CONTINUAL GRAPH MEMORY AGENT FOR AI RESEARCH REPLICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Replicating modern AI research is often a multi-hour, error-prone process that hinges on locating implementation-critical details dispersed across text, code, and execution logs. Recent methods on AI research replication expose a recurring bottleneck: agents repeatedly re-discover the same reusable implementation patterns across papers. To address this challenge, we present the Congram — continual graph memory agent for paper replication that accumulates reusable "recipes" from papers and generated code to allow the continual experience collection and reuse for future code generation. Congram also uses the iterative code refinement based on the pruned code requirements tree review feedback. Evaluated on papers from PaperBench, Congram demonstrates significant improvements in replication scores for code generation compared to multiple memory-augmented and memoryless baselines. On Qwen3-Next and DeepSeek-R1 backbones, Congram improves the replication scores by 35.6 and 24.6 percentage points compared to the one-pass code generation that uses the original paper implementations.

1 INTRODUCTION

Automatically reproducing artificial intelligence research papers is a stress test for LLM agents. A successful reproduction requires understanding a paper, implementing its method, preparing data and evaluation code, and running experiments in a complete repository. Recently presented benchmark works (Siegel et al., 2024; Huang et al., 2024; Chan et al., 2025) study this setting as a long-horizon ML engineering task, using agents that reproduce papers from existing repositories or solve end-to-end ML tasks. PaperBench (Starace et al., 2025) makes the task challenging by asking agents to replicate recent ML papers from scratch, without access to the authors' original code.

Despite progress in code generation, current agents remain inefficient in this setting. Each paper is typically treated as an independent task: the agent reads the paper, plans a repository, writes code, runs experiments, and debugs failures from scratch. However, many reproduction attempts share a reusable structure. Across papers, agents repeatedly implement similar methods, dataset loaders, model wrappers, training loops, evaluation scripts, logging utilities, ablation pipelines, and repository layouts. Re-generating these components for every new paper wastes tokens and compute, and prevents the agent from benefiting from previous reproduction experience.

We propose a continual knowledge graph memory for paper-to-code reproduction. Rather than treating each paper as an isolated first-time implementation, our agent accumulates experience from previously reproduced papers and uses the LLM reviewer for iterative refinement of generated submissions. After each attempt, it stores reusable knowledge in a graph connecting methodological patterns, repository building blocks, and code snippets. For a new paper, we convert the paper text into a graph of the required method and experiments, query the accumulated memory, and retrieve relevant prior implementations that can be adapted instead of being generated from scratch.

We use papers and rubrics from PaperBench(Starace et al., 2025) in a continual learning setting, where papers are implemented sequentially, memory is updated after each attempt, and performance on later papers measures transfer from prior reproduction experience and refinement steps. We find that the proposed continual knowledge graph memory improves reproduction quality over standard and iterative paper-to-code baselines. It also reduces the number of generated code tokens, being tested in a single-pass (without iterative refinement) setting. These results show that reusable im-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

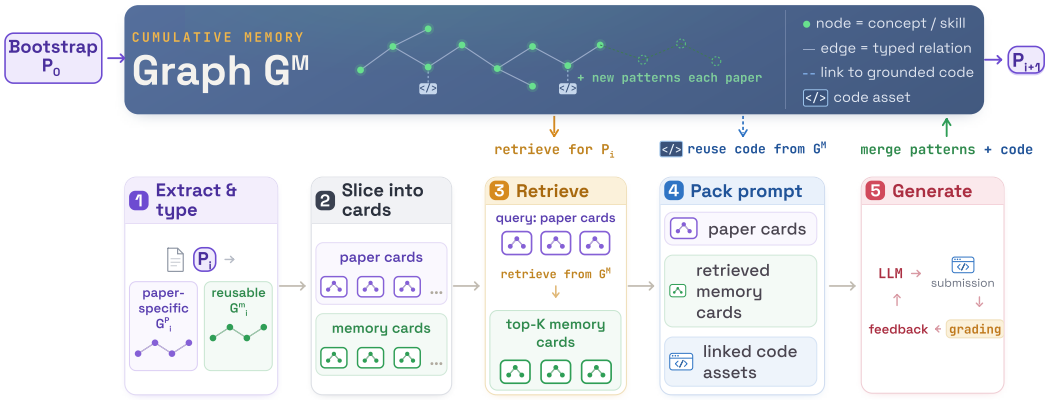

Figure 1: Congram code generation pipeline. The cumulative memory is initialized with the bootstrap paper P_0 knowledge: paper-specific triplets G_0^P and reusable patterns triplets G_0^m are extracted from the paper, and the original authors' code is linked to G_0^m. For each subsequent paper $P_i, i \geq 1$, the paper-specific G_i^P and reusable patterns G_i^m graphs are extracted, ontology-typed, and sliced into cards. Next, the paper cards are used as queries to retrieve relevant cards and code from the cumulative memory graph G^M. The paper cards and retrieval results are further packed into the prompt for code generation. The generated candidate submission is refined: LLM reviewer grades the submission, returns the feedback, and the feedback is propagated to the code generation prompt for the next iteration of refinement. The procedure is repeated until all the reviewer comments are satisfied or 5 iterations are exhausted. The best-scored candidate is marked as the target submission. To update the cumulative memory, the target submission is parsed and linked to paper memory graph G_i^m, which is further used to enrich cumulative G^M.

