CONTEXTWEAVE: A REAL-WORLD WORKFLOW BENCHMARK

Anonymous authors
Paper under double-blind review

ABSTRACT

Memory evaluation requires benchmarks that capture the temporal structure and cross-task dependencies of real work. We introduce ContextWeave, a longitudinal benchmark comprising 1,005 executable tasks reconstructed from 14 participants' work records collected over multiple months, including 568 core evaluation tasks. It preserves chronological order, cross-task dependencies, and background tasks while protecting participant privacy. We compare performance on the same target tasks with and without recall under fixed histories and matched initial workspace states. Workspace Score measures task completion and output quality, while Preference Score measures adherence to participant-specific work practices. Across six memory components under a fixed model, the strongest configuration raises Workspace Score from 68.08 to 78.20 and Preference Score from 41.50 to 70.60. With a fixed memory component, recall improves both scores across all five tested base models. Experience-rich configurations outperform compact summaries and reduce repeated exploration, but also exhibit higher rates of memory-induced execution problems. These findings highlight the need to evaluate memory through downstream outcomes, behavioral effects, and robustness in sustained workflows.

1 INTRODUCTION

Language model agents are increasingly being developed to support everyday work rather than solve isolated problems. Such work unfolds over weeks and months: users revisit earlier documents, carry decisions across sessions, and expect established preferences to inform later actions without repeating them in every instruction. Memory systems aim to support this continuity by retaining and reusing past experience (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2024). Evaluating these systems therefore requires more than testing whether an agent can recall information from a long input. It requires observing how effectively an agent uses the history of an ongoing workflow to complete subsequent tasks.

A central challenge is obtaining evaluation data that preserves the temporal structure of real work. In workflows spanning months, relevant information is interspersed with unrelated activity, earlier artifacts become inputs to later tasks, and the relevance of a past interaction depends on what the user needs next. These relationships are difficult to capture through isolated task instances or repeated attempts at the same problem. Long-context question-answering benchmarks assess retrieval and reasoning over extended inputs (Hsieh et al., 2024; Bai et al., 2024), while end-to-end agent benchmarks typically evaluate independent tasks in interactive environments(Mialon et al., 2024; Zhou et al., 2024; Jimenez et al., 2024). Evaluations of self-improvement methods examine gains across repeated trials or epochs (Zhou et al., 2025; 2026), but do not fully capture how distinct tasks build on one another in a user's evolving workflow. Consequently, these evaluations provide limited evidence about the effectiveness of memory systems in sustained, real-world work.

We introduce ContextWeave, a longitudinal benchmark grounded in revision histories of real users' work reports collected over multiple months. These records capture how work develops over time, providing a basis for reconstructing tasks and identifying the historical context relevant to subsequent work. Through privacy protection, task reconstruction, and human annotation, we organize them into executable task streams that retain chronological order, cross-task dependencies, and interleaved background tasks. Each evaluation instance links a target task to preceding tasks within the same user's workflow. We compare agent performance on the same target task with and without recall,

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.

using fixed historical trajectories and matched initial workspace states to isolate the contribution of recalled experience. By combining months-long records of real work with controlled downstream evaluation, ContextWeave enables the study of memory in sustained workflows.

Our experiments show that memory helps agents preserve continuity across ongoing workflows. We measure task completion and output quality through Workspace Score and adherence to participant-specific work practices through Preference Score. Under a fixed base model, the strongest memory configuration raises Workspace Score from 68.08 to 78.20 and Preference Score from 41.50 to 70.60. In a separate comparison using a fixed memory component, recall improves both scores across all five tested base models. Without memory, agents often produce seemingly complete outputs that overlook previously established workspace state and conventions. Comparisons across memory components further show that, in our tested configurations, experience-rich memory supports stronger downstream performance and reduces repeated exploration more effectively than compact summaries. However, these benefits are accompanied by higher rates of memory-induced execution problems, highlighting the importance of reliable memory use alongside the actionability of recalled experience. Together, these findings motivate evaluating memory through downstream task outcomes, behavioral effects, and robustness to misleading recall.

Our contributions are as follows:

- **A longitudinal benchmark grounded in months of real work.** We introduce ContextWeave, comprising 1,005 executable tasks reconstructed from the multi-month work records of 14 participants, with 568 primary-workflow tasks serving as evaluation targets. The participant-specific streams retain chronological order, cross-task dependencies, and interleaved background tasks, providing realistic historical context for evaluating memory in sustained workflows.
- **A controlled protocol for evaluating memory through downstream outcomes.** We compare performance on the same target tasks with and without recall, using fixed historical trajectories and matched initial workspace states to control for accumulated execution drift. The protocol evaluates task completion and output quality through Workspace Score and adherence to participant-specific work practices through Preference Score, complemented by diagnostics of relevance, repeated exploration, problem solvability, and memory-induced execution problems.
- **An empirical study of the benefits and risks of agent memory.** We compare six memory components under a fixed base model and evaluate five base models using a fixed memory component. Recall improves both downstream scores across all tested models. Across memory components, experience-rich configurations achieve stronger task performance and reduce repeated exploration more effectively than compact summaries, but also exhibit higher rates of memory-induced execution problems.

2 Related Work

Memory has been studied in different contexts across language-model and agent research, each with a distinct evaluation target. In long-context modeling, memory emphasizes information access over extended inputs. And it often refers to working memory within an episode or episodic memory across sessions in agent systems. We organize existing memory evaluations and provide a brief overview here. See Appendix A for the full discussion.

2.1 Memory Evaluation

Downstream Evaluation. Long-context QA benchmarks evaluate whether models can answer questions grounded in long inputs or multi-session histories (Hsieh et al., 2024; Bai et al., 2024; Maharana et al., 2024; Wu et al., 2025). End-to-end agent benchmarks assess task completion in interactive environments, but typically organize tasks as independent instances (Mialon et al., 2024; Jimenez et al., 2024). Self-improvement methods further study experience reuse across attempts or tasks (Shinn et al., 2023; Zhou et al., 2025; 2026). MemoryArena evaluates cross-session memory through human-crafted, interdependent subtasks (He et al., 2026).

Memory Component Evaluation. MemBench, MemoryAgentBench, and MemoryBench evaluate targeted memory capabilities, including factual and reflective memory, retrieval, learning, and forgetting (Tan et al., 2025; Hu et al., 2026; Ai et al., 2025). These evaluations help diagnose how memory systems store, organize, and update information. Such diagnostics complement our evaluation of memory through its contribution to subsequent task performance.

Our benchmark primarily evaluates memory through its contribution to downstream task performance. We complement existing evaluations with task streams reconstructed from real work records. These streams allow us to recover complex dependencies between tasks from actual workflows, rather than design them explicitly for evaluation. We also introduce diagnostic metrics to help interpret the effects of memory.

2.2 Memory Systems

Memory systems differ in how they extract, organize, and update past experience. MemoryBank and Mem0 support long-term memory maintenance and consolidation (Zhong et al., 2024; Chhikara et al., 2025). LangMem provides modular utilities for memory construction and updating (LangChain, 2025). A-MEM organizes experience into linked notes (Xu et al., 2025), while MemOS manages heterogeneous memory representations (Li et al., 2025). These memory components present past experience as summaries or few-shot examples. Our benchmark evaluates these components within a fixed agent harness to compare their effects on downstream task performance.

3 Design Principle

3.1 Mathematical Formulation

We model a user's workflow as a chronological sequence of related tasks:

$$D = (T_1, T_2, \ldots, T_n). \tag{1}$$

For each target task T_i, let $R(T_i)$ denote agent performance without access to prior task histories. Let $R_M(T_i)$ denote performance with memory produced by component M from the trajectories of preceding tasks $(T_1, \ldots, T_{i-1})$. We measure the contribution of memory as

$$\Delta R_M(T_i) = R_M(T_i) - R(T_i). \tag{2}$$

Positive values indicate that memory improves performance on the target task.

3.2 Benchmark Design Considerations

To ensure that the benchmark faithfully reflects memory usage in realistic agent scenarios, we consider four design principles.

Privacy-preserving realism. Realistic memory evaluation requires authentic workflow histories rather than artificially constructed interactions. We therefore build our benchmark from real-world collaborative workflows and apply privacy-preserving processing to remove sensitive information while maintaining the structural properties of the original tasks, including temporal order, entity consistency, and task dependencies.

Executable and reproducible environments. Memory evaluation should go beyond passive information retrieval and measure whether agents can leverage historical experience during task execution. We therefore provide executable environments that allow agents to directly interact with task contexts. At the same time, we ensure reproducibility by controlling external dependencies and reconstructing unavailable resources when necessary, enabling consistent evaluation across different memory systems.

Controlled longitudinal evaluation. Although real workflows naturally evolve through sequential task execution, directly replaying the entire process introduces efficiency challenges and error accumulation. We therefore decouple historical experience from environment evolution by preserving controlled task trajectories as memory inputs, allowing memory components to process realistic past experiences while maintaining a stable evaluation setting.

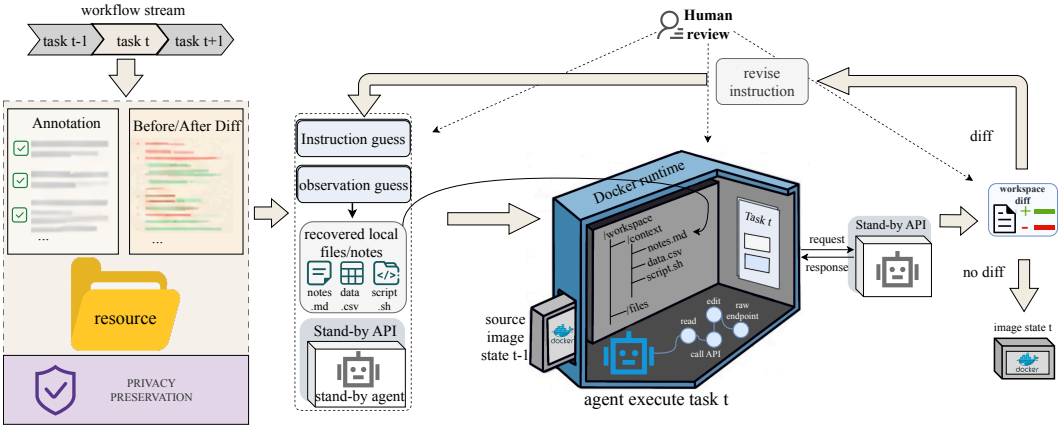

Figure 1: Benchmark construction from real work records. We extract and anonymize task streams, reconstruct executable tasks, and align historical trajectories to recorded workspace states for controlled memory evaluation.

Downstream-oriented evaluation. The ultimate goal of memory is to improve an agent's ability to complete future tasks, rather than merely optimize intermediate memory operations. Therefore, we prioritize downstream task outcomes, including task execution quality and alignment with user preferences, while treating memory-specific measurements as complementary diagnostics for understanding agent behavior.

4 Data Construction

We reconstruct executable task streams from longitudinal records of real work. The benchmark preserves dependencies within users' workflows while controlling the histories and environments used for memory evaluation. Figure 1 summarizes the construction pipeline.

4.1 Anonymous Task Streams from Real Workflows

Task Segmentation. We collect revision histories of daily work reports from 14 participants in an open-source project over multiple months. Each edit is recorded with its document identifier, before-and-after content, and corresponding diff. Successive revisions of these reports serve as chronological worklogs. We use an LLM to group consecutive edits into tasks representing user-level objectives. Human annotators verify task boundaries and annotate primary-workflow membership, required resources, and reconstruction constraints. This process yields 1,005 tasks, including 568 primary-workflow tasks used for evaluation.

Task Anonymization. Real-world records contain interconnected sensitive information, including different names referring to the same entity. We first use an LLM to identify sensitive entities and their relationships. We then apply consistent replacements across each task stream. This preserves entity relationships, temporal order, and task dependencies while removing private information.

4.2 Executable Task Reconstruction

Worklogs record the outcomes of human work but do not directly provide the instructions and context needed for agent execution. We use an LLM to infer task instructions from document changes and refine them through human review. The instructions resemble everyday user requests and specify goals, resources, and constraints. They do not prescribe an execution procedure, which reduces the risk of leaking the final document content.

Worklogs provide only partial observations of a task. To make tasks executable, we reconstruct missing dependencies through two mechanisms. Recoverable static resources are materialized as local files or structured notes. Resources that cannot be safely exposed or practically restored are accessed through controlled APIs. All API inputs, including code snippets, are treated as text. Requests are

routed to a simulator agent, which generates responses based on the query and resource information recovered from workflow records and annotations.

We validate reconstructed tasks through execution in isolated Docker environments. A verifier checks whether the execution corresponds to the original task, without requiring exact textual agreement. Mismatched instructions are revised, while execution failures receive human review for missing observations or environment issues. We retain the validated instructions, observations, trajectories, and container states. Together, LLM-assisted reconstruction and human annotation transform each message-log segment into an executable agent task.

4.3 Controlled Longitudinal Evaluation

Different valid executions of the same task can produce different workspace states. Across a task stream, these differences can accumulate and change the conditions of subsequent tasks. We control this drift by aligning validated historical trajectories to the outcomes recorded in the original worklogs.

Specifically, we rewrite each validated trajectory while preserving its high-level sequence of observations, reasoning, tool calls, and file operations. Concrete outputs are adjusted so that applying the trajectory reproduces the recorded workspace changes. The aligned trajectory establishes a canonical post-task state for the next task. This alignment prepares historical inputs and starting environments. Each target task is then executed and evaluated from its prepared starting state. This supports independent or parallel evaluation of memory use under fixed histories and matched task conditions.

4.4 Evaluation Criteria

We evaluate memory through two downstream outcomes and four complementary diagnostics.

Workspace Score is designed to reflect how completed work is assessed in real-world settings. We assess task completion from the final Docker environment rather than the agent's response. We annotate task requirements using the validated reference workspace and express them as evaluation rubrics. Scores range from 0 to 100, with approximately 60 indicating minimum completion and 80 indicating reference-level quality. Higher scores require clear improvements over the reference.

Preference Score aims to capture implicit preferences in office workflows. It measures adherence to participant-specific work practices. We supplement human-defined preferences with cross-task rubrics derived by an LLM from historical messages and document changes. An LLM judge evaluates document writes against the applicable rubrics. Weighted rubric scores are normalized to 0–100. We also report anonymized pairwise win rates between with-recall and without-recall trajectories.

To help interpret these outcome scores, we report the following diagnostic metrics. **Relevance** compares recalled information with historical evidence identified by an LLM from preceding tasks. **Continuity** measures repeated exploration through tool-call counts and proportions. **Solvability** considers problems left unresolved without recall. It reports the proportion that an LLM judges could be resolved or avoided using the recalled information. **Hallucination Robustness** reports the proportion of tasks containing execution problems that an LLM attributes to misleading recall. We additionally report 1–5 LLM ratings of relevance, environment familiarity, and hallucination robustness. Full construction and scoring procedures are provided in Appendix B.

5 Empirical Findings

5.1 Data Statistics

Benchmark scale. Following the data construction process described above, we constructed 1005 executable agent tasks from the long-term real-world workflows of 14 participants. Of these, 568 tasks were manually labeled as main tasks under the participants' primary project goals and served as core evaluation tasks. The remaining 437 tasks were background or temporary non-main tasks. Non-main tasks were not directly used as evaluation targets but were retained in the same participant's historical sequence to maintain noise and contextual dependencies in the real workflow. We used a GPT-5 tokenizer to calculate the size of the message log for each task. All 1005 message logs contained an average of 36.3K tokens, with the longest reaching 212.3K tokens.

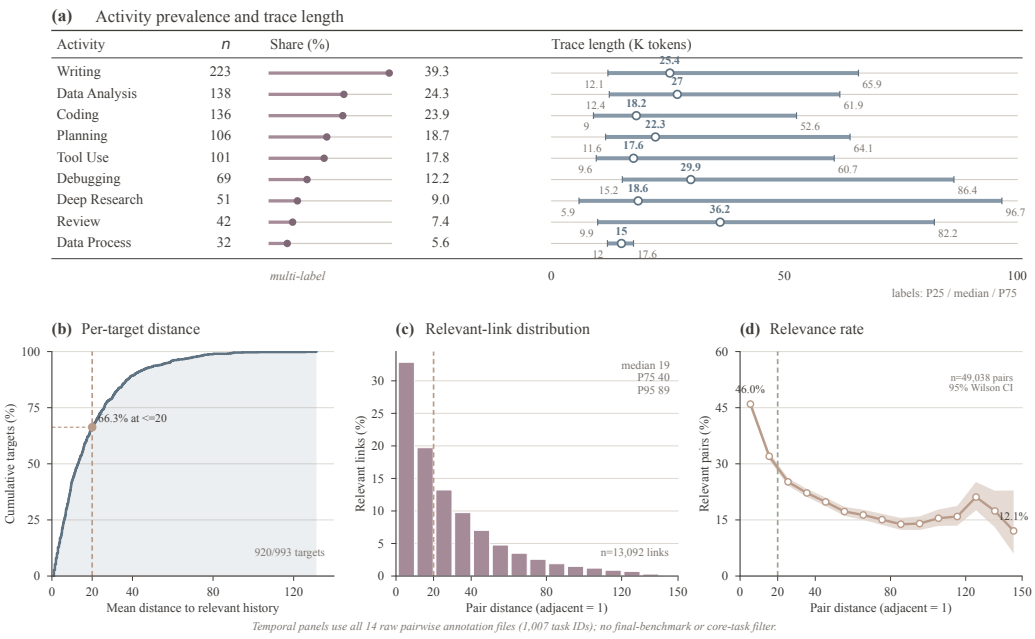

Figure 2: Task diversity and temporal relevance. (a) Multi-label activity prevalence and trace-length distributions. (b–d) Aggregated temporal-distance and relevance statistics.

Task diversity. To characterize the diversity of task content, we manually labeled the agentic activities of 568 core tasks using multi-label annotation. As shown in Figure 2(a), these core tasks cover various real-world agent activities, including writing, data analysis, and coding. The trace size also varies depending on the task type; for example, debugging, review, writing, and deep research tasks typically contain longer contexts.

