` when it establishes an incompatible value, and `unknown` when the evidence cannot decide. A request is supported only when all declared conditions are supported. Any contradiction makes it contradicted. Otherwise, an unresolved condition makes it unknown.

Supported object-bound requests may proceed to compilation. Unsupported checks or unknown results leave the affected candidate uncompiled, while other admissible requests may proceed. If the interaction evidence supports an independently reusable information product, the Tracker may register a separate candidate through the source-validation route.

These checks serve as supporting consistency safeguards within the runtime, helping the agent identify explicit contradictions and outstanding task requirements.

A.3 Agent Integration and CarryOn Settings

Agent integration. CarryOn runs alongside the agent through a benchmark-specific adapter. The adapter records interactions, supplies their textual representations and exposed thinking summaries to the runtime, and installs committed carrier records and execution context before the next agent call. Within each benchmark, the baseline and CarryOn use the same agent model, environment-action interface, and task budget. CarryOn additionally supplies the context-management interfaces and feedback described below.

Model configuration. We use Claude Opus 4.7 for the agent, Tracker, and Compiler, without additional training. Table 7 lists their output limits and execution budgets. On OSWorld 2.0, the agent uses adaptive thinking at maximum effort; Tracker and Compiler requests contain no explicit thinking or effort override. On BrowseComp-Plus, the agent uses adaptive thinking with the provider's default effort, the Tracker uses adaptive thinking at high effort, and the Compiler has thinking disabled. Both adapters expose the provider's summarized thinking output to the runtime, marking it as agent-generated information rather than environment observations.

Runtime triggering. On OSWorld 2.0, the Tracker is normally invoked when interaction text accumulated since the previous scan or fold exceeds an estimated 6,000 tokens. This estimate is the character count of the runtime's text projection divided by four and excludes the agent's thinking summaries. These thresholds measure text volume, not predicted monetary savings.

On BrowseComp-Plus, scan scheduling uses growth in the agent's recorded prompt size, including cached input. The initial trigger requires 120,000 tokens of growth plus a 1,024-token margin and newly ingested search results. Consecutive scans that produce no fold double the token thresholds, up to six doublings; a successful fold resets this backoff.

Table 7: Agent and CarryOn settings. Output limits are measured in tokens per model call.

Setting	OSWorld 2.0	BrowseComp-Plus
Agent execution budget	500 steps	60 turns
Agent output limit	16,000	32,000
Tracker output limit	2,048	8,192
Compiler output limit	8,192	8,192
Compilation requests per scan	4	4

OSWorld integration. The adapter preserves the initial task message and installs a serialized checkpoint containing carrier records and execution context. It retains the contiguous suffix of complete message groups needed to preserve screenshots kept by the agent's native image filter, together with the pending assistant message needed for protocol continuity. The native filter is configured with a recent-image threshold of 10, and screenshot eviction remains under its control. Retained tool calls, results, and signed thinking blocks preserve their protocol relationships. A fixed system instruction identifies the checkpoint as runtime-supplied context while treating its contents as task data.

BrowseComp-Plus integration. The agent uses BM25 search with up to five results per call and 512-token snippets, without full-document retrieval. The runtime receives bounded textual projections of search results, including document identifiers, ranks, scores, and the first 160 characters of each snippet, alongside agent text and exposed thinking summaries. After a fold, pending search results remain available to the next agent call.

Installation and recovery. Before installing a new context, the adapter snapshots the agent's message state. A failed installation restores this snapshot and the previous active-checkpoint bookkeeping. If restoration is verified, execution continues with further runtime intervention disabled. If the previous state cannot be verified, the run stops with an error. This separates commitment of a runtime representation from its successful installation in the agent's context.

B Experimental Details

B.1 Benchmark and Task Selection

OSWorld 2.0. OSWorld 2.0 contains long-horizon tasks spanning seven professional domains and 21 subcategories (Yuan et al., 2026). Its workflows combine desktop applications and web services, requiring cross-source reasoning, state tracking, and coordinated operations on multiple task outputs. The benchmark reports a median estimated human completion time of approximately 1.6 hours. The duration and diversity of these workflows make it suitable for evaluating context management over extended task execution. We use the `v2026.08.08` release.