plementation experience stored in knowledge graph-based memory can make code generation more efficient and scalable.

Our main contributions are:

- Congram uniquely applies the continual accumulation and update of paper replication experience to the knowledge graph-based storage, combining textual information from papers and related code from agent-generated submissions. It also employs the LLM review-based code refinement loop to ensure the high quality of generated and stored code.
- In practice, Congram outperforms the memory-less single-pass and iterative baselines, and also outperforms the non-iterative baseline that uses the original implementations of the evaluated papers on Qwen3-Next and DeepSeek-R1 backbones.

2 Related Work

Paper replication benchmarks such as PaperBench (Starace et al., 2025) evaluate agents on producing submissions that satisfy expert-annotated requirements, highlighting the gap between paper descriptions and executable artifacts. MLAgentBench (Huang et al., 2024) and MLE-Bench (Chan et al., 2025) cast LLM agents as ML engineers on long-horizon research tasks. CORE-Bench (Siegel et al., 2024) evaluates reproduction from an existing repository rather than from scratch. The Agent-Native Research Artifact proposal (Liu et al., 2026) argues that replication should be represented as an executable knowledge package, including structured intermediate artifacts, and PaperRepro (Li et al., 2026) proposes to recover the tacit knowledge that underlies reproducible code.

Our pipeline conditions code generation on retrieved context, in the line of retrieval-augmented generation (Lewis et al., 2020) and its graph-structured extensions: GraphRAG (Han et al., 2025) shows that graph-indexed corpora improve multi-hop QA over flat retrieval. Closest to our setting, xKG (Luo et al., 2026) constructs per-paper executable knowledge graphs and plugs them into PaperBench scaffolds. We differ in two respects: we maintain a single *continual* memory graph that is carried across papers, and is updated come only from the agent's own generated code.

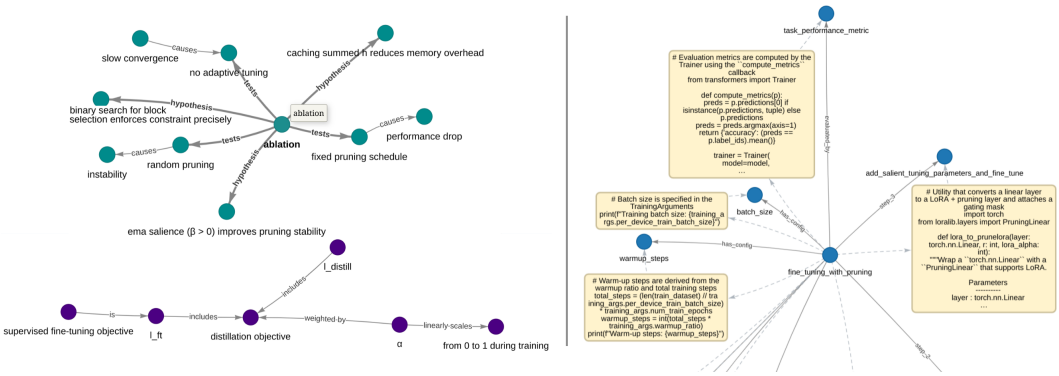

Figure 2: The example paper and memory graphs for `adaptive-pruning` paper from PaperBench. The paper is devoted to the APT mechanism that adaptively prunes and tunes parameters for the language models. Left: two connectivity components of the paper graph, covering the ablation study and supervised fine-tuning objective information. Right: a subgraph of the memory graph with linked node-relevant implementation code.

Long-term memory for LLM agents has been studied as both unstructured stores (Packer et al., 2024; Park et al., 2023) and structured graph-shaped memories. HippoRAG (Gutiérrez et al., 2025) ports hippocampal indexing to LLM memory, AriGraph (Anokhin et al., 2025) maintains a typed knowledge graph as an interactive agent's world model, and A-MEM (Xu et al., 2025) explores episodic-plus-semantic memory for tool-using agents. Our memory follows this graph-structured line and specializes it to repository-scale code generation: each replication pattern is linked to a code snippet via a symbol-index prompt.