Temporal relevance. For each core evaluation task, we used an LLM to identify relevant preceding tasks and extracted supporting evidence spans from their message logs to form gold historical evidence. Based on these annotations, we calculated the temporal distance between relevant preceding tasks and the current task. Of the 568 core tasks, 541 depended on at least one preceding task, resulting in 8084 relevant links. Among these, 75.6% of the tasks had an average distance of no more than 20 subtasks between their relevant preceding tasks, demonstrating significant locality; however, 24.4% still depended on earlier history with an average distance of more than 20 subtasks, indicating that the benchmark simultaneously includes short-term workflow continuity and long-term episodic dependency. Complete statistics can be found in Appendix C.

5.2 How Different Memory Components Affect Agents

To evaluate different memory components, we fix both the agent harness and the underlying model. We assess not only whether the agent produces an apparently complete response, but also whether the resulting artifacts are usable in the actual workspace. We also consider whether the agent preserves work practices established by the participant across earlier tasks.

5.2.1 Evaluation Setting

Execution Setting. We use Codex(OpenAI, 2026) as the agent harness and GPT-5.5 as the execution model. For each primary-workflow task, the agent starts from the canonical pre-task state produced by the trajectory-alignment procedure described in Section B.4 and executes independently in an isolated Docker environment. All conditions use the same task instruction, initial workspace, model, and tool permissions; they differ only in the recalled context provided to the agent.

Metric Setting. We evaluate downstream outcomes using the Workspace Score and Preference Score introduced in Section B.5. We use GPT-5.5 for all grading calls and keep the grader model

Table 1: Unified downstream and diagnostic results. We compare a `no-recall` baseline, in which no historical context is injected, with `with-recall` conditions augmented by different memory components. Column groups indicate the primary context representation used by each component. Diagnostic metrics are not defined for No Recall. Count and rate changes are computed as with-recall minus the shared without-recall trajectory; count changes are rounded to two decimal places and rate changes are reported in percentage points (pp).

	Baseline	*Summary-Based Memory*				*Task Summary*	*ICL Experience*
Metric	**No Recall**	**mem0**	**memos**	**supermemory**	**memorybank**	**langmem**	**a-mem**
Downstream Performance							
Workspace Score ↑	68.08	72.48	70.01	70.57	73.24	75.29	**78.20**
Preference Score ↑	41.50	49.73	46.38	48.04	55.40	57.37	**70.60**
With-recall Win Rate ↑	–	50.70%	53.30%	55.60%	65.08%	62.79%	**72.70%**
Relevance							
LLM Annotated ↑	–	2.97	**4.09**	3.28	3.64	3.98	3.73
Mean Similarity ↑	–	0.48	0.57	0.51	0.56	0.62	**0.72**
Recall ↑	–	0.08	0.23	0.11	0.21	0.37	**0.69**
Precision ↑	–	0.22	**0.32**	0.18	0.27	0.22	0.30
Continuity							
LLM Annotated ↑	–	3.13	3.20	3.15	3.44	3.93	**4.00**
Exploration Δ ↓	–	-0.79	-1.20	-0.77	-1.19	-2.57	**-5.37**
Execution Δ	–	+0.74	+1.41	-0.17	+2.19	+1.66	-0.09
Exploration Rate Δ ↓	–	-2.22 pp	-4.27 pp	-0.69 pp	-5.59 pp	-6.33 pp	**-7.06 pp**
Execution Rate Δ ↑	–	+2.38 pp	+3.93 pp	+0.40 pp	+5.37 pp	+6.22 pp	**+6.63 pp**
Hallucination Robustness							
LLM Annotated ↑	–	**4.90**	4.85	4.87	4.83	4.76	4.66
Memory-induced Task Rate ↓	–	**0.35%**	**0.35%**	0.70%	1.23%	5.11%	7.39%
Solvability							
Problem Solvability Rate ↑	–	1.45%	1.45%	0.96%	1.45%	6.59%	**7.07%**

and prompts fixed across memory conditions. Workspace grading is performed through Codex: the grader receives a task-specific evaluation bundle and read-only access to the candidate and reference Docker images, and is instructed to ground its score in inspected artifacts and executable checks. For Preference Score, each applicable participant-specific rubric is evaluated in an independent GPT-5.5 call. For the complementary pairwise evaluation, the with-recall and without-recall trajectories are randomly anonymized as candidates A and B, and the judge is required to select one candidate without ties. Detailed grading prompts are provided in Appendix D. Table 1 reports the main results.

Diagnostic Setting. We use the Memory Diagnostics introduced in Section B.5 and keep the reference evidence, grader model, and prompts fixed across memory components. All LLM-based annotations and judgments use GPT-5.5. Reference historical evidence is constructed by comparing each evaluation task with all earlier tasks in the participant's workflow. Relevance, Continuity, and Hallucination Robustness are judged from the complete paired with-recall and without-recall trajectories. Tool-call classification, problem solvability, and memory-induced problems are evaluated through separate calls with fixed prompts. For the embedding-based relevance metrics, we encode recalled chunks and reference excerpts using `text-embedding-3-small` and apply a cosine-similarity threshold of 0.65. See Appendix D for detailed prompts. Table 1 reports the main results.

5.2.2 Memory Turns Apparent Completion into Usable Work

Table 1 shows that memory improves both the final workspace and alignment with participant-specific work practices. In particular, the Workspace Score increases from approximately 68 without memory to approximately 78 with the strongest memory component, moving performance from basic but unreliable completion toward reference-level usability. Preference follows the same overall trend.

Taken together, the Workspace and Preference results suggest that no-recall agents often achieve apparent completion without producing a usable, workflow-consistent outcome. To understand this pattern, we manually inspect the paired execution trajectories and their final workspaces. The resulting artifacts often appear plausible in isolation but fail to preserve the state and conventions established by prior work. Memory restores these latent task states and behavioral constraints, allowing the agent to continue an existing workflow rather than reconstruct a generic solution from the current instruction alone. Its primary value is therefore not to provide additional background knowledge, but to turn apparent completion into an output that is consistent with the work that came before it.

5.2.3 IN-CONTEXT EXPERIENCE IS MORE EFFECTIVE THAN SUMMARIZATION

Table 1 highlights the distinction between summary-oriented memory and in-context experience. Summary-based systems perform well on semantic relevance and precision but remain weaker in evidence coverage, solvability, and downstream performance. In contrast, `a-mem`, a representative of ICL-style memory, provides broader historical coverage, stronger continuity, and a larger reduction in exploration. The structured task-level summaries produced by `langmem` fall between these two regimes. Overall, the closer recalled history is to a concrete working example, the less context the agent must reconstruct before acting.

The Solvability results provide a direct explanation for this pattern: ICL-style memory and structured task summaries cover a substantially larger share of execution problems for which historical evidence is useful than the lighter-weight summary baselines. Summarization primarily preserves task conclusions and state, whereas in-context experience additionally retains the paths, constraints, existing artifacts, and procedures needed to act on that history. This actionability comes with a trade-off, as richer recall is also associated with more memory-induced problems. The results therefore favor curated in-context experience over pure summarization, rather than indiscriminately retaining more history. Because the components also differ in retrieval strategy and context length, we interpret this comparison as an empirical association and leave a controlled study under a fixed token budget to future work.

5.3 DIFFERENT BASE MODELS USE MEMORY DIFFERENTLY

Having compared memory components under a fixed execution model, we next examine how memory interacts with different base models. We ask whether models benefit equally from recall, how recall changes their execution behavior, and how reliably they use the information it provides.

5.3.1 EVALUATION SETTING

We follow the execution setting in Section 5.2, using Codex as the agent harness and the same task instructions, canonical pre-task workspaces, and tool permissions. We fix the memory component to `mem0` and vary only the execution model, evaluating DeepSeek-V4-Pro, GPT-5.5, GLM-5.1, Kimi-K2.6, and Qwen3.7-Max under both no-recall and with-recall conditions on all 568 core tasks.

We use the Workspace, Preference, Continuity, and Hallucination Robustness metrics described in Sections 5.2 and B.5. All grading, annotation, and judgment calls use GPT-5.5 with fixed prompts. We additionally examine which unresolved problems could have been solved using recall. GPT-5.5 identifies unresolved problems in each with-recall trajectory, and a separate GPT-5.5 call checks whether the recall contains enough information to solve or avoid each problem. Recall-Solvable Problems is the percentage of unresolved problems that could have been solved using the recall; lower is better. Table 2 reports the results.

5.3.2 MEMORY IMPROVES EVERY MODEL, BUT BY DIFFERENT AMOUNTS

Table 2 shows that `mem0` improves both Workspace and Preference scores for every model. However, the model ranking is already largely established without recall and remains similar after memory is added. GLM achieves the highest Workspace score in both conditions, whereas DeepSeek obtains the largest Workspace improvement. The same pattern appears in Preference: every model benefits from recall, but the magnitude of the gain varies substantially. Memory therefore provides useful historical state without removing differences in the models' underlying ability to complete tasks.

5.3.3 MODELS WITH LARGER GAINS EXPLORE LESS AND EXECUTE MORE

The behavioral results distinguish reducing tool use from using memory productively. Kimi and GLM show the largest reductions in the absolute number of exploration calls, but their execution calls also decrease and their Workspace gains are relatively small. DeepSeek instead shifts its tool-call distribution: its Exploration Rate decreases by 3.40 percentage points while its Execution Rate increases by 3.26 points, accompanied by the largest Workspace gain of 5.61. GPT shows the same direction with a smaller behavioral shift and the second-largest Workspace gain. Memory is most

Table 2: Downstream performance and memory-use diagnostics across base models with `mem0`. All changes are computed as with-recall minus no-recall; count changes are rounded to two decimal places and rate changes are reported in percentage points (pp).

Metric	DeepSeek-V4-Pro	GPT-5.5	GLM-5.1	Kimi-K2.6	Qwen3.7-Max
Downstream Performance					
Workspace Score, No Recall ↑	61.76	67.53	**71.91**	66.18	62.93
Workspace Score, With Recall ↑	67.36	72.48	**74.10**	69.16	65.99
Workspace Score Δ ↑	**+5.61**	+4.95	+2.19	+2.99	+3.06
Preference Score, No Recall ↑	32.48	42.07	**46.09**	40.20	35.83
Preference Score, With Recall ↑	42.09	49.73	**51.91**	48.57	41.39
Preference Score Δ ↑	**+9.61**	+7.66	+5.83	+8.37	+5.55
With-recall Win Rate ↑	**61.09%**	50.70%	55.46%	57.39%	51.94%
Continuity					
LLM Annotated ↑	3.19	3.13	**3.33**	3.29	3.22
Exploration Δ ↓	+0.26	-0.32	-2.49	**-3.01**	+0.37
Execution Δ ↑	**+2.28**	+0.27	-1.07	-0.76	+0.56
Exploration Rate Δ ↓	**-3.40 pp**	-0.79 pp	-0.68 pp	-1.20 pp	-1.06 pp
Execution Rate Δ ↑	**+3.26 pp**	+0.94 pp	-0.05 pp	+0.49 pp	+1.12 pp
Hallucination Robustness					
LLM Annotated ↑	4.32	**4.90**	4.35	4.41	4.31
Memory-induced Task Rate ↓	0.53%	**0.00%**	2.11%	0.88%	1.23%
Memory Utilization					
Recall-Solvable Problems ↓	1.89%	**1.52%**	2.27%	3.03%	2.23%

useful when it reallocates effort from reconstructing workspace state to modifying, running, and validating task outputs, rather than merely shortening the trajectory.

5.3.4 Models with Larger Gains Use Recall Better

GPT and DeepSeek have the lowest rates of Recall-Solvable Problems, at 1.52% and 1.89%, and also obtain the two largest Workspace gains. Qwen and GLM fall in the middle, at 2.23% and 2.27%, while Kimi has the highest rate at 3.03%. This pattern suggests that one source of variation in memory gains is whether a model turns recalled information into effective action: recall may already contain a useful path or constraint, yet the corresponding problem can remain unresolved. Because the memory component is fixed to `mem0`, this metric focuses on differences in how base models act on the provided recall rather than differences between memory systems.

5.3.5 Some Models Are More Easily Misled by Memory

The model also affects the risk introduced by memory. Although every condition uses `mem0`, Memory-induced Task Rate ranges from 0.00% for GPT to 2.11% for GLM. These cases include all execution problems caused or worsened by misleading recall, regardless of whether the agent later resolves them. Memory robustness is therefore determined jointly by what the memory system retrieves and how the base model interprets, verifies, and applies that information.

6 Conclusion

We introduced ContextWeave, a longitudinal benchmark that transforms privacy-preserved real-world workflows into 1,005 executable tasks with controlled environments and downstream-oriented evaluation, including 568 core tasks. Experiments across six memory components and five base models show that recall improves workspace quality and preference alignment in the tested settings, but that its value depends on whether the retrieved experience is actionable and whether the base model can apply it reliably. By jointly measuring task outcomes and the relevance, continuity, solvability, and robustness of recalled history, ContextWeave provides a reproducible basis for developing memory systems that support sustained workflow execution.

AI USE STATEMENT

In this work, we used generative AI tools for synthetic data generation, translation assistance, and dataset cleaning and reformatting. Additionally, we used these tools to create or modify software code, transcribe recordings of research material, and edit the manuscript to improve readability. We did not use generative AI tools for any other tasks listed in the required or recommended disclosure categories. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the assistance of generative AI.

REFERENCES

Qingyao Ai, Yichen Tang, Changyue Wang, Jianming Long, Weihang Su, and Yiqun Liu. MemoryBench: A benchmark for memory and continual learning in LLM systems. *arXiv preprint arXiv:2510.17281*, 2025. URL https://arxiv.org/abs/2510.17281.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3119–3137, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.172. URL https://aclanthology.org/2024.acl-long.172/.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory. In *ECAI 2025 - 28th European Conference on Artificial Intelligence*, volume 413 of *Frontiers in Artificial Intelligence and Applications*, pp. 2993–3000. IOS Press, 2025. doi: 10.3233/FAIA251160. URL https://doi.org/10.3233/FAIA251160.

Yunhao Fang, Weihao Yu, Shu Zhong, Qinghao Ye, Xuehan Xiong, and Lai Wei. Artificial Hippocampus Networks for Efficient Long-Context Modeling. *CoRR*, abs/2510.07318, 2025. doi: 10.48550/arXiv.2510.07318. URL https://doi.org/10.48550/arXiv.2510.07318.

Zexue He, Yu Wang, Churan Zhi, Yuanzhe Hu, Tzu-Ping Chen, Lang Yin, Ze Chen, Tong Arthur Wu, Siru Ouyang, Zihan Wang, Jiaxin Pei, Julian McAuley, Yejin Choi, and Alex Pentland. Memoryarena: Benchmarking agent memory in interdependent multi-session agentic tasks. *arXiv preprint arXiv:2602.16313*, 2026. URL https://arxiv.org/abs/2602.16313.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? *CoRR*, abs/2404.06654, 2024. doi: 10.48550/arXiv.2404.06654. URL https://doi.org/10.48550/arXiv.2404.06654.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in LLM agents via incremental multi-turn interactions. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=DT7JyQC3MR.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. *CoRR*, abs/2607.07946, 2026. doi: 10.48550/arXiv.2607.07946. URL https://doi.org/10.48550/arXiv.2607.07946.

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 *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

LangChain. LangMem: Modular memory for agentic systems, 2025. URL https://github.com/langchain-ai/langmem. Software repository.

Zhiyu Li, Shichao Song, Chenyang Xi, Hanyu Wang, Chen Tang, Simin Niu, Ding Chen, Jiawei Yang, Chunyu Li, Qingchen Yu, Jihao Zhao, Yezhaohui Wang, Peng Liu, Zehao Lin, Pengyuan Wang, Jiahao Huo, Tianyi Chen, Kai Chen, Kehang Li, Zhen Tao, Junpeng Ren, Huayi Lai, Hao Wu, Bo Tang, Zhenren Wang, Zhaoxin Fan, Ningyu Zhang, Linfeng Zhang, Junchi Yan,

Mingchuan Yang, Tong Xu, Wei Xu, Huajun Chen, Haofen Wang, Hongkang Yang, Wentao Zhang, Zhi-Qin John Xu, Siheng Chen, and Feiyu Xiong. MemOS: A memory OS for AI system. *CoRR*, abs/2507.03724, 2025. doi: 10.48550/ARXIV.2507.03724. URL https://doi.org/10.48550/arXiv.2507.03724.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.747. URL https://aclanthology.org/2024.acl-long.747/.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: A benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=fibxvahvs3.

OpenAI. Codex CLI: A coding agent for the terminal. Computer software, May 2026. URL https://github.com/openai/codex/releases/tag/rust-v0.134.0. Version 0.134.0, Git tag rust-v0.134.0, commit a75c443fdb64db48c3cf4bdb247c7ee52c0144c9.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=0IOX0YcCdTn.

Haoran Tan, Zeyu Zhang, Chen Ma, Xu Chen, Quanyu Dai, and Zhenhua Dong. MemBench: Towards more comprehensive evaluation on the memory of LLM-based agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 19336–19352, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.989. URL https://aclanthology.org/2025.findings-acl.989/.

Bo Wang, Tianxiang Sun, Hang Yan, Siyin Wang, Qingyuan Cheng, and Xipeng Qiu. In-memory learning: A declarative learning framework for large language models. *CoRR*, abs/2403.02757, 2024. doi: 10.48550/ARXIV.2403.02757. URL https://doi.org/10.48550/arXiv.2403.02757.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *The Thirteenth International Conference on Learning Representations, ICLR 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=pZiyCaVuti.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/19909c36f51abc4856b4560aff3d36d6-Abstract-Conference.html.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, October–November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024. doi: 10.1609/AAAI.V38I17.29936. URL https://doi.org/10.1609/aaai.v38i17.29936.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024*, pp. 19724–19731. AAAI Press, 2024. doi: 10.1609/AAAI.V38I17.29946. URL https://doi.org/10.1609/aaai.v38i17.29946.

Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, and Jun Wang. Memento: Fine-tuning LLM agents without fine-tuning LLMs. *CoRR*, abs/2508.16153, 2025. doi: 10.48550/arXiv.2508.16153. URL https://doi.org/10.48550/arXiv.2508.16153.

Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, Runyu Yang, Qiangbin Liu, Xinlei Yu, Jianmin Zhou, Na Wang, Chunyang Sun, and Jun Wang. Memento-Skills: Let agents design agents. *CoRR*, abs/2603.18743, 2026. doi: 10.48550/arXiv.2603.18743. URL https://doi.org/10.48550/arXiv.2603.18743.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations, ICLR 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=oKn9c6ytLx.

A Related Work

A.1 Long History QA

A common line of work evaluates memory through a query-over-history format: a model is given a long context or conversation history and asked to answer a query grounded in that record. RULER extends Needle-in-a-Haystack-style evaluation with retrieval, multi-hop tracing, and aggregation tasks over synthetic long contexts (Hsieh et al., 2024). This form of evaluation is widely used in long-context modeling (Fang et al., 2025; Bai et al., 2024). Long-term conversational benchmarks such as LoCoMo and LongMemEval instantiate a similar format over multi-session dialogue histories, with questions targeting facts, events, preferences, temporal relations, or updated user information (Maharana et al., 2024; Wu et al., 2025). These benchmarks capture an important aspect of memory: whether key information can be extracted from historical context and used when queried. However, because they reduce evaluation to answering a query over a given record, they leave open whether memory improves downstream performance on more complex agentic tasks.

A.2 Independent Agent Task

Another line of work studies memory through end-to-end agent tasks. Benchmarks such as GAIA, SWE-bench, and DeepSWE require agents to solve complex problems over extended trajectories, thereby implicitly testing their ability to maintain, organize, and use task state during execution (Mialon et al., 2024; Jimenez et al., 2024; Huang et al., 2026). These benchmarks provide useful probes of working memory in realistic agent tasks, even though they are not designed specifically for memory. To approximate episodic memory, prior work often runs agents for multiple trials or epochs on tasks such as HotpotQA and ALFWorld, using changes in end-to-end performance as evidence of experience reuse (Yang et al., 2018; Shridhar et al., 2021; Shinn et al., 2023; Zhou et al., 2025; 2026). This extends memory evaluation beyond QA-style interfaces by measuring how accumulated experience affects agent behavior in task execution. However, real longitudinal use involves dependencies that may be sparse, implicit, and user-specific, making it unclear whether repeated-trial evaluations capture memory effectiveness in realistic workflows.

A.3 Constructed multi-session agent tasks

Recent benchmarks have begun to evaluate memory in multi-session agentic settings. MemoryArena constructs human-crafted tasks with explicitly interdependent subtasks, where agents must distill experience from earlier sessions and use it to guide later actions in a Memory-Agent-Environment loop (He et al., 2026). This direction moves beyond isolated recall and evaluates memory through downstream agent behavior. However, the cross-session dependencies in these benchmarks are manually constructed, which may not fully reflect the sparse and implicit dependencies that emerge in real-world workflows.

A.4 Component-level memory diagnostics

Some benchmarks evaluate memory by first specifying the functional capabilities that a memory system is expected to support, and then designing targeted metrics for each capability. For example, MemBench measures LLM-agent memory across factual and reflective memory, participation and observation scenarios, and metrics such as effectiveness, efficiency, and capacity (Tan et al., 2025). MemoryAgentBench decomposes memory agents into competencies including accurate retrieval, test-time learning, long-range understanding, and selective forgetting (Hu et al., 2026). MemoryBench simulates user feedback to study memory and continual learning in LLM systems, covering declarative and procedural memory needs (Ai et al., 2025). These benchmarks are useful for diagnosing whether a system can store, retrieve, update, abstract, or forget information properly. Such component-level measurements are better viewed as explanatory signals than as direct estimates of downstream agent performance: a memory system that performs well on retrieval or update metrics may not necessarily help an agent select and use past experience effectively in future workflows.

A.5 Memory Components in LLM Agents

Beyond memory evaluation, we briefly review how memory components are constructed in LLM-based agents. Existing systems differ in how they abstract, organize, and update past experience. MemoryBank maintains long-term user memories through continuous updates and an Ebbinghaus-inspired forgetting and reinforcement mechanism, enabling agents to adapt to users over time (Zhong et al., 2024). Mem0 dynamically extracts, consolidates, and retrieves salient information from conversations, with a graph-based variant for capturing relational structure (Chhikara et al., 2025). LangMem is a software framework that provides modular utilities for constructing semantic, episodic, and procedural memories, supporting both in-conversation and background memory updates (LangChain, 2025). A-MEM organizes experience as Zettelkasten-inspired structured notes and dynamically creates and updates links among memory items (Xu et al., 2025). MemOS treats memory as a manageable system resource and unifies the representation, scheduling, and evolution of plaintext, activation-based, and parameter-level memories (Li et al., 2025). Together, these systems span a design space of memory abstraction, organization, and lifecycle control: compact fact- or summary-based stores favor efficient retrieval, linked-note systems maintain explicit relationships among memories, and OS-style frameworks coordinate heterogeneous memory representations.

B Consturction Detail

We construct our benchmark from longitudinal records of real-world workflows rather than manually designed task sequences. Specifically, we collect complete work histories from an open-source project involving 14 participants over multiple months.

Starting from these raw workflow records, we construct executable agent tasks through a series of processing steps, including task stream extraction, privacy-preserving data processing, task reconstruction, controlled trajectory alignment, and evaluation design. The resulting benchmark preserves the temporal structure and dependencies of real workflows while transforming human activity traces into reproducible agent task sequences with executable environments and downstream-oriented evaluation criteria.

B.1 From Real Workflows to Task Streams

The first step in evaluating episodic memory is to recover realistic task sequences from raw workflow traces. For each participant, we collect historical document-editing events, order them chronologically, and retain the document identifier, local content before and after each edit, and the corresponding document diff. We then use LLM-assisted segmentation to group consecutive events into semantically coherent task units, where each unit corresponds to a user-level objective that could naturally be delegated to an agent. Since real document histories contain noise and interleaved workstreams, we further conduct human verification to refine task boundaries and annotate task-level metadata, including the task objective, whether it belongs to the participant's primary workflow, required external resources, and constraints for later task reconstruction. After human verification, we obtain 1005 task instances in total, including 568 primary-workflow tasks. Detailed annotation guidelines and LLM prompts are provided in Appendix D.

B.2 Privacy Preservation

Real-world workflows contain sensitive information, including personal identifiers, private documents, internal services, local paths, and credentials. To preserve privacy while maintaining task structure, we apply a structure-preserving anonymization process that combines rule-based detection with agent-assisted verification. Specifically, sensitive entities are consistently replaced across the entire task sequence, private domains and paths are normalized, and credential-like information is replaced with format-preserving placeholders. Public resources and non-sensitive information are retained when possible. This procedure preserves entity relationships, structural patterns, and task dependencies without exposing private content.

B.3 TASK RECONSTRUCTION

Through the steps described in Section B.1, we transform raw workflow records into segmented worklog entries, each corresponding to a user-level task that could in principle be delegated to an agent. However, these entries primarily capture what changed after a human completed the task, rather than the explicit instruction and execution context required for an agent to perform it. We therefore reconstruct each segmented entry into an executable agent task instance. Specifically, using the worklog evidence and human-annotated metadata obtained in the previous step, we independently construct each instance through three stages: instruction generation, observation and environment construction, and execution-based verification. We describe these stages below.

Instruction generation. For each task represented by a segmented worklog entry, we use the LLM to reconstruct its underlying work objective. The model considers the evidence associated with the entry—including the pre-task and post-task document states and the document diff—together with task-level annotations and any available historical context. It is prompted to infer a plausible user request that could have produced the observed outcome and to express that request as a natural agent instruction. The resulting instruction specifies the task goal, necessary resources, expected outcome, and key constraints, while remaining goal-oriented rather than prescribing a step-by-step procedure. This formulation allows the agent to plan its own execution while reducing the risk of leaking the final document content. The prompt used for this request-inference and instruction-generation process is provided in Appendix D.

Observation and environment construction. Real worklogs often omit observations that were available to the human worker but are not preserved in the document history. These missing observations may include:

- meeting discussions or informal decisions;
- local data files or intermediate artifacts;
- experiment outputs or model evaluation results;
- third-party services, APIs, or tool responses;
- results that depend on unstable tool versions or expensive resources such as GPUs.

We construct task observations using one of two mechanisms, depending on whether the underlying resources can be faithfully, safely, and practically reproduced. When a resource is static and recoverable with high confidence from the workflow evidence, we materialize the relevant information in the container environment as local files, structured notes, or summaries. We similarly encode lightweight meeting or discussion context as structured notes when it is needed to preserve dependencies between consecutive tasks.

When a resource cannot be directly exposed or practically restored, we emulate access to it through a controlled mock API. The evaluated agent submits a query to the API in the form of a prompt, a structured request, or a code snippet. All such inputs are treated purely as text. A dedicated simulator agent then generates a task-relevant response, such as a statistic, summary, or simulated tool output, conditioned on the query and on known properties of the resource derived from the workflow records and human annotations. The responses are designed to be consistent with the role of the resource in the original workflow. This mechanism enables the evaluated agent to interact with otherwise unavailable resources without exposing private raw data or requiring the original resources to be restored and operated.

Reconstruction loop. We validate each reconstructed task through an execution loop. Given the generated instruction and reconstructed files or APIs, an agent executes the task in the prepared container environment. A verifier then compares the agent's result with the outcome recorded in the original worklog, judging whether they correspond to the same underlying task rather than requiring exact textual matching.

If the verifier determines that the execution corresponds to a different task, it produces revision feedback for refining the instruction, and the task is executed again. If the execution targets the correct task but fails during the process, we conduct human review to determine whether the reconstructed observations or environment are incomplete, and supplement them when necessary. Once the agent

successfully performs the same task as reflected in the worklog, we retain the validated instruction, reconstructed observations, execution trajectory, and container state for benchmark construction. This loop ensures that each task is semantically faithful to the original workflow and executable in a controlled environment.

B.4 Controlled State Progression and Trajectory Alignment

A challenge in constructing sequential agent benchmarks is that open-ended task execution can introduce execution drift. For many tasks, there may be multiple reasonable ways to complete the same user objective, and differences in intermediate decisions, file edits, or tool usage are not necessarily errors. However, if an agent freely executes the entire task stream from the beginning, these local differences can accumulate and alter future environments, making later tasks incomparable to the original workflow and difficult to evaluate consistently.

To prevent such drift, we align the validated trajectory obtained during task reconstruction to the real workflow outcome. Given the original pre-task and post-task workspace states, we rewrite the agent trajectory into a result-aligned trajectory. The rewritten trajectory preserves the high-level execution pattern of the agent, such as the sequence of observations, reasoning steps, tool calls, and file operations, but adjusts the concrete outputs so that applying the trajectory produces the same final workspace changes as recorded in the worklog. We then apply this aligned trajectory to update the container state, obtaining a canonical post-task environment for the next task.

This design gives every task a consistent pre-task history and execution environment, while still preserving realistic trajectory information from prior tasks. It also makes the benchmark more efficient: once canonical pre-task states are prepared, target tasks can be evaluated independently or in parallel. More importantly, by eliminating cumulative drift from earlier executions, the measured performance on each target task is more directly attributable to how the memory component uses fixed historical experience, rather than to uncontrolled variations in previous task execution.

B.5 Metric Generation

We prioritize evaluating whether memory improves downstream agent performance. Therefore, we use two primary outcome metrics: Workspace Score, which measures whether the task is successfully completed, and Preference Score, which measures whether the agent preserves participant-specific working preferences. In addition, we introduce four diagnostic metrics, including Relevance, Continuity, Solvability and Hallucination Robustness, to better understand how memory influences agent behavior.

Preference Score. User preferences often reflect implicit requirements beyond task completion, such as collaboration habits and information organization styles. To capture preferences that persist across tasks, we construct participant-specific preference rubrics in two stages: human annotators first define root preferences, and an LLM analyzes historical message logs and document changes to derive additional cross-task rubrics. For each task, applicable rubrics are identified from the reference document changes, and an LLM judge evaluates whether the candidate trajectory's document writes satisfy these preferences. The weighted rubric scores are normalized to a 0–100 Preference Score. We additionally report an anonymized pairwise win rate between with-recall and without-recall trajectories under the same preference criteria.

Workspace Score. Workspace Score evaluates task completion based on the final Docker environment rather than the agent's response. Its reference is the validated post-task workspace obtained during task reconstruction. The evaluator derives task requirements from the reference workspace at three levels: minimum completion, reference-level quality, and above-reference improvements. Candidate code, documents, configurations, and artifacts are evaluated against these requirements, with additional execution checks applied when necessary. Scores are normalized to 0–100, where approximately 60 indicates minimum completion, 80 indicates reference-level quality, and higher scores require clear improvements in correctness, coverage, validation, usability, or robustness.

Memory Diagnostics. Workspace and Preference scores show whether memory improves the final result, but not how recalled history affects execution. We therefore examine both retrieval quality

and its behavioral effects. Since real workflow traces do not specify which earlier tasks are useful, an LLM compares each evaluation task with all earlier tasks in the participant's workflow, identifies relevant predecessors, and extracts supporting excerpts from their agent traces. Together, they form the reference historical evidence. We organize our diagnostics around four questions:

- **Relevance: Does memory recall the history needed by the current task?** We compute pairwise cosine similarities between recalled chunks and reference excerpts using an embedding model. Mean similarity averages the best match for each reference excerpt. Given a fixed threshold, recall and precision measure the matched proportions of reference excerpts and recalled chunks, respectively. In addition, an LLM judge assigns a 1–5 relevance score to the actual recall.
- **Continuity: Does memory reduce repeated exploration?** We use an LLM to classify every tool call as exploration (searching or reading), execution (modifying, running, or validating), or other. We compare their counts and proportions. In addition, an LLM assigns a 1–5 environment-familiarity score to the overall trajectory.
- **Solvability: Can memory solve problems left unresolved without recall?** An LLM reads the no-recall trajectory to identify execution problems and determine whether each is resolved by the end of the trajectory. For each unresolved problem, another LLM judges whether the actual recall contains sufficient information to resolve or avoid it. We report the proportion of unresolved problems that are solvable using the recalled memory.
- **Hallucination Robustness: Does memory cause new execution problems?** To measure errors introduced by memory, an LLM first identifies execution problems from the with-recall trajectory. For each problem, another LLM checks whether it is caused by misleading recall. We report the proportion of tasks containing at least one memory-induced problem. In addition, an LLM assigns a 1–5 hallucination-robustness score to the overall trajectory.

C Data Statistics

We report the token distribution of converted message logs, anonymized participant-level history lengths, multi-label task type annotations, token statistics by task type, and the temporal locality of relevant historical tasks. These statistics complement Section 5.1 by giving more detailed evidence about the scale, diversity, and long-horizon dependency structure of the benchmark.

Table 3: Token statistics of message logs.

n	Mean	P25	Median	P75	Min	Max
1005	36.3K	8.4K	21.2K	56.1K	641	212.3K

Table 4: Anonymized participant-level pre-task history token statistics. Participants are grouped by the number of core evaluation tasks. For each core target task, history includes all previous sessions from the same participant, including both core and non-core tasks.

Group	Ppl.	All	Core	Non-core	Mean	P25	Median	P75	Max
0–9 core tasks	1	4	3	1	4.3K	1.9K	3.9K	6.5K	9.0K
10–19 core tasks	2	38	32	6	82.1K	18.3K	48.8K	109.5K	351.2K
20–39 core tasks	4	226	104	122	927.3K	183.6K	685.1K	1.54M	3.13M
40–59 core tasks	3	251	155	96	917.5K	141.7K	576.3K	1.18M	4.01M
60–79 core tasks	4	486	274	212	1.59M	255.4K	962.6K	2.25M	8.43M
Overall	14	1005	568	437	1.19M	153.6K	654.1K	1.77M	8.43M

D Prompts and Annotation Protocol

This appendix reports the prompt templates and human annotation protocol used in our data construction pipeline.

Table 5: Multi-label task tag distribution over 568 core tasks.

Tag	n	% of 568
Writing	223	39.26
Data Analysis	138	24.30
Coding	136	23.94
Planning	106	18.66
Tool Use	101	17.78
Debugging	69	12.15
Deep Research	51	8.98
Review	42	7.39
Data Process	32	5.63

Table 6: Message log token statistics by task type.

Tag	Tasks	Mean	P25	Median	P75	Max
Writing	223	46.9K	12.1K	25.4K	65.9K	212.3K
Data Analysis	138	44.6K	12.4K	27.0K	61.9K	184.1K
Coding	136	36.1K	9.0K	18.2K	52.6K	188.3K
Planning	106	42.2K	11.6K	22.3K	64.1K	188.3K
Tool Use	101	42.7K	9.6K	17.6K	60.7K	188.9K
Debugging	69	51.5K	15.2K	29.9K	86.4K	183.0K
Deep Research	51	46.6K	5.9K	18.6K	96.7K	185.8K
Review	42	49.9K	9.9K	36.2K	82.2K	167.4K
Data Process	32	21.2K	12.0K	15.0K	17.6K	112.5K

D.1 Task Segmentation

We first apply a rule-based temporal pre-merge to nearby document-editing events. An LLM assistant then decides whether each candidate temporal chunk should be merged into the current task or split into a new task. The prompt emphasizes fine-grained task boundaries and treats over-merging as more harmful than mild over-splitting.

Task Segmentation Prompt

You are given a segmented worklog entry together with its associated You are merging consecutive temporal worklog chunks into larger tasks.

You are given: 1) the current merged worklog group, 2) one candidate next temporal chunk, 3) a short lookahead of later temporal chunks.