Our main OSWorld 2.0 evaluation covers the full evaluation set, with both the baseline agent and CarryOn independently run by us. For the additional context-management comparisons and ablation experiments, we sample a shared subset of 40 tasks, with the following task IDs:

> 001, 002, 003, 004, 005, 006, 008, 009, 012, 013, 014, 018, 030, 031, 038, 039, 040, 046, 047, 048, 049, 057, 059, 061, 063, 064, 066, 070, 071, 073, 076, 079, 082, 083, 087, 089, 093, 094, 095, 102.

BrowseComp-Plus. BrowseComp-Plus contains 830 challenging questions over a fixed corpus of 100,195 documents (Chen et al., 2025). Its questions require iterative search and reasoning across multiple sources, producing growing histories of retrieved evidence and intermediate analysis. This setting allows us to evaluate context management under sustained information accumulation. We use BM25 retrieval and evaluate both the baseline and CarryOn on the same 100-question subset, with the following question IDs:

> 11, 12, 15, 18, 36, 39, 209, 223, 228, 231, 233, 237, 240, 241, 246, 249, 251, 255, 256, 264, 266, 268, 480, 484, 485, 487, 495, 496, 500, 501, 502, 505, 509, 513, 515, 517, 519,

Table 8: Nominal token prices used for resource accounting.

Token category	Price ($/M tokens)
Uncached input	5.00
Cache read	0.50
Cache write (5 minutes)	6.25
Cache write (1 hour)	10.00
Output	25.00

527, 528, 529, 530, 537, 544, 546, 549, 553, 555, 562, 770, 771, 773, 774, 776, 784, 787, 793, 796, 798, 801, 802, 805, 809, 819, 820, 821, 822, 827, 828, 833, 844, 850, 852, 856, 863, 869, 872, 873, 986, 991, 998, 1005, 1015, 1016, 1020, 1023, 1025, 1029, 1030, 1033, 1036, 1037, 1038, 1039, 1040, 1042, 1046, 1049, 1060, 1063, 1066.

B.2 Previously Reported Results

OSWorld 2.0. We include the published 500-step results from the original OSWorld 2.0 evaluation (Yuan et al., 2026), retaining its reported scores and estimated model cost per task. All models receive screenshot observations through the benchmark's official agent implementations. Claude models use their native computer-use interface, whereas all other models generate PyAutoGUI actions. GPT-5.5 executes actions in batches, while the reported Claude, MiniMax, Kimi, and Qwen results use the same single-action setting as our evaluation.

BrowseComp-Plus. We compute external reference results on the same 100 questions used in our experiments. For GPT-5 and o3, we use the benchmark authors' publicly released BM25 runs (Chen et al., 2025) and grade their available final responses with Qwen3-32B using the benchmark's official grading prompt. For GPT-OSS-120B, we use the BM25 configuration released by Zhang et al. (2026) and aggregate the authors' published correctness labels over the same subset.

B.3 Evaluation and Resource Accounting

Accounting scope. We report resource usage as per-task averages. For each selected task attempt, we accumulate recorded usage from both the agent and auxiliary context-management calls. For CarryOn, the latter includes the Tracker and Compiler. For summarization-based methods, it includes summarizer calls. We report agent usage, auxiliary usage, and their combined totals, so the total captures the resources consumed by both task execution and context management.

Token accounting. Token counts come from API usage records. For task i, let $\mathcal{C}_i$ contain its recorded agent and context-management calls. Total token usage is

$$T_i = \sum_{j \in \mathcal{C}_i} (I_j + R_j + W_j + O_j),$$

where I_j denotes uncached input tokens, R_j cache-read tokens, W_j cache-write tokens, and O_j output tokens. Cache writes include both five-minute and one-hour entries. When usage records provide an overall cache-write count and its lifetime breakdown, the breakdown partitions that count and is not added again. Reasoning tokens already included in output usage are also counted once.

Token usage is cumulative across requests: input processed again in a later call is counted again, including input served from the cache. This measures the total tokens processed during execution. We report token usage in millions, with $1\,\mathrm{M} = 10^6$ tokens.

Cost accounting. For our Claude Opus 4.7 runs, we calculate nominal API-equivalent costs using the fixed rates in Table 8. Each call's cost is the sum of its token counts multiplied by the corresponding category rates, and task cost is the sum across calls. The same rates apply to the agent and CarryOn's auxiliary models.

Cached input remains part of total token usage but receives its corresponding cache price. Consequently, reductions in token usage and cost can differ depending on the proportions of uncached

input, cache reads, cache writes, and output. Reported totals include context-management overhead and exclude evaluator, user-simulator, and infrastructure costs. They cover the recorded usage of the selected task attempts.