Important setup: - Before you see the payload, a rule-based pre-merge has already grouped very close-in-time edits into temporal chunks. - These temporal chunks are only there to reduce merge pressure on the model. - The final saved task will still keep the finest-grained original event ids. - So your job is only to decide whether the candidate chunk should join the current task, not to reason about UI granularity.

Goal: - Decide whether the candidate next temporal chunk should be merged into the current task. - Prefer a fine-grained segmentation. - Be actively willing to split tasks. - Unless there is strong evidence that the candidate chunk is still part of the same concrete task, lean toward SPLIT rather than over-merging. - When uncertain, prefer SPLIT over MERGE. - In this task, over-merging is worse than mild over-splitting. - Be comfortable splitting aggressively when there is any plausible boundary. - Do not treat MERGE as the default. Treat SPLIT as the safer default unless the evidence for MERGE is genuinely strong. - Do not keep extending the same merged group by default. Re-check the boundary fresh at every step. - Especially when the current merged group is already long or contains many accumulated edits, be more willing to

Table 7: Temporal locality of relevant previous tasks. Among 568 core tasks, 541 have at least one relevant previous task, with 8084 relevant links in total.

Mean subtask-id distance	Subtasks	% among nonzero
0–10	261	48.24
10–20	148	27.36
20–50	114	21.07
50+	18	3.33

SPLIT rather than continue merging. - A task should usually correspond to one coherent goal / artifact / investigation / pipeline stage, not one tiny worklog edit.

Payload field meanings: - temporal_premerge.median_gap_seconds / merge_gap_threshold_seconds: the rule-based timing statistics used to build temporal chunks. These are only context; they are not direct merge commands.

- group_chunk_count: how many temporal chunks are already inside the current merged group before considering the candidate.

- group_event_count: how many raw worklog events are already inside the current merged group before considering the candidate.

- large_group_split_hint: a stronger reminder string that appears when the current merged group is already large (¿15 events). Treat it as a prompt to actively inspect the current boundary and look for a reasonable place to split.

- group_markdown_view.group_start_markdown: the full markdown snapshot from the beginning of the current merged group, before the first event in the group happened.

- group_markdown_view.current_markdown: the full markdown snapshot of the current merged group after all chunks already in the group have been applied. This is the main current state of the merged group.

- group_markdown_view.candidate_after_markdown: the full markdown snapshot you would get if the candidate chunk were merged into the group. Compare this against current_markdown to judge whether the candidate still looks like the same task.

- group_markdown_view.large_markdown_omitted / omission_note: when the full markdown views would be too large, some or all of them may be intentionally omitted.

- group_markdown_view.before_omitted: when true, group_start_markdown was omitted.

- group_markdown_view.candidate_after_omitted: when true, candidate_after_markdown was omitted.

- group_markdown_view.local_only_mode: when true, all full markdown views were omitted, including the before view. In that case, rely only on local chunk markdown and change summaries.

- group_chunks: concise summaries of the temporal chunks already inside the current merged group.

- candidate_chunk.markdown_view.before_local_markdown: local markdown near the candidate chunk's focus path before that chunk happens.

- candidate_chunk.markdown_view.after_local_markdown: local markdown near the candidate chunk's focus path after that chunk happens.

- candidate_chunk.change_view: auxiliary structured metadata about the candidate chunk, including the raw events inside it.

- lookahead_chunks[*].markdown_view.before_local_markdown / after_local_markdown: local markdown around the next few future temporal chunks. These are only short future hints, not the main evidence.

Important:

- Do not force everything under one broad heading to stay together. If the work under the same heading clearly branches into different concrete tasks, you may split it.

- It is acceptable to produce more tasks when that makes each task map more cleanly to one concrete thing a person or agent would actually do.

- If merging would create one broad mixed bundle that really contains multiple plausible standalone tasks, choose SPLIT.

- A large existing merged group is itself a warning sign against further merging. Do not let momentum alone keep unrelated or weakly related edits inside the same task.

- If ‘group_event_count‘ is already greater than 15, explicitly ask whether the group has already become too broad and whether a split is now warranted.

- If ‘large_group_split_hint‘ is present, do not passively note it; actively inspect the current boundary and look for a reasonable split point.

- Prefer group_markdown_view.current_markdown and group_markdown_view.candidate after markdown plus the candidate_chunk local BEFORE/AFTER markdown as the primary evidence.

- Treat focus paths and change summaries only as auxiliary hints.

- If a heading, bullet, or label clearly introduces content that continues immediately after it, keep that continuation together with the introduced content.

- In particular, patterns like ”Prompt design: ”, ”Result: ”, ”Problem: ”, ”Development process: ”, or similar label-plus-colon structures usually belong with the text or list items that immediately follow, so do not split them apart unless a clearly different task begins later.

- Use lookahead carefully when a date heading is still being formed.

- If a genuinely new date bucket appears, split immediately at that point.

- THIS RULE IS IMPORTANT: a genuinely new date bucket starts a new task immediately.

- If you say in the reason that the candidate introduces a genuinely new date section / new day bucket, then the final decision must be SPLIT, not MERGE.

- This date rule has higher priority than ”the current group is still small”, ”the work is still related”, ”lookahead continues the same thread”, or any other merge-leaning heuristic.

- So even if the current merged group is tiny, even if the broader topic is the same, and even if the lookahead keeps filling the new date section, a genuinely new date bucket must still SPLIT.

- If the markdown delta is subtle or nearly invisible, use the change-view / focus-view to understand what was edited.

- The payload may explicitly tell you whether the normalized BEFORE/AFTER markdown is identical; if so, that is a very strong signal to merge, because the edit likely did not change substantive content.

- If the markdown shows no substantive content update, prefer merging by default.

- A temporal chunk may contain multiple tiny edits that happened very close in time. Do not split just because the chunk itself contains multiple raw events. Judge whether the candidate chunk, as a whole, still belongs to the same concrete task.

- Pure formatting changes, structural cleanup, heading reshaping, or other non-content edits usually should not start a new task on their own.

- If the previous edit introduced or reshaped a large block of content and the candidate mainly edits a smaller piece inside that same newly introduced block, that usually should MERGE.

- In other words: ”big block added, then a local refinement inside that block” is normally one continuing task, not a new one.

- A small current merged group is only a weak secondary heuristic. It must not outweigh clear evidence for SPLIT.

- Early in a task, sparse context is not by itself a reason to merge. If the candidate already looks like a separate plausible task, choose SPLIT.

- Conversely, when the current merged group is already large, you should require strong positive evidence to keep merging. If that strong evidence is missing, choose SPLIT.

- Once the merged group becomes large, the burden of proof flips: do not ask ”can I keep merging?” Ask ”is there strong evidence that this still must stay together?” If not, output SPLIT.

- For evaluation / benchmarking / model-comparison work, a model switch often suggests a new task, because the target under evaluation may have changed even if the surrounding section title remains similar.

- Treat this as a soft hint rather than a hard rule: use the full markdown context to decide whether the work is still one coherent task or has shifted to a new evaluation target.

- If the edit mainly creates a new date section such as ”8.27”, ”9.14”, or similar day markers, and that is a genuinely new day subsection rather than a revision of an old heading, that new date section itself should be treated as the new task boundary.

- Do not write a reason like ”this is a genuinely new date section, but still merge because it is part of the same weekly thread”. That is incorrect for this task.

- But once you are already inside one date bucket, that day can still contain multiple tasks.

- If one day clearly contains multiple different goals, deliverables, investigations, or pipeline stages, you should still split within that day at the appropriate boundary.

- So the same date can contain multiple tasks, but introducing a genuinely new date bucket is itself already a split signal.

- When the candidate touches dates or day buckets, structure your reason in this order:

- First, explicitly state whether the candidate introduces a genuinely new date section or merely edits an existing one.

- Second, explicitly state whether the lookahead shows continued content under that same new date section.

- Third, state the merge / split conclusion.

- Keep this reasoning concise, but do include these date checks in the reason whenever dates are involved.

- End the reason with exactly one final sentence in this exact format:

- ‘FINAL DECISION: I will MERGE.‘

- or ‘FINAL DECISION: I will SPLIT.‘

- Do not use any other final wording, and do not include both.

- Return STRICT JSON only.

Date Rules:

- If the edit is mainly renaming an existing heading into a date label, correcting an existing date label, or otherwise revising a date marker that already belongs to the current thread, that usually should MERGE with the current task.

- If the candidate chunk creates a genuinely new date subsection / new day bucket, it MUST start a new task immediately.

- A transition like ”8.19” -¿ ”8.20” is therefore itself a split boundary when 8.20 is a genuinely new date section.

- Do not override it because the current group is small.

- Do not override it because the candidate looks like a continuation of the previous day’s work.

- Do not override it because the lookahead keeps adding content under the new date.

- If it is a genuinely new date bucket, output SPLIT.

- When judging this, compare current_markdown with candidate_after_markdown: if the candidate clearly opens a new day container rather than merely revising the existing one, output SPLIT.

D.2 Human Annotation Protocol

After LLM-assisted segmentation, annotators reviewed the candidate tasks through an annotation interface. This step did not use an LLM prompt. Annotators followed a simplified standard operating procedure:

1. Inspect each candidate task together with its document snapshots, local markdown diff, temporal context, and neighboring tasks.
2. Verify whether the candidate should remain separate or be merged with adjacent tasks. The target unit is a coherent and relatively atomic work objective that could naturally be delegated to an agent.
3. Mark whether the task belongs to the participant's main workflow.
4. Write a concise task description that captures the work objective, boundary, and expected outcome.
5. Write a metric description that identifies the evaluation focus for the task.
6. Assign applicable agentic tags, such as coding, planning, debugging, data analysis, writing, tool use, review, or deep research.
7. Select applicable participant-level preference metrics.
8. Mark whether additional files or intermediate resources are needed. If so, describe the required resource traits, format, granularity, and content pattern.

Annotation

{
”annotation_status”: ””,
”task_description”: ””,
”metric_description”: ””,
”selected_preference_metric_ids”: [],
”agentic_tags”: [],
”file_supplement_needed”: ””,
”file_supplement_traits”: ””,
”updated_at”: ””
}

D.3 Instruction Generation and Task Reconstruction

Given a segmented worklog entry, we infer the underlying work objective and then convert it into an executable agent task. The first prompt asks the model to infer what work likely produced the observed document change. The second prompt converts that inferred work into a natural agent instruction with resource dependencies, success criteria, expected outputs, and caveats.

Instruction Generation Prompt

You will receive the following information:

- The main task currently being handled by the user

- Task comments for the current subtask

- A partial snippet of ‘before_markdown‘ / ‘after_markdown‘

- A brief summary of the markdown diff

- A summary of the historical work and most recent version related to this document prior to the current task. This summary comes from the incremental document state maintained in subtask order.

- The markdown diff itself, if necessary.

Your task is not to write agent instructions, but to infer: what work was actually done this time.

Requirements:

- You need to infer ”what the executor most likely did to produce this document change,” not just restate the results already written in the document.

- You need to combine the main task, ‘task_description‘, and document differences to make this judgment.

- If historical context of ”what was done before the current task” is provided, treat it as an important clue: the current subtask is often a continuation, supplementary verification, summary, or status confirmation based on this existing work.

- If the ‘task_description‘ is ”summary/organization/induction”, but the document differences show new additions such as research results, example summaries, rule summaries, or status confirmations, then you should try to revert to the original description of ”first conduct research/verify/design/screen, then form a summary or conclusion,” rather than simply ”organize existing content.”

- ‘summary‘ should be a concise summary of the actual work done.

- ‘primary_goal‘ should describe the core objective of the work itself, not the document update objective.

- ‘likely_actions‘ should list 2-5 high-level actions, describing what is typically done to complete this task.

- ‘result_signal‘ should summarize the resulting signals from this document change.

- ‘notes‘ should only contain necessary conservative explanations or points of uncertainty.

- If the evidence is insufficient, set ‘status‘ to ‘insufficient_evidence‘, but still try to provide conservative inferences.

Returns STRICT JSON, returning only one JSON object.

Instruction Generation Prompt

You will receive:

- A pre-inferred ‘inferred_work‘

- A list of optional middleware interfaces

Your job is to transform the ‘inferred_work‘ into a fully executable agent task.

Requirements:

- The output ‘agent_task.instruction‘ should be natural, specific, and executable.

- The instruction should read like a person handing over a task to another agent, not like a data summary or debriefing report.

- The instruction should be a high-level task handover, not written like an SOP, troubleshooting manual, or step-by-step guide.

- If this is an inspection/analysis/verification task, you can point out 2-4 aspects to focus on, but don't specify overly detailed execution orders, interface call rounds, or question templates.

- If the input provides interfaces, the instruction should naturally explain "what information you can obtain using these interfaces," but don't create new interfaces.

- Don't directly copy the final before/after text; extract the underlying task objectives and state changes.

- Avoid using specific target document wording in the instruction; do not write final, text-level statements like "change the status from A to B." Only abstract expressions like "determine whether a status upgrade/tightening/supplementing stronger conclusions are supported."

- Key point: ‘agent_task‘ describes what the task itself needs to accomplish, not how the document will ultimately be written. Document modifications are merely traces of the completed task.

- For data inspection tasks, the real goal should be "determining whether the data is usable, its quality, and what the limitations are," not "determining whether it is sufficient to support document updates."

- If evidence is conflicting or insufficient, be conservative, but avoid vague statements.

- Only restore ‘agent_task‘; do not output ‘writeback_task‘ or ‘evidence_summary‘. How the document is written back is not part of the current fixed process.

- ‘inferred_work.primary_goal‘ describes the actual task objective; prioritize it.

- ‘inferred_work.likely_actions‘ is for high-level reference only; do not mechanically copy it as a checklist.

Good style example of instruction:

"Examine existing GPT distillation data to determine its current usability and whether there are any constraints that require retention. You need to focus on overall usability, consistency of output quality, whether there are obvious problem patterns, and whether the existing evidence is sufficient to support a more explicit quality judgment. You can use the query_xxx interface to obtain the overall conclusion, main problems, and coarse quantification basis. Finally, give a clear but conservative usability conclusion and supplement it with a minimal evidence summary."

Returns STRICT JSON, returning only a single JSON object.

D.4 Meeting Discussion Rewrite

Some static resources represent informal meetings or discussion records. We rewrite these records into a consistent dialogue style while preserving task-relevant facts.

Meeting rewrite prompt

You are editing resource JSON files inside one Membench resource ‘trace‘ directory.

Current working directory: {trace_dir}

Candidate JSON files: {candidate_paths}

Primary meeting files detected by code: {primary_meeting_paths}

Primary meeting hint note: {primary_meeting_hint_note}

Task: Some meeting files are detected by code: a JSON object with a top-level ‘messages‘ array whose items have this shape: ‘{{"speaker": "...", "content": "..."}}‘ Those files are definitely meeting/discussion records.

The code-detected list is only a hint. Even if it is empty, you must still inspect all candidate JSON files in this same resource directory to see whether any meeting files were missed. A missed meeting file must still have explicit speaker/content style dialogue, for example a list of objects with speaker-like and content-like fields. If a file has no speaker/person field paired with spoken content/text, it is not a meeting file.

If you do not find any meeting file, do not modify any file. End with no changes. This is mandatory.

You may freely rewrite the meeting dialogue structure inside the message list: split one long message into several shorter turns, merge adjacent turns when natural, reorder nearby turns if it makes the conversation flow better, and adjust speaker turn distribution. The goal is to make the dialogue read like a real meeting conversation instead of stiff reconstructed notes.

Output shape requirement for every meeting file you rewrite: The final JSON file must contain only one top-level key: ‘messages‘. Delete every other top-level key, including fields such as ‘record_type‘, ‘topic‘, ‘participants‘, ‘student‘, ‘time_window‘, ‘time_scope‘, ‘source_basis‘, ‘level_notes‘, ‘constraints‘, ‘notes‘, ‘summary‘, ‘metadata‘, and any similar non-message fields. Do not preserve the original top-level schema. Before deleting those fields, inspect them: if they contain meaningful facts, constraints, dates, participants, source notes, task decisions, TODOs, blockers, or other useful information, fold that information naturally into the dialogue turns in ‘messages‘. If a field is only administrative bookkeeping and contains no useful information for the later agent, discard it. After rewriting, the file should look like ‘{{”messages”: [{{”speaker”: ”...”, ”content”: ”...”}}, ...]}‘ and nothing else.

This is especially important for structured context files where useful information may live outside ‘messages‘, for example keys like ‘strategy_chain‘, ‘dedup_policy‘, ‘script_anchors‘, ‘observed_dedup_statistics‘, ‘prompt_anchors‘, ‘rules‘, ‘requirements‘, ‘statistics‘, ‘examples‘, or other nested objects and lists. You must not leave those as separate keys. Read them, decide which facts matter for a later agent, and turn only the useful task-relevant facts into natural meeting dialogue. If a field or nested value is unimportant, redundant, purely administrative, or not needed for understanding the task, it is better to discard it than to force it awkwardly into the conversation. Do not paste JSON-like bullet lists into a single message, and do not write stiff lines like ”metadata says...” or ”the strategy_chain is...”. Instead, make it sound like real people are talking through the information: one person asks or confirms, another person explains, they clarify important numbers, file names, thresholds, model names, and constraints in short conversational turns. The final file must still contain only ‘messages‘.

The one thing you must not change is the useful task meaning. Preserve task-relevant decisions, facts, constraints, task details, names, dates, numbers, model names, file paths, URLs, examples, conclusions, TODOs, blockers, and assignments when they matter for understanding or executing the task. Do not add new information. You may discard unimportant administrative details or redundant metadata that a later agent does not need.

Strong style priority: Make the result as daily, conversational, and meeting-like as possible. This is very important. Prefer natural back-and-forth dialogue over polished written summaries. It should feel like people are talking in a sync meeting, clarifying points, agreeing, interrupting lightly, confirming details, and deciding how to record the work.

Do not rewrite ordinary documents. Files without speaker/content style dialogue are not meeting rewrite targets. Docs/docx/wiki/tutorial/onboarding/excerpt/readme resources, document summaries, internal documentation snippets, API instructions, benchmark artifacts, and reference material are not meeting files, even if they are useful context.

Very important constraints:

- Only edit files under the current working directory.

- You may edit the ‘messages‘ array itself when needed: split turns, merge turns, reorder nearby turns, add short natural acknowledgements, and adjust speaker distribution.

- Keep each message item in speaker/content style.

- For every meeting file you rewrite, delete all top-level fields except ‘messages‘.

- Important non-message information must be moved into natural dialogue before the original field is deleted.

- Unimportant, redundant, or purely administrative non-message information should be deleted without forcing it into dialogue.

- For nested structured fields outside ‘messages‘, convert important values into realistic back-and-forth conversation instead of preserving them as schema keys or dumping them as lists.

- The final rewritten meeting JSON must have exactly this top-level shape: one object with one key named ‘messages‘.

- Do not edit non-candidate files.

- Do not edit files outside this directory.

- Do not edit middleware config files such as ‘middleware/registry.json‘, ‘middleware/hidden_context.json‘, or ‘middleware/agent_config.json‘.

- Do not edit docs, doc excerpts, onboarding docs, wiki excerpts, tutorial notes, non-meeting data files, experiment outputs, benchmark rows, schemas, manifests, code configs, or pure artifact JSON.

- Do not turn a document excerpt into a conversation. If the source is a document, leave it as a document.

- If no candidate has explicit speaker/content-style dialogue, leave every file unchanged.

- Preserve valid JSON. The file must still parse as JSON after editing.

- Do not preserve the surrounding top-level JSON schema for meeting files. Keeping only ‘messages‘ is required.

- Do not change factual content: names, dates, numbers, task decisions, model names, file paths, URLs, project names, conclusions, TODOs, blockers, and assignments must remain the same.

- Do not invent new facts, decisions, people, claims, or results.

- Do not delete useful information.

- The goal is style only: less stiff, less synthetic, more like real Chinese meeting speech.

Target style: - Natural daily spoken Chinese, with realistic starts/stops, short fragments, mild repetition, and informal connectors.

- * Strongly prefer realistic conversation or meeting-note dialogues: short turns, acknowledgements, follow-up questions, clarifications, ”yes, yes”, ”hmm”, ”OK”, ”right?”, ”let’s leave it here for now”, and similar natural meeting flows.

- It is better to split a stiff long paragraph into several natural turns when that makes the exchange feel like an actual meeting.

- * It can use phrases such as ”it’s just”, ”I think”, ”yes, yes”, ”right?”, ”then”, ”maybe”, ”let’s leave it for now”, ”we can come back to it later”, ”how about”, ”this thing”, ”more or less”, ”not really workable”, and ”we can give it a try”.

- It should feel like people are talking in a meeting, not like a formal report.

- Do not overdo filler words. It should be human, not parody.

- Keep technical terms intact: API names, model names, dataset names, benchmark names, file names, commands, links, and numbers should stay recognizable.

Style reference as JSON, do not copy facts from this reference: {style_reference}

Few-shot examples: {fewshot_examples}

Now inspect the candidate files and make the edits directly. If there is no meeting-like JSON in this directory, leave every file unchanged.

D.5 Instruction Alignment Loop

After an agent task is reconstructed and executed, an alignment checker determines whether the execution corresponds to the work implied by the original document diff. If the checker finds a mismatch, a reviser rewrites the agent task while preserving the intended work semantics.

Execution alignment prompt

You will receive:

- The original worklog markdown diff

- The inferred work from pipe2

- The agent task constructed by pipe2

- The final message and major file changes after the agent's actual execution

Your task is to determine whether this agent task and its actual execution trajectory are performing the type of work reflected in the original worklog diff.

Judgment rules:

- Focus only on whether the actions performed are correct, not on whether the specific content matches the markdown diff.

- If the worklog is about research/verification/analysis, and the agent is also conducting related research/verification/analysis, even if the research conclusions, material sources, and details are inconsistent with the markdown diff, it should still be judged as ‘action_aligned‘.

- If the worklog is about accumulating prompts/rules/plans/summaries, and the agent is also producing similar specifications/plans/summaries, even if the text content is different, it should still be judged as ‘action_aligned‘.

- Do not predefine limited types for worklogs, and do not rely on fixed keyword enumeration. You need to first analyze the worklog using open text: what specific content was added/modified; what actions did this content imply the executor actually perform; which are definitive results, and which are pending tasks or process logs.

- Then deduce ‘expected_agent_action‘: what the agent should actually do if this worklog were to be reconstructed into an agent task. ‘expected_agent_action‘ must be open text and can be something like ”ensure X runs successfully/attempt to start and verify X/download and check X/complete and run X/organize the prompt word framework for X,” etc., not selected from fixed categories.

- If the worklog contains definitive execution results, such as something running successfully, completed, downloaded, verified, fixed, or implemented, you should usually deduce an action that can accomplish this, rather than simply having the agent confirm whether this event exists.

- ”Ensure/attempt to run X successfully” and ”confirm whether X has run successfully” are not the same task: the former requires the agent to attempt to start, configure, invoke, and verify X, and record observable evidence or reasons for blocking; the latter only audits the current state. When the worklog shows that X has been successfully executed, the revision should change the instruction to a specific execution task related to ensuring X's success, rather than simply making a yes/no judgment.

- If the agent_task weakens, shifts, or changes the expected_agent_action, for example, changing ”ensure X is successfully executed” to ”confirm whether X has been successfully executed,” it should be judged as ‘action_mismatch‘, and the issue_types should include ‘state_presupposition_mismatch‘ or ‘wrong_action_type‘.

- Only when the agent_task or execution trajectory has clearly changed to a different type of work should it be judged as ‘action_mismatch‘. For example: originally investigating data filtering rules, but instead implementing an unrelated webpage; originally organizing plans, but instead modifying business code; originally verifying resources, but only writing an unrelated summary.

- Do not directly judge an action as wrong simply because the agent's final conclusion contradicts the conclusion in the worklog diff; this may simply be due to different results from a re-investigation. Only when such a contradictory conclusion indicates that the agent was asked to perform a different type of task that it shouldn't have been doing should it be considered supporting evidence.

- If the evidence is insufficient to make a judgment, return 'needs_review'; do not force a judgment of aligned or mismatch.

- Provide a structured 'revise_note' and a specific, actionable 'suggested_reconstruct_guidance' to reconstruct the 'agent_task'.

- The 'revise_note' should contain seven types of information:

1. 'worklog_observation': What definite content, process records, or pending signals are in the worklog.

2. 'expected_agent_action': What the agent should actually perform, deduced from the worklog. This must be open text; do not use type enumeration.

3. 'instruction_gap': The difference between the current 'agent_task' and 'expected_agent_action', such as weakening, offsetting, changing the task, or auditing without execution.

4. 'instruction_problem': Where the current instruction is wrong; point out the task structure that was incorrectly rewritten, rather than criticizing the specific execution result.

5. 'preserve_work_semantics': Which work themes, task stages, or action relationships should be preserved after the correction.

6. 'avoid_instruction_patterns': Which instruction patterns should be avoided after the revision.

7. 'rewrite_strategy': How to rewrite the instruction into a new, executable one.

- 'suggested_reconstruct_guidance' should be a natural language compressed version of 'revise_note', containing three types of information:

1. What is 'expected_agent_action'?

2. What are the differences between the current instruction and 'expected_agent_action'?

3. How should it be rewritten, and which instruction patterns should be avoided?

- Neither 'revise_note' nor 'suggested_reconstruct_guidance' should hardcode specific status conclusions in the worklog as mandatory inherited facts; avoid writing "X must be retained as successful / Y as verified". Instead, write "The worklog implies that the agent task should ensure/attempt to complete X, and log the reason for the blockage if it fails." - If the problem is 'state_presupposition_mismatch', the guidance should explicitly state:

- The current error might be that a task of "ensuring/attempting to complete an action" has been changed into a yes/no audit task of "confirming whether it has been completed".

- The instruction should be changed to an executable task surrounding 'expected_agent_action', such as starting, configuring, calling, verifying, downloading, running, modifying, completing, and cleaning up artifacts.

- If the current environment cannot complete the task, the agent should also log the actions attempted, the specific bottlenecks, and the subsequent required conditions, rather than treating the failed audit conclusion as the main artifact.

- The agent can be allowed to log "current environment cannot be verified/verification failed/environment gap exists," but this should only be as supplementary information and not the primary task objective.

- The 'revise_note' should be specific enough to directly guide the reviser, but should not write complete instructions for the reviser; the complete instruction should be reshaped by the reviser in conjunction with the current 'agent_task'.

- Guidance should be written as a direct suggestion for modification that can be used directly by the reviser, avoiding overly abstract statements.

Returns STRICT JSON, returning only a single JSON object.

Instruction revision prompt

You will receive:

- The inferred work already deduced by pipe2

- The currently reconstructed agent_task

- The LLM checker's assessment and modification suggestions for agent_task/execution alignment

Your task is: to reshape agent_task based on the checker's modification suggestions without modifying the old reconstruction prompt process.

Requirements:

- Use alignment_check.revise_note first; if no revise_note is available, then use suggested_reconstruct_guidance.

- Expected_agent_action in revise_note is the most important reconstruction target; instruction_gap/instruction_problem are used to locate errors in the current task structure; preserve_work_semantics are used to preserve task phases, work topics, and action relationships; avoid_instruction_patterns are used to delete or rewrite dangerous expressions; rewrite_strategy is used to generate new instruction mainlines.

- Only correct the task semantic issues pointed out by the checker; do not replicate specific text to conform to markdown diffs.

- If the checker points out that task prerequisites have been rewritten, for example, changing "ensure/attempt to run, start, verify, download, fix, achieve something" to a yes/no audit of "whether it runs/whether it is completed/whether it is available," you should revise the task structure: Have the agent actually execute, verify, and log evidence or reasons for blockages around the expected_agent_action, rather than simply re-evaluating whether the status is met.

- Do not interpret checker guidance as requiring the inheritance of every specific factual conclusion in the worklog. The focus should be on preserving task phases, work topics, and action relationships.

- Instructions should describe "what the agent should do next," not how the final documentation should be written.

- Keep tasks as high-level, executable handovers; do not write them as step-by-step SOPs.

- Retain reasonable success criteria, outputs, and caveats in the current agent_task, but simultaneously revise any statements that conflict with the checker's opinions.

- If the checker determines it is not an action mismatch, make as few changes as possible or revert to the original task.

- Do not introduce new resources, interfaces, file paths, or specific facts that the checker has no basis for. - Instructions can be allowed to require the agent to log "current environment cannot be verified/verification failed/environmental gap exists," but this should only be used as supplementary information and cannot be the primary objective.

Return STRICT JSON, only one JSON object is returned, the schema is the same as reconstructed_agent_task.json: {

"subtask_id": "string",

"status": "ok — needs_review — insufficient_evidence",

"agent_task": {

"title": "string",

"instruction": "string",

"resource_dependencies": {

"static_files": [{"path": "string", "why_needed": "string"}],

"middleware_operations": [{"name": "string", "purpose": "string", "usage": "string"}]

},
”success_criteria”: [”string”],
”expected_outputs”: [”string”],
”caveats”: [”string”]
} }

D.6 CONTROLLED TRAJECTORY REWRITE

After obtaining a semantically aligned execution, we rewrite the trajectory so that the message log and workspace evolution are natural, executable, and consistent with the validated target document state.

Trajectory rewrite prompt

You are a trajectory rewrite planner. Your task is to reconstruct the real target document diff into a natural, executable, and disk-persistent agent message log.

You will receive:

- Original agent task

- Real target document before/after markdown

- Behavioral summary of the original message log

- Summary of changed files in the workspace after execution

- Summary of the current workspace file tree and candidate target files

The objective must satisfy three constraints:

1. The rewritten trajectory must execute the task from scratch like the real agent, without appearing to know the final document answer in advance.

2. There must be a natural target file write action at the end, and the written content must be verbatim equal to the after_markdown.

3. Before writing the target file, sufficient evidence must have naturally generated in the trajectory; the workspace must also have corresponding supporting files or original changed files.

Important requirements:

- Do not use phrases that reveal the benchmark, such as ”based on the real diff,” ”target answer,” ”not written incorrectly,” or ”verbial matching.”

- Do not hardcode the task type. Plan reasonable execution steps based on the current target document diff.

- Refer to the original message_log pattern: for example, first check the workspace, then query/read files/call APIs/write files/validate/summarize. Do not copy irrelevant content.

- If the target document records research, the trajectory should first show research actions; if it records running/fixing/implementation, the trajectory should first show attempts, verifications, or blocking records; if it records plans/summaries, the trajectory should first show information organization and summarization.

- Supporting files can be generated, but they must be files that the agent will naturally write during execution. The target document should be the final brief record, not the only output.

- The execution_plan should only record the process of ”generating evidence/completing the task,” do not write the final target_path in the execution_plan; the final target file will be appended uniformly by the outer program.

- When planning, first work backward from after_markdown: if this task is executed from scratch, what checks, research, runs, verifications, and organization are needed to naturally obtain these records? Then express this using the interaction rhythm of the original message_log.

- ‘target_path‘ must be determined based on the current workspace file tree, ‘candidate_target_files‘, ‘doc_title/doc_id‘, ‘before_markdown‘ content similarity, and task semantics. Do not default to ‘worklog/worklog.md‘; only select it if current evidence indicates it is the target file.

- Output must be STRICT JSON.

Return schema:

{ ”status”: ”ok” — ”needs_review”,

”execution_plan”: [

{ ”purpose”: ”Natural language explanation of why this stage is necessary”,

”assistant_message”: ”Agent’s explanation of the next steps to the user/itself”,

”tool_command”: ”Optional; shell command or equivalent action”,

”tool_output”: ”Optional; summary output that the command should reasonably see”,

”workspace_writes”: [

{”path”: ”Path relative to /workspace”, ”content”: ”Complete file content”}

]

}

],

”target_path”: ”Target file path relative to /workspace”,

”target_path_reason”: ”Why this file was chosen as the final write target”,

”target_content”: ”Must be verbatim equal to after_markdown”,

”worklog_path”: ”Legacy-compatible, optional”,

”worklog_content”: ”Legacy-compatible, optional”,

”worklog_line_support”: [{”line”: ”A line in the target document”, ”supporting_plan_indices”: [0, 1], ”supporting_paths”: [”...”]}

],

”notes”: [”Any points requiring human attention”]

}

D.7 Evaluation Prompts

The following boxes reproduce the prompts used to construct and evaluate the reported metrics. Task-specific inputs are supplied at runtime; the boxes retain the evaluation instructions and required output schemas.

D.7.1 Downstream Evaluation Prompts

Workspace Score

You are a strict workspace evaluator for one Membench subtask.

Module rubric:

‘workspace_task_completion‘

Question:

Did the final Docker workspace actually complete the task implied by the worklog/document change and the task instruction?

Use evidence from:

- ‘./input/task_context.json‘, especially ‘task_description‘, ‘metric_description‘, ‘planned_tasks‘, ‘inferred_work‘, and ‘agent_task‘.

- ‘./input/worklog_ground_truth.json‘, especially ‘worklog_changes[*].before_markdown‘ and ‘worklog_changes[*].after_markdown‘.

- ‘./input/agent_trace.json‘, optional execution evidence from the evaluated run.

- ‘./input/rollout.jsonl‘, optional supporting raw evidence.

- ‘bundle.docker.candidate_image_ref‘.

- ‘bundle.docker.reference_image_ref‘.

Core workflow:

1. Read the task context and reference document change to understand the actual task.

- Identify the task type: research, analysis, code/config/environment work, prompt/rule/pipeline work, data work, or mixed.

- Identify the core deliverable and the quality signal the workspace should contain.

- Do not infer requirements from filenames alone.

2. Inspect ‘bundle.docker.reference_image_ref‘ first.

- Treat it as the solid reference baseline for this task, roughly around the 80-point level.

- It is not perfect and does not need to be copied.

- Extract what it does well as content/task-quality requirements, not as path or filename requirements.

3. Write ‘./output/criteria.json‘ before inspecting the candidate.

- ‘minimum_requirements‘: what a barely passing workspace must accomplish for this task.

- ‘reference_requirements‘: what the reference workspace accomplishes at a solid level.

- ‘better_requirements‘: what would be meaningfully better than the reference workspace.

- Good requirements describe task quality: correctness, completeness, depth, breadth, concrete evidence, validation, robustness, usefulness, and actionability.

- Bad requirements overfit to filenames, paths, exact document names, or exact wording.

- Write enough requirements to cover the task in detail. Small tasks may need 5-8 requirements; broader research/code/analysis tasks often need 12-20 requirements.

- Use the same requirement ids across ‘minimum_requirements‘, ‘reference_requirements‘, and ‘better_requirements‘, so each basic/reference/better standard is easy to compare.

- Requirements should be specific and checkable. Use task-relevant concrete standards such as scope, coverage, behavior, correctness, validation, environment setup, edge cases, evidence quality, or actionability.

- Numbers or ranges are only one way to make a requirement concrete. Use them when they naturally fit the task, but do not force numeric requirements when behavior, correctness, or validation is the real standard.

- Example: if the reference clearly inspected about 30 relevant webpages/sources for a research task, a concrete requirement can say the reference-level workspace should cover about 30 relevant sources with useful categorization; a basic passing workspace might cover at least 15 relevant sources; a better-than-reference workspace might cover more sources or provide stronger validation, comparison, and risk analysis.

4. Inspect ‘bundle.docker.candidate_image_ref‘.

- Find what the candidate actually produced or changed.

- File paths are evidence only; they do not define success unless the task specifically requires a path.

5. Directly compare the candidate workspace against the reference workspace.

- Compare content and task quality, not whether the same file name exists.

- Ask whether the candidate is below minimum, near minimum, weaker than reference, close to reference, or better than reference.

6. Score the candidate using all three anchors: task requirements, the minimum requirements, and the direct reference comparison.