Aggregation. For each method, we average task totals over its reported task set. Relative reductions in token usage and model cost are computed from these means:

$$\text{Reduction}(x) = 100 \left(1 - \frac{\bar{x}_{\text{CarryOn}}}{\bar{x}_{\text{baseline}}}\right) \%,$$

where $\bar{x}$ is mean token usage or model cost per task. All calculations use unrounded values, with rounding applied only to the displayed results.

C Supplementary Results and Analysis

C.1 Statistical Procedures for Context and Carrier Analysis

Context and cost dynamics. Figure 3 uses the per-call usage traces of all baseline and CarryOn trajectories included in the resource comparison. An agent turn denotes a metered agent-model response, including retries and terminal-review responses with recorded usage. For method m, let $L_{m,i}$ denote the number of such responses in trajectory i, and define the active set at turn t as $\mathcal{A}_m(t) = \{i : L_{m,i} \geq t\}$. We compute

$$\bar{P}_m(t) = \frac{1}{|\mathcal{A}_m(t)|} \sum_{i \in \mathcal{A}_m(t)} P_{m,i,t}, \qquad \bar{C}_m(t) = \frac{1}{|\mathcal{A}_m(t)|} \sum_{i \in \mathcal{A}_m(t)} C_{m,i}(t).$$

Here, $P_{m,i,t}$ is the input token count reported by the API, including uncached input, cache reads, and cache creation. Screenshot tokens remain included. $C_{m,i}(t)$ accumulates agent costs through turn t and runtime costs assigned to the same boundary, using the accounting rules in Appendix B.3. Gray bars show the fraction of baseline trajectories that remain active at each turn, sampled every ten turns.

Curve smoothing and annotations. We first compute the per-turn means and then apply a Gaussian filter with a standard deviation of 12 turns, a kernel truncated at four standard deviations, and symmetric reflection at the boundaries. Input-context curves use the complete recorded series before the displayed range is restricted to 500 turns. Cost curves are smoothed over turns 0–500, with the unsmoothed series initialized at zero cumulative cost. Gray bars remain unsmoothed. Figure annotations are calculated from the raw data. Shading shows the difference between the two cost curves.

Post-fold text length. For Table 4, we measure the textual context immediately after each installed fold, before the next observation is appended. All methods use the same text projection, including task instructions, retained records or summaries, interaction text, returned thinking summaries, and role/action/observation labels. Image data, signatures, fixed system prompts, and tool schemas are excluded. Length is measured in thousands of Unicode characters. We first average lengths across folds within each task and then average equally across tasks with at least one fold. The baseline has no post-fold measurement. We use this measure together with task performance to compare how much text each method retains and how well the agent performs.

Carrier persistence across folds. For Figure 4(a), we compare current carrier records before and after each committed and installed fold. We examine all 494 committed and installed folds across the CarryOn trajectories included in this analysis. Carrier identities are tracked across record versions using registered identities and successor relations. A carrier is *retained* when its current record version is unchanged, *updated* when a successor version is committed, and *added* when it has no predecessor. Each carrier contributes one observation per fold in which it is present afterward. The same carrier can therefore contribute across several folds, and initial creations are included. Pooling these observations gives 278 retained, 372 updated, and 163 added observations, for 813 in total. The retained-or-updated proportion is $650/813 \approx 79.95\%$, rounded to 80% in the figure. These percentages describe record continuity across folds.

Table 9: **Continuing from a populated form (Task 009).** Selected steps from a main-experiment CarryOn trajectory. Green highlights continuation from the existing form. Entries are condensed from the recorded trajectory; the submission confirmation is also visible in the saved screenshot.

Step	Recorded state or agent statement	Action
203	The agent reports that all four external authors have been entered.	Continues through the form.
205	Publication metadata is entered alongside the author list.	Types the NeurIPS conference dates.
F4 after 240	Carrier: entered authors and publication metadata. EC: classify authors by affiliation at publication time.	Installs the updated context view.
247	Reviews the existing author classifications against the guidelines.	Presses Escape after the review.
249	The publishing status, peer-review status, and attached PDF remain in the form.	Returns to the form without re-entering the authors.
250	The existing form is ready for submission.	Clicks Save & Submit; the page confirms successful submission.
253	The submitted paper appears in the Research Output list.	Ends the run.