Docker inspection:

- Use Docker commands to inspect '/workspace' in both images.

- Read relevant files, not just filenames.

- Inspect code, scripts, configs, generated artifacts, documents, and environment setup when relevant.

- Run substantive validation commands when the workspace contains runnable code and the task requires running, fixing, validating, downloading, building, testing, or checking behavior.

- Do not rely only on a candidate's written report, final message, or claimed command output. Enter the Docker image and execute the relevant workflow yourself when it is possible: scripts, tests, builds, package commands, config loading, server startup, tool/client calls, data-processing pipelines, generated-artifact checks, or other task-specific verification.

- Validation is not limited to a smoke test. If the task needs a complex command or multi-step run to know whether the workspace works, run that command or workflow.

- If the full workflow requires unavailable external services, GPU/display hardware, credentials, network access, or would take an unreasonable amount of time, run the strongest feasible partial validation and explain the exact blocker.

- When you run a validation command, cite the observed result in the requirement 'reason': what command or behavior you checked, whether it passed, and where it failed if blocked.

- Do not modify either image. Use 'docker run –rm' for inspection.

Scoring logic:

- The reference image gives the solid baseline for this exact task.

- The minimum requirements define what just barely counts as basic completion.

- The better requirements define what would be meaningfully better than the reference baseline.

- Do not score only by checking the minimum requirements. Direct comparison with the reference workspace is a core part of this metric.

- The reference comparison should shape the requirements themselves. If the reference succeeds because it researches more sources, gives clearer categorization, validates behavior, or implements a more complete solution, write that as a reference requirement.

- Better requirements should be plausible improvements over the reference, not generic ideals. They should say what deeper, broader, more validated, more robust, or more useful work would look like for this exact task.

- Compare the candidate and reference on task-specific content quality, depth, breadth, correctness, implementation completeness, validation evidence, robustness, and usefulness.

- For research tasks, compare whether the candidate covers the same conceptual ground as the reference, and whether it is deeper, broader, better organized, more concrete, or more actionable. Do not require the same filenames.

- For code/config tasks, compare whether the candidate has the same functional behavior and validation quality as the reference, not whether it copied the same files line by line.

- For toolchain/code/environment tasks, runnable validation is part of the task quality. A script, config, install, or report is not enough by itself if the task requires something to run; the requirements should ask for inspected execution evidence.

- For analysis tasks, compare whether the candidate is as grounded, complete, nuanced, and useful as the reference.

- Candidate below the minimum requirements should usually score below 60.

- Candidate that only meets the minimum requirements should score around 60.

- Candidate clearly better than the minimum but weaker than the reference baseline should usually score 60-79.

- Candidate close to the reference baseline should score around 80.

- Candidate satisfying important better requirements can score above 80.

- Score above 90 only when the candidate has strong task-specific correctness, depth, validation, coverage, robustness, or usefulness beyond the reference baseline.

Anti-hack review:

Penalize heavily for:

- Only writing a worklog/report that claims success while required code/config/artifacts are missing.

- Empty stubs, placeholder files, toy examples, or fake pass outputs.

- Hardcoded outputs that avoid the task's real requirement.

- Removing or weakening checks/tests instead of fixing the underlying issue.

- Creating unrelated polished artifacts while the required task implementation is absent.

- Claiming a command/test/build passed without inspected workspace evidence.

- Overfitting to the worklog text instead of doing the task.

Criteria file:

Before returning the final result, write exactly this JSON object to ‘./output/criteria.json‘:

{
”task”: ”short task summary”,
”candidate_image”: ”candidate docker tag”,
”reference_image”: ”reference docker tag”,
”minimum_requirements”: [
{
”id”: ”req_1”,
”requirement”: ”detailed minimum requirement for basic completion”
}
],
”reference_requirements”: [
{
”id”: ”req_1”,
”requirement”: ”what the reference image shows for the same requirement”
}
],
”better_requirements”: [
{

”id”: ”req_1”,

”requirement”: ”what would be meaningfully better than the reference for the same requirement”

}

]

}

Requirement writing rules:

- Requirements must describe task/content quality, not exact file names or paths.

- Do not write requirements like ”the workspace contains /workspace/foo.md” or ”uses the same filename as reference”.

- File paths and filenames are evidence only. Put them in ‘candidate_files‘ or in the ‘reason‘ field when judging a requirement.

- Requirements should be concrete, but only in ways a different valid solution can satisfy.

- Write enough requirements to make scoring discriminative. For broad tasks, prefer many concrete requirements over a few vague ones; around 20 requirements is acceptable when the reference workspace has enough substance.

- Keep each requirement focused on one checkable quality point. Do not merge unrelated quality points into one vague requirement.

- Use matching ids across ‘minimum_requirements‘, ‘reference_requirements‘, and ‘better_requirements‘; for example, ‘req_7‘ should describe the same aspect at basic, reference, and better-than-reference levels.

- Good concrete requirements include task-specific scope, coverage, behavior, correctness, validation, dependency/config correctness, environment reproducibility, edge cases, and evidence quality.

- Numeric thresholds are useful only when the task naturally has a countable scope. They are examples of concreteness, not the main goal.

- If the reference suggests a useful count or range, use it. If the important standard is behavior, correctness, validation, or environment quality, write that standard concretely instead.

- Bad concrete requirements require identical filenames, identical wording, identical line count, or exact document layout unless the task explicitly requires that.

- For research tasks, requirements should describe quality such as depth, breadth, correct categorization, concrete examples, useful comparisons, limitations, and actionability.

- For code tasks, requirements should describe behavior, integration, correctness, validation, robustness, and maintainability.

- For environment/config tasks, requirements should describe dependency correctness, runnable commands, path consistency, configuration validity, and reproducibility.

- For toolchain, code, script, environment, or ”run/fix/validate” tasks, include explicit requirements for actual execution or the closest possible safe validation. Do not let a candidate score high only because it wrote plausible code, commands, or a report.

- Validation requirements should exist at all three levels:

- Minimum: the candidate must run the core command/script/tool or explain a real inspected blocker with evidence.

- Reference: the candidate should provide execution evidence comparable to the reference workspace, such as command output, installed artifacts, generated files, test results, logs, or inspected failure location.

- Better: the candidate should provide stronger end-to-end validation, preserved logs/scripts, generated artifacts, repeated checks, edge-case checks, or clear reproducibility.

- If a task says something was ”run through”, ”fixed”, ”verified”, ”connected”, ”validated”, ”started”, ”built”, ”downloaded”, or ”tested”, write a requirement that checks whether this actually happened in the workspace. Treat unverified claims as weak evidence.

- If the candidate includes runnable code, a toolchain, scripts, tests, generated artifacts, or environment setup, verify the important executable paths yourself. This may require complex commands or multi-step workflows. If you cannot run them, explain the concrete blocker and do not give the same credit as a verified run.

- For analysis tasks, requirements should describe coverage, grounding, nuance, alternatives, failure modes, and useful conclusions.

- ‘minimum_requirements‘ are the basic passing requirements.

- ‘reference_requirements‘ are stronger requirements inferred from the reference image.

- ‘better_requirements‘ describe what would be deeper, broader, more detailed, better validated, more robust, or more useful than the reference.

Concrete requirement examples:

- Research task where breadth matters:

- Minimum: cover the main relevant source categories and record why each category matters.

- Reference: cover the important source categories with concrete examples, correct categorization, useful comparisons, and limitations.

- Better: add stronger coverage, validation notes, collection feasibility, risk analysis, or more actionable prioritization than the reference.

- Research task where the reference clearly has a countable scope:

- Minimum: if the reference inspected about 30 relevant webpages/sources, a basic completion might inspect at least 15 relevant sources and record the main findings.

- Reference: cover about 30 relevant sources with categories, examples, and useful comparisons matching the inspected reference depth.

- Better: cover more than the reference or add materially better validation, source-quality checks, risk analysis, prioritization, or actionability.

- Research task where the reference distinguishes core, adjacent, and weak sources:

- Minimum: separate clearly relevant sources from adjacent or weak sources.

- Reference: explain why each category matters and give concrete caveats for adjacent sources.

- Better: add an actionable prioritization plan and criteria for accepting or rejecting future sources.

- Code task:

- Minimum: implement the core requested behavior in the relevant code path, avoid obvious stubs, and run at least one focused command/test/manual check that exercises the core behavior.

- Reference: integrate the behavior cleanly, update related config or tests, and preserve inspected validation evidence comparable to the reference.

- Better: handle important edge cases, add meaningful tests, preserve reproducible logs or commands, and keep the implementation maintainable.

- Environment/config task:

- Minimum: required dependencies/config files exist and the main command is actually tried, or a real blocker is inspected and documented with evidence.

- Reference: commands, paths, and dependencies are consistent and there is inspected evidence of a successful or partially successful run, including where execution stops if blocked.

- Better: provide reproducible setup, preserved logs/scripts, clear failure handling, validation for the expected runtime, and generated artifacts when the task requires them.

- Toolchain validation task:

- Minimum: verify the requested toolchain component by running the relevant command or client path far enough to prove the installed component is real, and state the exact blocker if full completion is impossible.

- Reference: match the reference's evidence level: package/version checks, startup or handshake evidence, inspected entry points, command output, downloaded/generated artifacts, and a conservative readiness judgment.

- Better: complete an end-to-end run if possible; otherwise preserve raw logs/scripts and produce the smallest meaningful generated output or failed call evidence that proves where the chain stops.

- Analysis task:

- Minimum: answer the core question with evidence from inspected materials.

- Reference: cover main alternatives, limitations, tradeoffs, and concrete examples.

- Better: add deeper failure-mode analysis, clearer decision criteria, and more actionable next steps.

Return exactly this JSON object:

{

”module”: ”workspace_task_completion”,

”score”: 0,

”task”: ”short task summary”,

”criteria_file”: ”output/criteria.json”,

”candidate_image”: ”candidate docker tag”,

”reference_image”: ”reference docker tag”,

”candidate_files”: [

{”path”: ”path inside /workspace”, ”summary”: ”what is actually present”}

],

”minimum_requirements”: [

{”id”: ”req_1”, ”requirement”: ”requirement from criteria.json”, ”result”: ”short judgment”, ”reason”: ”short evidence-based reason”}

],

”reference_requirements”: [

{”id”: ”req_1”, ”requirement”: ”requirement from criteria.json”, ”result”: ”short judgment”, ”reason”: ”short evidence-based reason”}

],

”better_requirements”: [

{”id”: ”req_1”, ”requirement”: ”requirement from criteria.json”, ”result”: ”short judgment”, ”reason”: ”short evidence-based reason”}

],

”score_rationale”: ”short concrete explanation comparing candidate against minimum requirements and the reference workspace”,

”confidence”: 0.0,

”summary”: ”short concrete judgment”

}

Preference Rubric Construction

You are the global preference extractor for Membench.

Objective:

Use only this participant's real message logs and the corresponding Markdown before/after changes in 'doc_change.json'. From this evidence, extract one globally reusable set of preference metrics. The same global metrics will later be used to evaluate every subtask for this participant.

You must complete two tasks:

1. Identify the participant's primary document: the path of the main, persistent worklog or work record.

2. Derive approximately 30 global preferences from all reference Markdown changes. Each preference must state a clear direction, not merely name a dimension.

Input files:

- './input/reference_subtasks.json'

- Lists all subtasks.

- Each subtask contains task information, root preference metrics, and 'before_markdown'/'after_markdown' from 'doc_change'.

- 'doc_change' contains Markdown document edits produced by the reference run. Consider only Markdown/document records, not code, scripts, webpages, packages, or asset files.

- './input/message_logs/<subtask_id>.json'

- Contains the real message log for each subtask.

- Use it to determine which Markdown worklogs, reports, specifications, prompts, or planning documents the reference run actually wrote, and to identify the primary document path.

Primary-document identification requires special care:

- Do not extract it using regexes, fixed keywords, or filename rules. Filenames, titles, paths, and write behavior are clues, but the final judgment must come from reading the actual message logs and document changes.

- Review all Markdown/document changes before choosing the primary document. Do not rely only on the first subtask or one filename.

- The primary document is the participant's long-term durable record. It typically carries updates across multiple subtasks rather than serving as the complete deliverable for one task.

- Names such as 'xxx worklog.md', 'work log.md', 'daily.md', 'weekly.md', 'record.md', and 'log.md' are clues, not rules. A primary document may have an unusual name, and an artifact may also contain 'log' in its name.

- Its content usually resembles an ongoing record: dates, times, tasks, TODOs, tentative checks, attempts, results, failures, next steps, short bullets, work-session order, participant names, task titles, and incrementally appended passages.

- It is usually updated by appending, inserting near related context, adding the current task record, or extending an existing heading or paragraph, rather than by creating a new self-contained report.

- It may cover many topics without fully developing each one. Its purpose is to support later continuation, not to become a polished one-off document.

- A document is more likely to be primary if it is repeatedly read and appended across subtasks, repeatedly mentioned as the final record, or explicitly called a worklog or record in the message logs.

- If 'after_markdown' resembles a personal worklog title, a daily record, a task list, chronological notes, or short work items but does not contain a path, use the message log to locate the actual path.

- If several documents look like worklogs, choose the one that most clearly serves as the participant's persistent primary record. Do not select a longer or more polished artifact merely because it is more complete for one task.

Distinguish supporting documents carefully:

- Other documents may be research reports, rules, prompts, plans, designs, explanations, checklists, experiment logs, generated reference documents, or doc-id-like Markdown files.

- They usually act as task-specific artifacts: they are more complete, formally titled, independently readable, or limited to one topic.

- These documents may be important, but they are not the primary document. When a reference change targets one of them, later evaluation should place less weight on matching its path and more on content, structure, language, scope, evidence, and preference consistency.

- If a reference run updates both the primary document and a supporting document, interpret their roles separately: the primary document usually records the final work state, while the supporting document provides fuller material or explanation.

- If a task updates only a supporting document, do not redefine that document as primary. The primary document is a global concept, not simply the file modified in the current subtask.

The primary document path is required:

- Always return ‘main_worklog_path‘, which downstream single-preference evaluation will use directly.

- It must be a workspace path inferred from the real message logs and document changes.

- If it cannot be determined, return ‘main_worklog_path: ””‘ and explain why in ‘summary‘. Never invent a path.

- ‘main_document.reason‘ must give one concise reason, such as repeated worklog appends, use as the final record across tasks, or explicit identification as a work record in the message logs.

- ‘main_document.changed_subtasks‘ must contain only subtasks that clearly updated the primary document, not every subtask that read it.

From a dimension to a preference:

- A dimension is only an observation axis, such as length, sentence length, table use, append versus rewrite, or preservation of failed attempts.

- A preference must reflect the aggregate pattern across all subtasks, not an arbitrary conclusion from one subtask.

- For each dimension, compare all reference Markdown changes. Determine the dominant direction, the exceptions, and whether the pattern is stable enough to be global.

- If most changes are short records with little explanation and lightweight bullets, while only a few are long reports, derive a preference for concise updates. Treat the long artifacts as contextual exceptions.

- If most short records avoid tables but a few formal comparison documents use them, state that the primary document and short records favor non-tabular presentation, while artifact documents may use tables when comparison requires them.

- If most changes append small updates near existing context while a few create complete standalone documents, state that the primary document favors local append/merge edits while artifacts may be written independently.

- If a dimension has no clear direction and the contexts do not explain the split, do not force a strong preference. Choose a more stable dimension.

- Every preference must state the favored direction. Examples:

- Dimension: word/character budget. Preference: favor short updates close to the reference record length; do not expand a brief worklog entry into a full report.

- Dimension: sentence length. Preference: favor short, note-like sentences over long explanatory paragraphs.

- Dimension: append versus rewrite. Preference: favor small additions or merges near the existing context over replacing a local update with a new complete document.

- Dimension: uncertainty. Preference: preserve states such as still investigating, pending verification, uncertain, or try first rather than presenting preliminary observations as final conclusions.

- The output must contain preferences, not dimension names alone.

Requirements for global metrics:

- Produce approximately 30 metrics; 27–33 is acceptable. Do not retain duplicate, overly narrow, or weakly supported preferences merely to reach a count.

- Every metric must be globally reusable. Do not mention a specific subtask, API, URL, project, command, date, or filename.

- Specific evidence may inform the trend, but the output preference itself must remain general and must not include long evidence passages.

- Every metric must have a direction: short versus long, append versus rewrite, tentative judgment versus formal conclusion, table versus non-tabular form, and so on.

- Keep metrics diverse. Do not restate the same idea in different words.

- If two metrics would normally receive the same score from the same evidence, merge them or remove one and replace it with a different dimension.

- Root preferences are references only. Do not simply paraphrase them; derive richer global preferences from all reference edits.

The review of global metrics is the most detailed and important part of this process:

- Work incrementally rather than generating the entire list at once.

- First create ‘./output/global_preferences_draft.json‘ and write an initial batch of 8–12 preferences. Review that batch, delete or revise invalid items, add a small number from different dimensions, review again, and continue until the list reaches approximately 30.

- Review every batch before adding the next. Do not draft everything first and perform only one superficial final review.

- Whenever a review removes a duplicate, non-global, or directionless preference, immediately replace it with a preference from a different dimension and review the replacement.

- Write the stable version to ‘./output/result.json‘ and return the same JSON object to stdout. ‘./output/global_preferences_draft.json‘ is temporary and need not remain. Do not include discarded drafts in the final result.

- Perform multiple rounds of drafting, review, revision, and completion until the list is stable.

- In every round, recheck global applicability, direction, overlap with root preferences, overlap with retained preferences, and whether message-log/document evidence can actually support evaluation.

- Return only preferences that survive all review rounds.

For every generated preference, check global applicability:

- Would it still be evaluable if the subtask, technical domain, and filename changed?

- If not, rewrite it more generally or remove it.

- Do not write ”preserve the Three.js demo link”; write ”favor retaining concrete, reviewable sources or examples without requiring exact names.”

- Do not write ”update a particular person’s worklog file”; write ”when the reference updates the primary document, favor writing the final record back to that document.”

For every generated preference, check direction:

- Do not output only ”word count” or ”sentence length.”

- State the favored side: shorter or longer, short or long sentences, append or rewrite, rough record or formal report, preserved uncertainty or final conclusion.

- If direction is uncertain, do not force a strong preference. Use a better-supported dimension.

For every generated preference, check overlap with root preferences:

- A root preference may inspire analysis, but a derived global preference cannot simply restate it.

- If a root preference already covers concision, do not add many variants of short, sparse, or unexpanded writing. Consider genuinely distinct dimensions such as line budget, heading count, append behavior, or scope boundaries, and remove them too if they rely on the same evidence.

- If a root preference already covers structure, do not accumulate redundant bullet, list, and section preferences.

- If it already covers informal work-record language, do not accumulate redundant preferences about light conversational tone, mixed technical language, or avoiding formal register.

- If it already covers concrete anchors, do not accumulate redundant preferences about retaining URLs, APIs, commands, and project names.

Compare every generated preference with previously retained ‘global_preferences‘:

- If two preferences mean nearly the same thing, remove the later one.

- If they would rely on the same passage, file behavior, or missing evidence, remove one.

- Do not retain several preferences that all penalize excessive length, excessive formality, failure to update the primary document, missing anchors, lost uncertainty, or needless structural complexity. Keep the clearest one and replace the rest with different dimensions.

- After every deletion, add a preference from a different dimension and review it again.

- Repeat deletion, revision, replacement, and review for multiple rounds.

Perform a final whole-list review:

- Avoid too many location-related metrics. A few may concern the primary document, but most of the list must not be about paths.

- Avoid filling the list with concise, short, sparse, or do-not-expand variants.

- Cover diverse dimensions: text form, editing behavior, the relationship between primary and supporting documents, record organization, expression granularity, evidence and verification, task scope, local language style, structural rhythm, and judgment or risk expression.

- Ensure every preference can be supported by message-log/document evidence visible to the single-preference evaluator.

- Keep the output concise; do not include full before/after text.

Few-shot dimension pool:

These are prompts for exploration, not fixed answers. Convert any selected dimension into a directional preference.

Text form:

- word/character budget; line budget; sentence length; paragraph length

- bullet count and nesting depth; heading count; table use

- blank-line and separator rhythm; balance of bullet lengths

- short work record versus long report

Information density:

- information per item; summary/detail ratio; number of examples

- amount of background explanation and repetition
- abstraction versus concrete record; scanability
- whether useful information is buried in low-value detail

Editing behavior:

- append versus rewrite; local patch versus whole-document restructuring
- preservation of prior context; restraint in deletion
- small traceable updates versus large untraceable rewrites
- insertion near related context versus an unrelated location
- durable record versus temporary analysis draft
- placement above or below existing text
- continuation of an existing topic rather than taking over the document opening

Primary/supporting-document relationship:

- use of the primary document as the durable record
- preservation of supporting documents in an auxiliary role
- clarity of the source of truth
- writing core conclusions back to the primary document
- avoiding use of an artifact as the only result
- different levels of detail in the primary and explanatory documents
- allowing fuller supporting documents without replacing the primary worklog

Record organization:

- chronological or work-session order; update granularity
- reuse of local labels and vocabulary; continuity with nearby context
- avoidance of unnecessary new classifications or taxonomies
- preservation of existing headings
- updating an existing section rather than creating excessive new sections

Expression granularity:

- observation versus conclusion; factual record versus recommendation
- process note versus final-outcome summary; raw note versus over-synthesis
- checklist versus narrative; label granularity
- omission discipline: do not expand everything the reference leaves implicit

Evidence and verification style:

- proximity of claims to evidence; source-attribution style; example-to-claim order
- visibility of failed attempts
- markers such as checked, not run, pending, or to verify
- explicit assumptions; preservation of negative evidence or rejected options
- preference for concrete records without requiring exact URLs, APIs, or commands

Task boundaries and scope:

- adherence to the requested small task; minimal sufficient deliverable
- avoidance of non-target work
- restraint in recording tool or process details
- avoidance of leaking excessive implementation detail when the deliverable is a record

- research depth proportional to the task
- avoidance of extra procedures that obscure the core preference

Local language style:

- modal language such as maybe, roughly, for now, still checking, try first, or later
- distinction among done, doing, and planned states
- imperative TODOs versus descriptive summaries; noun density
- lightweight work-record connectors versus formal-report connectors
- tolerance for abbreviations; reuse of local wording rather than full standardization
- preservation of mixed Chinese/English technical vocabulary when present

Structural rhythm:

- top-heavy versus bottom-heavy explanation; conclusion placement
- whether sections are forced into symmetry; heading specificity; hierarchy depth
- rough, scannable rhythm versus polishing into a specification

Judgment and risk expression:

- comparison style; ranking granularity; threshold or range expression
- trade-off wording; caveat placement; avoidance of unsupported certainty
- confidence gradients; visible but non-exaggerated risk
- preservation of preliminary findings as preliminary

Keep the output concise. Do not include long source passages, full before/after content, or input-file metadata.

Return exactly one JSON object and nothing else:

```
{
”module”: ”preference_global_derived_metrics”,
”person”: ”person id”,
”main_worklog_path”: ”workspace path to the primary document, or an empty string if it cannot be determined”,
”main_document”: {
”path”: ”path to the primary document”,
”reason”: ”one sentence explaining why this is the primary document”,
”changed_subtasks”: [”subtask_0001”]
},
”global_preferences”: [
{
”id”: ”global:d001”,
”dimension”: ”dimension name, for example word/character budget”,
”preference”: ”the participant’s explicit, evaluable preference direction”
}
],
”coverage”: {
”subtask_count”: 0,
”main_document_change_count”: 0,
”other_document_change_count”: 0
```

},
”confidence”: 0.0,
”summary”: ”one-sentence summary”
}

Preference Score

You are a strict evaluator for one Membench preference subtask.

Module rubric:

‘preference_metric_guarantee‘

Question:

Did the evaluated run preserve this person’s preference behavior for the current task?

Use evidence from the client input:

- ‘metric_under_test‘: the one metric to judge in this API call.

- ‘task_context‘: task description, root preference metrics, and ‘main_worklog_path‘.

- ‘worklog_ground_truth‘: reference before/after markdown changes for this subtask.

- ‘reference_document_writes‘: document writes extracted from the reference trace.

- ‘agent_document_writes‘: document writes extracted from the evaluated run trace.

Document-write evidence:

- ‘reference_document_writes‘ is extracted from the reference agent execution trajectory and keeps only detected document files that were written by the reference run.

- ‘agent_document_writes‘ is extracted from the evaluated agent execution trajectory and keeps only detected document files that were written by the evaluated run.

- Read-only commands, code files, structured data files, and unrelated tool events are omitted.

- If a document-write list is empty, treat that as no detected document write in that execution trajectory.

Core rules:

- Do not derive new per-subtask preference metrics.

- Do not select, merge, split, or rewrite metrics. Judge only ‘metric_under_test‘.

- Use ‘support: ”ignored”‘ narrowly. Only ignore ‘metric_under_test‘ when it is completely outside this subtask’s task type, reference markdown change, document role, writing behavior, and extracted write evidence, so no meaningful evidence could exist here.

- If ‘metric_under_test‘ is even somewhat related to the current reference change or evaluated run, do not ignore it. Score it as ‘none—weak—partial—strong‘ and explain the evidence or missing evidence.

- Use ‘main_worklog_path‘ to decide whether the reference change targets the person’s primary durable worklog or a supporting artifact/document.

Important metric isolation rule:

Each API call judges exactly one preference: ‘metric_under_test‘. Do not let any other preference, document-placement concern, root metric, global metric, or general worklog-quality concern affect the score unless it is explicitly part of ‘metric_under_test‘. In particular, do not punish a run for not updating the primary worklog when the current metric is about style, tone, wording, concision, roughness, uncertainty language, evidence style, scope control, or another behavior that can be judged independently of where the text was written. For those behavior/style metrics, first ask: ”Did the evaluated run show the behavior described by this metric in the relevant written output?” Judge that behavior directly. File path, main-worklog

placement, artifact-vs-worklog choice, and durable-record target should matter only when the current metric itself is about placement, target document choice, durable record location, or preserving the document-reference change. If the metric is about writing style, then a supporting artifact, a secondary document, or another written note can still provide valid evidence of ‘strong‘ support when it matches the reference style. Conversely, a run can fail a placement metric while still strongly satisfying a style metric. Keep these judgments separate.

Important applicability rule:

Before choosing ‘none‘, first decide whether the ground-truth/reference behavior actually gives evidence that ‘metric_under_test‘ applies to this subtask. The reference run is the anchor for applicability. If the current task and the ground-truth document change do not meaningfully exercise this preference, return ‘ignored‘, not ‘none‘. ‘none‘ is only appropriate when the metric is applicable because the reference/ground-truth behavior does show or require this preference, but the evaluated run clearly fails to show it. In other words: ‘ignored‘ means ”this metric is not meaningfully testable in this subtask”; ‘none‘ means ”this metric is meaningfully tested by the reference behavior, and the candidate completely misses it.” Do not mark a metric ‘none‘ merely because the evaluated run lacks some behavior that the reference task never called for. For global preferences especially, many person-level habits will not apply to every subtask. If the GT/reference change is a short status note, do not force metrics about long artifact structure unless that behavior is actually relevant. If the GT/reference change is a placement update, do not force unrelated style or evidence metrics unless the reference also makes those behaviors observable. Always check the GT/reference first for applicability, then judge the candidate against the applicable metric.

Modified document summary:

- ‘gt_modified‘ should list only the reference-changed markdown/document targets and a short summary.

- ‘candidate_modified‘ should list only document-like files the evaluated run actually created or updated, especially markdown/worklog/report/spec/prompt/planning records.

- Do not list files that were only read, listed, or mentioned.

- Do not list non-document implementation artifacts unless they are the main written record being evaluated.

- Do not output raw before/after markdown. Summarize.

- Do not output runner metadata paths.

Support labels:

- ‘ignored‘: the metric is completely unrelated to this subtask or impossible to judge from the available evidence.

- ‘none‘: the metric applies and the evaluated run clearly fails it.

- ‘weak‘: the metric applies but support is minimal, indirect, or mostly missing.

- ‘partial‘: the evaluated run satisfies part of the metric but misses important details.

- ‘strong‘: the evaluated run clearly satisfies the metric in the same preference behavior as the reference.

Judgment guidance:

- Judge the evaluated run against the specific behavior requested by ‘metric_under_test‘.

- Use the reference before/after change and extracted reference document writes to understand the intended content, scope, tone, and level of detail for this subtask.

- For document-reference or document-placement metrics:

- Decide whether the reference change targets the primary worklog or a supporting artifact/document.

- If the metric is about durable-record placement, missing the primary worklog update is strong evidence of failure.

- If the reference targets a supporting artifact/document, exact filename mismatch is usually less important than content, scope, tone, evidence, and preference behavior.

- For metrics not about document placement, do not over-focus on exact path, filename, main-worklog status, or whether the written output is in a supporting artifact. Judge only the preference behavior described by ‘metric_under_test‘.

- For writing-style or tone metrics, compare whether the evaluated run uses the same level of formality, roughness, concision, language mix, status wording, and note-taking rhythm as the reference behavior.

- For scope and structure metrics, judge whether the evaluated run keeps the same task boundary, avoids unrelated expansion, and organizes the output in a way that matches the person’s usual preference.

- For uncertainty or judgment metrics, check whether the evaluated run preserves tentative language, pending/blocked/failed states, comparative judgments, and limitations when the reference behavior calls for them.

- Content mismatch, missing central facts, wrong preference behavior, unsupported overreach, style mismatch, excessive verbosity, missing uncertainty, weak evidence, or poor scope control should lower the support label when relevant to the metric.

- Do not require byte-for-byte wording or exact URL/API/command names unless the metric is specifically about preserving exact concrete text. Prefer the same preference behavior and same recording style.

Metric application:

- ‘metric_under_test‘ may come from a human-selected root metric or a globally derived person-level preference. Apply it to this subtask when the current task gives meaningful evidence about that behavior.

- Relevance can come from document role, writing style, evidence-recording habits, status/uncertainty wording, scope control, concrete-detail preservation, output format, or the kind of artifact being produced.

- Use ‘ignored‘ only when the metric is unrelated to this subtask or the available evidence cannot meaningfully support or contradict it.

- If the metric is relevant but only weakly evidenced, choose ‘weak‘ or ‘partial‘ instead of hiding the uncertainty under ‘ignored‘.

- Apply the metric in the current context. For example, a concision preference should not blindly penalize a task whose reference output is intentionally a longer artifact; judge whether the length and detail are appropriate for this subtask.

Return exactly this JSON object:

{

”metric_id”: ”same id as metric_under_test.id”,

”support”: ”ignored—none—weak—partial—strong”,

”reason”: ”short reason with concrete evidence”

}

Preference Pairwise Comparison

Preference Recall Compare Judge

Use this section only when the evaluation bundle has ‘module:

preference_recall_compare‘.

You are comparing two evaluated runs for the same subtask:

- ‘candidate_a‘: one message log.
- ‘candidate_b‘: another message log.
Important:
- Candidate A and candidate B are randomly assigned by the runner.
- Do not assume candidate A is with-recall or without-recall.
- Do not infer labels from file names or directory names. The runner will map your raw candidate choice back to ‘with_recall‘ or ‘without_recall‘.
Question:
Which candidate better preserves the user’s preference/worklog behavior shown by the reference before/after change and selected root preference metrics?
This is not a general task-quality contest. A long, polished artifact is not automatically better. A candidate is better when it more closely follows the user’s durable-record and language-preference pattern for this subtask.
Evidence to use from the client input:
- ‘task_context‘, especially ‘task_description‘, ‘metric_description‘, ‘selected_preference_metrics‘, and ‘main_worklog_path‘.
- ‘global_preferences‘, especially when selected root preference metrics are empty.
- ‘worklog_ground_truth‘, especially summarized before/after markdown changes.
- ‘rewritten_message_log‘, only as supporting reference-trace evidence.
- ‘candidate_a.agent_trace‘.
- ‘candidate_b.agent_trace‘.
- Trace/message-log content may be head-tail truncated when ‘truncated‘ is true.
First decide whether the GT change updates the explicit ‘main_worklog_path‘.
If it does, writing the durable main worklog is an important preference signal.
If it does not, exact artifact filename is much less important and content/style alignment should dominate.
Comparison rubric:
1. Durable record / target behavior
- Prefer the candidate that updates the same durable record type as GT.
- If GT updates the explicit main worklog, prefer the candidate that writes that main worklog. A candidate that only writes a standalone artifact can still get content credit, but should usually lose this dimension.
- If both write or both miss the main worklog, compare the quality of the written record rather than double-counting path.
2. Worklog language style
- Prefer concise rough worklog notes when GT is concise.
- Prefer mixed Chinese-English technical note-taking, short bullets,

lightweight status wording, and local worklog rhythm when GT uses them.
- Penalize converting a short daily/worklog note into a formal report, architecture spec, or over-polished deliverable unless GT itself is formal.
3. Judgment/status/uncertainty style
- Prefer candidates that preserve tentative wording, TODO/pending/running status, blocked/failed checks, ”still checking” language, rough comparative judgments, and non-overconfident conclusions when GT uses them.
- Penalize unsupported completion claims, over-certainty, or hiding limitations in a polished artifact.
4. Proportional detail and scope control
- Prefer the candidate whose amount of detail matches the reference change.
- If GT is a short worklog continuation, a very long standalone document may be worse even if technically useful.
- If GT is an artifact/spec, a longer artifact may be fine, but added details should not invent requirements or obscure the core preference.
5. Root metric support
- Use selected root preference metrics as high-level constraints. Prefer the candidate that better supports those metrics in a way visible from the message log.
- If selected root metrics are empty, use global person-level preferences instead of blocking.
- Explain which selected root or global preference dimensions actually separate the two candidates.
Do not judge by:
- Which candidate has more tokens.
- Which candidate has a more formal file.
- Which candidate sounds more complete in final claims.
- Exact path/name matching for non-primary artifact documents.
- Exact URL/API/command equality when the concrete-recording style is otherwise preserved.
Suggested dimensions:
- ‘durable_record_target‘
- ‘worklog_concision_and_tone‘
- ‘status_uncertainty_and_judgment‘
- ‘proportional_scope‘
- ‘root_metric_alignment‘
Each dimension should choose one of:
‘candidate_a—candidate_b‘.
Decision labels:
- ‘candidate_a‘: candidate A is clearly better.
- ‘candidate_b‘: candidate B is clearly better.

- Always choose either ‘candidate_a‘ or ‘candidate_b‘. If both are weak, choose the less weak one.
- If the difference is small, still choose the slightly better candidate and use lower confidence.

Confidence:

- Use high confidence only when several dimensions point the same way.
- Use medium confidence when the winner is mostly from one major factor such as main-worklog target.
- Use low confidence when both candidates are similar, both fail the main preference, or message-log evidence is thin.

Return exactly this JSON object:

```
{
"module": "preference_recall_compare",
"winner": "candidate_a—candidate_b",
"confidence": 0.0,
"dimension_results": [
{"dimension": "durable_record_target", "winner": "candidate_a—candidate_b", "reason": "short concrete reason"},
{"dimension": "worklog_concision_and_tone", "winner": "candidate_a—candidate_b", "reason": "short concrete reason"},
{"dimension": "status_uncertainty_and_judgment", "winner": "candidate_a—candidate_b", "reason": "short concrete reason"},
{"dimension": "proportional_scope", "winner": "candidate_a—candidate_b", "reason": "short concrete reason"},
{"dimension": "root_metric_alignment", "winner": "candidate_a—candidate_b", "reason": "short concrete reason"}
],
"summary": "short concrete explanation of why the winner is better"
}
```

D.7.2 Reference Historical Evidence

For each target task, the following prompt is applied independently to every earlier task in the same participant workflow. Relevant pairs and excerpts extracted from the earlier trace form the reference historical evidence.