The earlier author and conference-entry steps precede the retained interaction window (221–240). The agent also reopens the guidelines after the fold.

C.2 Continuation Cases and Ablation Behaviors

We inspect recorded trajectories to show how execution continues after a fold and how the agent behaves when carrier records are hidden. The tables present selected steps, the information retained at the fold, and subsequent actions.

Continuing from established progress. Table 9 shows a publication form that remains populated across a fold. The agent checks the existing entries and submits the form without re-entering the authors. The saved screenshot confirms submission, and the paper subsequently appears in the Research Output list.

Returning to previously acquired information. Table 10 compares two runs of a prize-distribution task. In the full CarryOn run, the agent continues filling the form after the fold. In the carrier-masking run, it has already established the item-price mapping and used it to construct a plan. After the fold, it questions whether the prices have been confirmed, extracts the PDF again, and reopens the catalogue to reconstruct the mapping. This provides a concrete example of repeated information acquisition when carrier records are withheld.

D Prompts

The Tracker and Compiler use separate system prompts. The runtime selects an environment, information-product, or composite Compiler route according to the authorized candidates. For presentation, prompt terminology is standardized to match the paper: continuation carriers (carriers for short), the host agent, and the runtime scheduler. An environment-check specification lists conditions for the Grounder to check; selected evidence forms part of the compilation input package U_t. Internal policy labels, development annotations, and serialization-version tags are omitted. JSON field names and categorical values retain their implementation spelling.

Runtime inputs. The Tracker receives the task and current committed records, the live tail, and a current-state block containing candidates and runtime feedback. The Compiler additionally receives the authorized compilation requests and their source bindings. These blocks are populated for each invocation.

Summary inputs. GS and CGS receive the current visible conversation, including the task, any earlier summary, and recent messages. The input preserves message roles, tool actions, returned

Table 10: **Continuing a plan versus reacquiring its inputs (Task 013).** Selected steps from the full CarryOn and carrier-masking runs. Green marks continuation of the retained prize plan; red marks renewed uncertainty and repeated acquisition of the price mapping. Quoted spans are exact excerpts from recorded agent reasoning summaries; other descriptions are condensed.

Step	Recorded state or agent statement	Action
CarryOn		
30	Identifies all 15 catalogue items and their prices, then develops a prize-distribution plan.	Closes the catalogue viewer.
F2 after 57	Carrier: item-price mapping and chosen tier assignments. EC: enter the third-prize total, then complete the participation tier.	Both records remain visible to the agent.
58	Continues entering the planned third-prize total.	Types 125 in the total field.
60	Moves on to the planned participation-prize items.	Selects Chocolate Box.
w/o Carrier		
35	"Now I have all the pricing details I need"; selects a plan with tier totals 126/125/125/124.	Begins entering the plan.
F2 after 54	The runtime compiles the price mapping and plan, but carrier records are hidden from the agent. EC retains the tier totals and next entry step.	Installs the masked view.
55	"I don't actually have the item prices confirmed".	Types 125 in the total field.
56–57	Returns to "checking the PDF again for the item-to-price mapping".	Opens a terminal.
59–60	Extracts the source catalogue again.	Runs pdftotext on prize-items.pdf.
65–69	Renders the catalogue pages and reopens page 2.	Runs pdftoppm, then xdg-open.
71–72	Reads item-price correspondences from page 2 again.	Zooms in on the reopened catalogue.

observations, visible images, and returned reasoning summaries where available. They return a replacement summary as plain text.

The following tables present selected instructions from the main system prompts with the editorial changes described above; GS is reproduced in full. The marker `[...]` indicates omitted passages.

Table 11: **Tracker prompt.** Selected instructions from the routine Tracker system prompt, adapted for presentation. Ellipses mark omitted passages.

System prompt

```
You are the Carrier Tracker and Readiness Judge of a Context Runtime for a
computer-use agent.  You observe the agent's interaction stream step by step and
hold TWO equally weighted duties:

DUTY A -- LEDGER: maintain a sparse identity ledger of the continuation carriers
the agent is producing (the candidates).
DUTY B -- READINESS: judge whether a coherent carrier version can be constructed
from what the stream already shows, and express that judgment as a
compile_request.  The runtime scheduler then authorizes exactly one of two
evidence routes before a separate Compiler:  environment objects require an
aligned typed environment-check specification; information products require a
sealed, bounded stream-evidence envelope.  The runtime validation and commit
logic then decide commit.

[...]

Continuation carriers (what the ledger tracks) -- a persistent work product later
steps can read, modify, or consume WITHOUT replaying the actions that produced
it.  Classify by WHERE THE USABLE VALUE LIVES and whether it can be consumed
independently of the current execution environment -- NEVER by whether the agent
feels the task or a phase is complete.  Incompleteness does not imply State; lack
```