Pairwise Historical Relevance and Excerpt Extraction

You are determining whether two Membench subtasks have prerequisite-memory relevance.

Each request contains one pair:

- ‘previous_task‘: the earlier subtask.
- ‘target_task‘: the subtask being evaluated.
- Both tasks include a task card and the original execution message log.

Objective:

Determine whether ‘previous_task‘ provides useful prior memory for ‘target_task‘. Useful does not mean broadly similar. It means that remembering the earlier task would make the agent’s search path, implementation choice, validation method, task boundary, output document, risk judgment, or final deliverable more accurate.

Use a strict standard:

- Default to ‘not relevant‘. Mark the pair relevant only when the earlier task contains concrete information that would change execution or output for the target task.

- Relevant evidence must establish three links: the concrete fact in ‘previous_task‘, why ‘target_task‘ needs it, and how it changes current execution, validation, documentation, or conclusion boundaries.

- If the evidence says only that the tasks share a project, worklog, file, time period, or topic, or that historical content should be preserved, normally mark it not relevant.

- Knowing only the primary-worklog path, document title, or requirement not to overwrite existing content is generally global document state rather than strong task-specific prerequisite memory.

- Empty placeholders, path mappings, ordinary formatting continuity, simple date appends, and avoidance of overwriting old content are normally not relevant. The exception is when recovering, locating, or repairing that exact document state is the target task’s core objective and the previous task is unique or essential evidence.

- Consecutive updates to the same worklog are not automatically relevant. Relevance requires that the target task need a concrete conclusion, TODO, failure cause, experimental result, artifact, command, dataset, schema, code path, or judgment boundary from the earlier task.

- Broad conceptual analogy is insufficient. Statements such as both involve agents, environments, terminals, benchmarks, or failure analysis must identify the exact earlier fact that reduces wasted work or improves the current deliverable.

Relevant cases include:

- The target clearly continues the same pipeline, dataset, repository, experiment, document, prompt, evaluation, cleaning process, deployment process, or analysis thread.

- The previous task created or modified an artifact, schema, script, configuration, command, path, field, sample, conclusion, failure cause, TODO, or validation method needed by the target.

- A mainline worklog and a non-mainline artifact, report, specification, or prompt may be relevant to each other when the connection changes how the target task should be performed.

Not-relevant cases include:

- The tasks merely concern the same participant or broad project, occur near each other, or both involve documentation.

- They merely use the same primary worklog, file path, title, date format, or indentation style.

- The previous task wrote old content that a later edit to the same file must preserve.

- The previous task only created an empty file, placeholder, directory, title, or path mapping.

- The connection is only generic background, broad style, personality, formatting cleanup, or a content-free placeholder.

- The tasks share a broad topic but no concrete artifact, decision, data, validation, task boundary, or reusable conclusion.

- Remembering the previous task would not make the target faster, more accurate, less wasteful, or closer to the required deliverable.

Procedure:

1. Read both task cards to understand each objective and deliverable.

2. Read both message logs to determine what the agent actually inspected, ran, changed, verified, and summarized.

3. Ask whether remembering ‘previous_task‘ would materially change the approach, eliminate irrelevant search, enable reuse of an existing artifact, avoid repeated validation, preserve an important boundary, or produce a more accurate conclusion.

4. If the answer cannot be stated beyond shared file, path, topic, historical preservation, or formatting continuity, mark the pair not relevant.

5. If relevant, explicitly reason through: concrete fact from the previous task -> concrete need in the target task -> effect on execution or output.

6. Write one concise ‘evidence‘ statement describing which fact from ‘previous_task‘ affects ‘target_task‘.

7. In ‘memory_needed‘, state the essential information that must be remembered. Keep it short but understandable; do not repeat background.

8. If relevant, extract supporting excerpts from the original ‘previous_task.message_log‘.

9. If not relevant, set ‘evidence‘ and ‘memory_needed‘ to empty strings and ‘excerpts‘ to an empty array.

Excerpt requirements:

- Extract excerpts only from ‘previous_task.message_log‘, never from ‘target_task‘.

- Preserve the source wording as closely as possible. Do not rewrite it as your own summary or invent facts absent from the previous task.

- Each excerpt should be a compact, evidence-dense passage, usually about 100–300 Chinese characters or a comparable amount of English text. It may combine consecutive or adjacent sentences from the same local context, but must not copy a large portion of the log.

- Prefer concrete evidence: filenames, paths, artifact names, run IDs, checkpoints, model names, dataset names, API endpoints, commands, scripts, metric values, sample counts, distributions, error messages, blockers, failure causes, user preferences, output formats, prohibited actions, completion state, and next steps.

- Do not extract vague statements such as ”the user is working on a project,” ”this concerns a benchmark,” or ”further analysis is needed.”

- A relevant pair normally contains 3–5 excerpts.

- Each excerpt’s ‘reason‘ must explain only why that source passage is useful to ‘target_task‘.

Return strict JSON only:

{

”relevant”: true,

”evidence”: ”concise judgment, or empty when not relevant”,

”memory_needed”: ”essential information to remember, or empty when not relevant”,

”excerpts”: [

{

”text”: ”an evidence-dense excerpt copied from previous_task.message_log”,

”reason”: ”why this excerpt is useful to target_task”

}

]

}

Rules:

- ‘relevant‘ must be ‘true‘ or ‘false‘.

- Do not output Markdown or extra fields.

- Keep ‘evidence‘ and ‘memory_needed‘ empty when they are unnecessary and concise when present.

- ‘excerpts‘ must be an array and must be ‘[]‘ when not relevant.

- Each excerpt object may contain only ‘text‘ and ‘reason‘.

- Do not mark a pair relevant merely because the tasks are close in time.

- Do not be overly conservative merely because the dependency is indirect, but require a concrete fact that changes target-task execution or output.

Runtime user message:

Determine whether ‘previous_task‘ provides useful prior memory for ‘target_task‘. Input:

{

”person”: ”<person_id>”,

”previous_task”: {

”task_card”: <previous_task_card>,

”message_log”: <previous_agent_trace>

},

”target_task”: {

”task_card”: <target_task_card>,

”message_log”: <target_agent_trace>

}

}

D.7.3 Memory Diagnostic Prompts

Relevance. The embedding-based mean similarity, recall, and precision are computed directly and therefore do not use an LLM prompt. The following prompt produces the complementary 1–5 relevance score.

Memory Relevance Score

You are a strict evaluator for the memory_relevance dimension of one Membench memory ablation subtask.

Category: context.

Question: Were the recalled memories actually relevant prior context for the current task?

This dimension evaluates the recalled context visible in ‘with_recall.message_log‘, not whether the agent used it well. When present, recalled memory is inside ‘<memory_recall_context>...</memory_recall_context>‘ in an early user message. Use the surrounding task in the same message to judge relevance. Use ‘without_recall.message_log‘ only as task and trajectory context, not as recalled memory.

What to judge:

- Identify the current task intent and required background.

- Identify recalled prior tasks, artifacts, decisions, worklogs, conventions, or user constraints from the visible recalled memory block.

- Decide which recalled memories are relevant, partially relevant, unrelated, stale, or potentially misleading.

- Prefer task-specific historical continuity over generic workspace history. A prior task that merely says ”inspect the workspace” is weak unless it names artifacts or decisions needed now.

- If recall includes many irrelevant memories but also one clearly important item, score based on both signal and noise.

Do not judge:

- Whether with_recall executed efficiently.

- Whether without_recall recovered the same context.

- General task quality.

Score on a 1-5 scale:

1. Recall is absent, almost entirely unrelated, or actively misleading for the current task.

2. Recall has weak/generic relevance; useful anchors are sparse or buried in noise.

3. Recall contains some relevant prior context, but important anchors are missing, vague, or mixed with substantial noise.

4. Recall contains clearly relevant prior artifacts/decisions/worklog context with manageable noise.

5. Recall is highly relevant and specific, naming the key prior tasks/artifacts/decisions needed for this subtask.

Return exactly this JSON object:

{

”dimension”: ”memory_relevance”,

”category”: ”context”,

”score”: ”integer 1-5”,

”relevant_prior_tasks”: [

{”id”: ”subtask_xxxx or history id”, ”relevance”: ”low—medium—high”, ”reason”: ”brief reason”, ”anchors”: [”artifact/decision/convention”]}

],

”irrelevant_or_noisy_memories”: [”brief item”],

”missing_expected_context”: [”brief item”],

”evidence”: [”brief concrete evidence from Evaluation input”],

”confidence”: 0.0,

”summary”: ”short judgment”

}

Environment Familiarity Score

You are a strict evaluator for the environment_familiarity dimension of one Membench memory ablation subtask.

Category: ability.

Question: Did the with_recall run behave more familiar with the local workspace/project environment than the without_recall run?

This is the renamed and refined routing_gain dimension. It measures local-environment familiarity shown through navigation, search, and target selection. It is not a pure recall-relevance score.

Important neutrality rule:

- Checking ‘/workspace/context‘, listing the root workspace, or broad discovery at the start is often required by task instructions. Do not count those checks alone as evidence of familiarity or unfamiliarity.

- Count only differences beyond shared setup behavior: specific historical file names, targeted directories, known artifact names, project terms, or reduced irrelevant detours.

What to judge:

- Whether with_recall uses more specific workspace/project queries earlier.

- Whether it reaches relevant files, folders, prior artifacts, or project state with fewer irrelevant detours.

- Whether without_recall has to reconstruct context through broad scans, wrong files, or avoidable command/path mistakes.

- Whether with_recall knows where outputs should go in the local environment.

- Ignore failures clearly due to network outage, unavailable external services, or shared tool instability rather than agent choice.

- Use tool calls, assistant messages, and file/content references in ‘with_recall.message_log‘ and ‘without_recall.message_log‘.

Score on a 1-5 scale:

1. With_recall is less familiar: follows wrong local context, misses obvious files, or has more avoidable local detours than without_recall.

2. Little environment familiarity advantage; both mostly rely on broad scans or with_recall is only slightly more targeted.

3. Some familiarity: with_recall uses a few relevant local anchors earlier, but without_recall recovers them with similar effort.

4. Clear familiarity: with_recall navigates to relevant project files/artifacts more directly and avoids notable detours.

5. Strong familiarity: with_recall immediately or near-immediately uses precise local anchors/conventions and without_recall shows clear reconstruction cost.

Return exactly this JSON object:

{

”dimension”: ”environment_familiarity”,

”category”: ”ability”,

”score”: ”integer 1-5”,

”with_recall_behavior”: ”brief concrete trajectory summary”,

”without_recall_behavior”: ”brief concrete trajectory summary”,

”familiarity_signals”: [”specific signal”],

”neutral_setup_ignored”: [”generic setup behavior ignored”],

”avoidable_detours_or_failures”: [”detour/failure and variant”],

”evidence”: [”brief concrete evidence from Evaluation input”],

”confidence”: 0.0,

”summary”: ”short judgment”

}

Tool-call Purpose Classification

You label the purpose of every tool call in an agent execution trace.

You receive the task instruction, the complete execution trace after limited tool-output truncation, and the exact call IDs that must be labeled. Classify each tool call using its purpose in context, not merely its tool name.

Use exactly one of these labels:

- ‘exploration‘: The call is used to understand or recover the existing environment, workspace, data, dependencies, or prior state before deciding how to act. Examples include listing directories, searching for files or symbols, reading existing files, inspecting configuration, checking installed tools, and probing the current state. A targeted read can still be exploration when its purpose is to locate or understand the problem.

- ‘execution‘: The call directly advances the requested task. This includes editing or creating deliverables, installing a required dependency, running the target computation, generating data or artifacts, applying a fix, and testing or validating the implementation or output. Failed calls remain ‘execution‘ when their intended purpose was to perform or validate the task.

- ‘other‘: The call neither explores the environment nor executes the task itself. Typical examples are plan/status bookkeeping, user communication, or unrelated administrative actions.

Important distinctions:

- Do not label by command name alone. Reading a pre-existing file to understand the workspace is usually ‘exploration‘; reading a newly generated output to verify the task is usually ‘execution‘.

- Broad discovery and targeted diagnosis are both ‘exploration‘ when they primarily gather information needed to choose the next action.

- Tests, builds, smoke checks, and result inspection are ‘execution‘ when they validate work performed for the current task.

- A tool call that attempts the task but fails is still ‘execution‘; the outcome does not change its purpose.

- Label every occurrence in the provided call-ID list exactly once and preserve its order. A call ID may appear more than once when the source trace reuses an ID; in that case, return the same ID once for each occurrence. Do not invent, omit, merge, or reorder occurrences.

Return only a JSON array in the same order as the provided call IDs:

[

{

”call_id”: ”exact call id”,

”label”: ”exploration”

}

]

Each object must contain exactly ‘call_id‘ and ‘label‘. Do not include explanations or Markdown.

Execution Problem Identification

You are evaluating an agent execution trace. Identify only concrete execution problems that the agent actually encountered while carrying out the task, and determine whether each problem was resolved later in the same trace.

Count problems such as:

- a command, tool call, test, build, or program that failed unexpectedly;

- a missing local file, path, dependency, or executable that blocked an intended operation;

- an invalid local configuration, schema, data format, or implementation exposed during execution.

Do not count:

- weaknesses in the final answer, incomplete content coverage, stylistic issues, uncertainty, risks, plans, or suggestions;

- normal exploration, an expected negative check, or a search that legitimately returns no matches;

- a possible problem that was only discussed but never observed;

- failures caused by an unavailable external service, network, rate limit, authentication, credentials, permissions outside the workspace, or another resource the agent cannot repair.

Merge repeated failures caused by the same underlying issue. Keep separate problems with different causes.

Set "resolved" to true only when later execution evidence shows that the problem was fixed, or that a lossless alternative completed the same intended operation. The agent merely saying that it fixed the issue is not sufficient. Skipping the operation, accepting reduced functionality, leaving a TODO, or using a workaround that loses part of the requested result is unresolved.

Return only a JSON array. Each item must contain exactly:

{"problem": "a concise description of the observed execution problem", "resolved": true}

Use false instead of true when unresolved. Return [] when no countable execution problem occurred.

Problem Solvability from Recall

You are given candidate memory, a without-memory execution trace, and a fixed list of execution problems that remained unresolved in that trace.

For each problem, decide whether the candidate memory contains enough concrete information to solve it.

Set memory_can_solve to true only when the candidate memory provides a specific fact, correct path, constraint, existing artifact, verified command, configuration, or known procedure that addresses the actual cause of the problem. If an agent had this information and used it correctly, it should be able to resolve the problem without losing part of the requested result.

Set memory_can_solve to false when the memory is only topically related, provides general background, repeats the task goal, or lacks the concrete information needed to resolve the problem. Do not assume that information absent from the supplied memory was available elsewhere.

Do not identify new problems, rewrite the supplied problems, or reassess whether they were resolved.

Return only a JSON array in the same order as the fixed problems. Each item must contain exactly:

{"problem_index": 0, "memory_can_solve": false}

Include every supplied problem index exactly once. Return [] when the fixed problem list is empty.

Hallucination Robustness Score

You are a strict evaluator for the hallucination_risk dimension of one Membench memory ablation subtask.

Category: context.

Question: Did recalled memory cause or risk hallucinated, stale, irrelevant, or overfit behavior?

This replaces misuse_risk with a clearer direction: higher score means lower hallucination/misuse risk and better grounding.

Judge whether with_recall:

- Grounds historical claims in actual recalled or workspace-visible evidence.

- Avoids inventing prior decisions, files, constraints, or user preferences.
- Avoids following irrelevant recalled memory into the wrong artifact/task/person.
- Correctly treats weak or irrelevant recall as weak instead of overusing it.
- Does not ignore current-task instructions because of old memory.
- Judge only memory-related risk visible in ‘with_recall.message_log‘. If no recalled memory is visible, do not invent a memory misuse case.

Hallucination/misuse examples:

- Claims an artifact or convention exists without evidence.
- Updates a file because of stale/irrelevant memory rather than current task needs.
- Imports another person’s or another project’s context.
- Overfits to a previous task and solves the wrong problem.
- Treats a generic old workspace scan as a task-specific constraint.

Score on a 1-5 scale, where higher is safer:

1. Severe memory-induced hallucination or misuse; with_recall is materially harmed.
2. Clear risk or some harmful overfitting to irrelevant/stale memory.
3. Some uncertainty/noise; with_recall mostly avoids harm but makes weakly grounded historical assumptions.
4. Low risk; with_recall is mostly grounded and ignores irrelevant recall appropriately.
5. Very low risk; with_recall carefully grounds memory use, verifies important claims, and avoids overuse of weak recall.

Return exactly this JSON object:

{

”dimension”: ”hallucination_risk”,

”category”: ”context”,

”score”: ”integer 1-5”,

”risk_level”: ”low—medium—high”,

”grounded_memory_use”: [”specific grounded behavior”],

”hallucination_or_misuse_cases”: [”specific case if any”],

”irrelevant_recall_handling”: ”brief judgment”,

”evidence”: [”brief concrete evidence from Evaluation input”],

”confidence”: 0.0,

”summary”: ”short judgment”

}

Memory-induced Problem Classification

You are given a task instruction, the memory recalled before execution, an execution trace, and a fixed list of execution problems observed in that trace.

For each fixed problem, decide whether the recalled memory caused or materially contributed to the problem.

Mark memory_induced as true only when all of the following are supported by the inputs:

1. The recalled memory contains incorrect, outdated, misleading, or ambiguous information relevant to the problem.

2. The execution trace shows that the agent actually relied on or adopted that information.

3. This reliance caused or materially contributed to the observed problem.

Mark it as false when the memory merely contains questionable information that the agent did not use, when the connection is only speculative, or when the problem is a general coding, command, dependency, reasoning, or execution error not caused by recalled memory.

Do not identify new problems and do not change the supplied problem descriptions or resolution labels.

Return only a JSON array in the same order as the fixed problems. Each item must contain exactly:

{"problem_index": 0, "memory_induced": false}

Include every supplied problem index exactly once. Return [] when the fixed problem list is empty.