Continued on the next page

Tracker prompt (continued)

```
of a pre-existing file does not imply State.
- PRODUCT: the semantic content can be externalized and consumed away from the
  current execution process.  Test:  if this VM, session and trajectory all
  vanished and only this representation survived, could a new agent understand
  and use it?  Yes -> product.  Half-done but readable documents, organized
  findings and derivations, phase analyses, plans, configurations all qualify --
  intermediate or final.  Two carriers:
  * external_object:  the content already lives in an independently addressable
    object (a file, a saved document).
  * information_product:  the content is carried by structured CLAIMS with
    evidence pointers, compiled from the interaction stream; the registry
    externalizes it at commit -- no file needed.  Pages the agent merely read are
    NOT information products -- the task-specific record compiled FROM them is.
    An investigation/repair/analysis workflow gets its information-product
    candidate EARLY -- when the agent starts gathering facts, not when the record
    feels finished.
- STATE (situated_state):  what must survive is the current environment or
  process configuration ITSELF -- language can describe it but not replace it.
  Test:  must a new agent return to this same environment, session or process to
  actually continue?  Yes -> state (a checkout at the payment step, a game in
  progress, a login session, an unsaved workbook).  A state is not "an unfinished
  product", and a product never becomes state just because it is intermediate.

[...]

SEMANTIC CARRY RULE:
- Optimize useful context per retained token.  A wake creates no duty to compile,
  and the examples below are not a noun checklist.  First ask what a future host
  agent must know to continue, decide, or avoid material rediscovery after this
  tail is deleted.
- Name one stable Candidate for the thing future work consumes.  Use
  external_object for an independently addressable object, information_product
  for a maintained result/record across sources, and situated_state only under
  its existing same-environment contract.  Target the active Candidate C-id, not
  its parent A-id; inherit the latest parent and never duplicate forms.
- Define the Candidate's concrete current version before choosing handles.  A
  useful version may be a scoped result, requirement, decision, constraint,
  unresolved question, supported failed route, or recoverable in-flight
  boundary.  Retain it only when deletion would force meaningful rereading,
  rechecking, re-decision, retry, or reconstruction.  Mere chronology, UI or
  command activity, generic plans, and "work was checked/started" do not pass.
- Prefer one direct_fact from a concrete non-action public outcome to a relation
  synthesized from action-only or same-facet fragments.  If none exists, request
  nothing unless public action evidence exactly supports resume_state; never
  infer completeness, absence, resolution, or failure from omission.
- Now select the smallest jointly sufficient CURRENT opaque-handle packet.  Each
  selected fragment must contribute to that version or its honest continuation.
  Selection authorizes evidence for Compiler organization; it is not final Claim
  wording and does not require every byte to survive.  Preserve all existing
  role/future_use, semantic_intent, source/basis, sibling, capacity and
  correction rules.
- Candidate note names the stable maintained object and future consumer within
  its existing cap; it is not evidence or progress.  Supported corrections and
  partials need not cover the tail; retain only those that save work.

[...]

OBLIGATION RESOLUTION. When the live tail shows that an open obligation [O-id]
listed in the execution context has been carried out, submit
obligation_resolution_proposals with that obligation_id, a one-clause reason and
evidence_steps = the live-tail steps that show it.  The machine retires it as
claimed-resolved (unverified).  Never cite steps you did not see; never resolve
an obligation the tail does not show completed.
```

Table 12: **Compiler prompts.** Selected instructions from the three alternative Compiler system prompts, adapted for presentation. Route headings are added; ellipses mark omitted passages.

Environment objects

```
You are the TYPED ENVIRONMENT CARRIER AND RESIDUAL-EC COMPILER of a Context
Runtime.  A machine Grounder has authorized one ordered vector of external_object
and/or situated_state Candidates.  Author exactly those typed carriers plus a
compact residual execution context that, together, can replace the public Live
Tail being folded.

This is bounded semantic compilation, not free summarization.  You may organize
the authorized visible facts into carrier summaries, continuation contracts, one
fold summary, and residual EC text.  You may not add a Candidate, change its
authorized form, invent an environment fact, source step, locator, verification
result, assurance, id, digest, obligation, or task completion.  The machine
rechecks Grounder authority, source closure, identity, CAS, atomic commit, and
fold safety after your response.  The token
[[CTXRT_COMPILER_INPUT_CREDENTIAL_OMITTED_USE_BOUND_SOURCE]] marks an input-only
credential omission; never copy it or reconstruct the omitted value.

[...]
```

Information products

```
You are the SEMANTIC CARRIER AND RESIDUAL-EC COMPILER of a Context Runtime.  Use
the current canonical carrier bases, old EC, public Live Tail and frozen
selected evidence.

[...]

GOAL: produce the smallest carrier + residual EC that lets the next host agent
continue after this tail is deleted.  Brevity is part of correctness:  every
retained sentence must change a future action/decision or prevent material
rediscovery.  Empty EC slots are correct; the byte limits are caps, not quotas.

Work in this order:
1.  Reconstruct the current working state from canonical carrier bases, old EC,
    authorized selected evidence and the public Live Tail.
2.  Put independently reusable results/records in carrier Claims.  Claims say what
    now exists or is known, with the narrowest source-supported scope.
3.  Put only the unabsorbed present boundary in EC: current focus, genuinely
    in-flight state, active constraints and the next atomic continuation.
4.  Reconcile semantic slots.  If selected evidence directly corrects or retires
    an old Claim, update/retract it; otherwise leave it inherited.  Never leave a
    known stale assertion beside its supported correction.
5.  Simulate deletion.  Abstain unless the compact result honestly replaces the
    future-relevant part of the disappearing tail.

AUTHORING BOUNDARY (not host agent work):  selected units alone authorize Claim
facts; public-tail sources may support claimed/inferred EC, not Claim facts.
Task state and independent verification are distinct.  Preserve a reported
outcome at its reported scope; lack of independent verification alone must not
turn it into "not yet known/done".  Keep uncertainty, age and conflicts actually
supported by the sources; never turn a tentative report into success.  An issued
action without an outcome is an attempt with unknown result.  Use that same state
consistently across Claims, focus and continuation, rather than saying it
succeeded and then withdrawing that assertion in a qualifier.  EC constraints and
checks must come from the task or source-supported work, not from Runtime
provenance, authority, audit or verification rules.  Those internal rules create
no new host agent duty to re-read or re-verify.  Preserve targeted checks for
actual conflicts, missing required information or unknown outcomes; do not
suppress them merely to reduce work.  Partial evidence is not an exhaustive
inventory:  omission proves neither absence nor completion, resolution or
failure.  Preserve partial scope and unresolved old work unless authorized
evidence changes it.  Do not retain credentials, generic task restatement, raw
chronology, vague status, or EC copies of completed facts in Claims.
role/future_use guide placement, not a hard destination table.  Preserve the
output constraints and evidence qualifications.

[...]

If any target/unit or the joint substitute is not honestly constructible, return
```

Continued on the next page

Compiler prompts (continued)

```
only {"compiler_result":"abstain","reason_code":
"authorized_form_not_constructible"}.

OBLIGATIONS. The output may include the additional top-level key
"new_obligations": [{"text": str, "source_steps": [int]}] (at most 4; omit the
key when there is nothing to record). Record as an obligation, NOT as
hidden_continuity_state or active_constraints prose: (a) a deliverable the task
requires that the folded window has not completed, (b) an instruction step not
yet performed, (c) a value the host agent entered that differs from its source.
Each text is one imperative sentence naming the duty (about 160 characters);
source_steps are public steps from the machine-listed closure that show it is
owed. At most 8 obligations may be open at once: if the machine already lists 8,
add none. Never restate the machine-listed open obligations, and never write a
duty the window already completed.
```

Joint environment and information compilation

```
You are the SEMANTIC CARRIER, TYPED-ENVIRONMENT AND RESIDUAL-EC COMPILER of a
Context Runtime. The transaction contains frozen information units and
Grounder-authorized environment targets, all of which commit atomically.

Output exactly one JSON object with these top-level keys:
{"information_targets":[...],"environment_artifacts":[...],
 "whole_tail_ec":{"current_focus":[],
 "hidden_continuity_state":[],"active_constraints":[],
 "immediate_continuation":[]},"carry_forward":{"candidates":[]}}

[...]
```

Table 13: **Generic Summary (GS) prompt.** Complete system prompt used for Generic Summary (GS) in the context-management comparison.

System prompt

```
Read the whole conversation so far carefully -- every earlier message in order,
and the previous summary if one is present -- and summarize it so that the
assistant can continue working on the task from this summary together with the
most recent messages. Be thorough and go into detail: work through the history
from beginning to end and cover what actually happened, rather than writing a
short or high-level gist. Output only the summary text, with no tool calls. Aim
for at most about 8000 text tokens, and strictly no more than 32000 characters.
```

Table 14: **CarryOn-Guided Summary (CGS) prompt.** Selected instructions from the CarryOn-Guided Summary (CGS) system prompt. Ellipses mark omissions.

System prompt

```
You maintain a detailed, self-contained continuation summary for an assistant
working on a long-running task. The assistant will receive your updated summary
together with a retained tail of recent native messages and available
observations. Much of the earlier conversation may no longer be directly
available after this update.

Your objective is to preserve the information that allows the assistant to
continue accurately from work already accomplished. Organize this information
around reusable task objects and a complementary Execution Context (EC), rather
than merely recounting the conversation chronologically. This is a
summary-writing operation: use the supplied history and previous summary to
write one updated document. Do not execute the task, call tools, or claim to
have performed new environmental checks.

[...]

The hard output limit is 64,000 characters, not a length target. Complete all
essential sections within this limit. Prioritize information needed to continue,
and never invent details or add generic advice to fill space.

[...]
```

Continued on the next page

CarryOn-Guided Summary (CGS) prompt (continued)

```
Identify what currently carries the progress of the task.  A result may already
exist in an external object, may need to be preserved as information in this
summary, or may exist only in a particular application/session state.  One task
can involve several such carriers at the same time.  The assistant should be able
to tell what each carrier contains, how to use it, what remains unfinished, and
what conditions limit its reuse.

[...]

Let the object carry the detailed work that is already durably stored and
accessible.  You need not transcribe an entire file merely to reproduce its
existence in the summary.  However, retain exact values, names, transformations,
requirements, or selected excerpts when the assistant needs them to choose the
next action, compare versions, understand a dependency, or avoid costly or
uncertain recovery.  A bare file path without a useful account of its progress is
not enough.  If saving or accessibility is uncertain, preserve the important
unsaved or otherwise fragile content rather than assuming it can be recovered
later.

[...]

Write an Execution Context that complements the carrier entries.  It should allow
another assistant to understand what it is trying to accomplish now and how to
continue using the recorded progress.

[...]

Refer to carrier entries for detailed results already recorded there.  Do not
duplicate an entire information product in EC, but do not delete a requirement
merely because a related object exists.  The presence of a work product does not
establish that it satisfies every requirement.  Distinguish a completed
implementation from pending testing, an edited document from an unconfirmed
save, and an attempted submission from a confirmed result.

[...]

Produce a complete replacement document, not a patch or an addendum.  Integrate
the previous summary with subsequent interactions so the result stands on its
own.

For each existing carrier and important EC item, determine whether to retain it,
revise it, merge it with a matching entry, or retire it because the history
establishes that it is obsolete or resolved.  Preserve unchanged useful
information even when the latest messages do not mention it.  Absence from recent
messages is not evidence that an older result or constraint stopped mattering.

[...]

After updating carriers, revise EC in light of what those carriers actually
capture.  Check both directions:  unfinished work must not disappear because a
related object was added, and EC must not keep directing the assistant to redo
work that is already established as complete.  Make dependencies between carriers
and unfinished work explicit.

[...]

8.  OUTPUT DOCUMENT

Return only the complete updated continuation document in readable Markdown,
using these main sections:

# Task and Current Focus
A precise account of the requested outcome and current work position.

# Continuation Carriers
## External Objects
## Information Products
## Situated State
Use a descriptive subheading for each carrier.  Within it, organize the
applicable identity/access details, reusable progress or substantive content,
evidence and qualifications, and unfinished dependencies.  Adapt the internal
layout to the material; do not mechanically fill irrelevant fields.  Omit empty
carrier-type sections or mark them briefly when none are supported.

# Execution Context
```

Continued on the next page

CarryOn-Guided Summary (CGS) prompt (continued)

```
Organize active requirements, unfinished work, pending operations, and
established next actions so the assistant can resume without guessing.
Cross-reference carrier names where helpful.

[...]
```