Continual learning for coding agents formalizes task streams in which later tasks can benefit from earlier ones (Joshi et al., 2025). A separate literature studies the dynamics of learning from model-generated data, including bootstrapped reasoning and recent analyses of failure modes when agents recursively ingest their own outputs (Zverev et al., 2025). Iterative coding loops with self-feedback (Yao et al., 2022; Shinn et al., 2023) attack a complementary failure mode – noisy single-shot generation rather than missing implementation knowledge. We focus on a continual setting where the agent maintains and updates a structured memory across papers.

3 METHOD

Congram processes a sequence of paper manuscripts $P_0, P_1, \ldots, P_N$, maintaining one cumulative memory graph G^M that accumulates reusable patterns across the sequence (Figure 1). G^M is initialized at bootstrap with G_0^m – reusable implementation patterns extracted from P_0's text and linked to code from P_0's original authors' implementation. For each subsequent paper P_i, the system extracts paper-specific details into G_i^P and reusable patterns into G_i^m from the paper text, types both graphs against a hand-crafted ontology and slices them into cards (bundles of triplets sharing an anchor node that describe one localized sub-method such as a training loop or an evaluation block), retrieves the memory cards most relevant to the paper cards, packs them with their linked code assets into a single generation prompt to produce a candidate submission finalized via the iterative refinement loop, and finally links the generated code back to G_i^m's triplets and merges G_i^m into G^M. The full ontology is presented in Appendix C. In the remainder of this section, we describe each stage of the Congram pipeline in detail.

Triplet extraction Paper text is split into sections and chunks. Each chunk is sent to the LLM twice, once with a paper-graph prompt and once with a memory-graph prompt, so the two graphs differ by extraction objective rather than by post-hoc filtering. The example of two connectivity components of the paper graph for the `adaptive-pruning` (Zhao et al., 2024) paper from PaperBench is presented in Figure 2, left. The components present the structured information about the ablation study and the fine-tuning objective.

Failed requirement

```
The submission fails to permanently
    merge APT adapter parameters into
    the base weights before inference.
     Although it dynamically computes
    merged weights during forward
    passes, it retains and recomputes
    W_A and W_B every time, without
    removing them or permanently
    updating the model weights. This
    negates the core inference
    efficiency benefit of APT, as
    model size and computational cost
    remain unchanged.
```

After repair

```
The submission correctly implements
    the merge_adapters_weights()
    method that fuses APT adapter
    weights (W_A and W_B) into the
    base model weights using W_final =
     W_original + 2 * W_B * W_A, as
    specified in the paper. The method
     is called before inference and
    model saving, ensuring adapter
    parameters are eliminated during
    deployment. The implementation
    aligns with the paper's
    requirement for inference
    efficiency through parameter
    merging, and pruning masks are
    appropriately handled implicitly
    through trained weights.
```

Figure 3: The example of refinement loop feedback. Left: the reviewer report on the unmet requirement before repair. Right: the requirement is positively assessed after the refinement procedure.

Submission linking to memory triplets When a body of source code is available (the authors' code at bootstrap or the agent's own generated code during the continual update), we ground memory triplets in that code. A symbol index listing the classes, functions, and methods of every file with their source locations is built, and for each episodic chunk that produced new memory triplets, the LLM is asked to point each triplet at the smallest symbol that implements it with a confidence score; the reported location is expanded to its smallest enclosing block. The example of the subgraph of the memory graph with linked code for the `adaptive-pruning` (Zhao et al., 2024) paper from PaperBench is presented in Figure 2, right. The presented subgraph covers the fine-tuning procedure stages (like step 3 of adding salient tuning parameters), fine-tuning configuration information (the batch size logging and warmup steps setting), and evaluation metrics computation information.

Graph typing Before retrieval, raw triplet lists are materialized into typed graphs: entities and relations normalized, each entity tagged with an ontology type from its relational context. Then graphs are sliced into cards by anchor node. A paper card carries an intent label (configuration, procedure, evaluation, or implementation), and a priority proportional to its triplet count. Memory cards inherit the code assets linked to their triplets.

Memory retrieval and code generation To perform the retrieval from cumulative memory, we embed paper and memory cards using Contriever (Izacard et al., 2022) and calculate the cosine similarity score. The generation prompt is then assembled from paper cards, retrieved memory cards and their linked code assets; blocks are filled greedily, concatenated with an instruction, and sent in a single chat completion. The response is parsed into a complete submission directory with all source files and a reproduction script as required by PaperBench.

Submission iterative refinement During the code generation stage, we apply the review-and-repair loop to refine the generated submissions. The candidate submission is graded by an LLM reviewer with the pruned *code development* part of the PaperBench requirement tree. The pruning collapses the tree at depth $d = 3$. Every graded node is weighted by its share of the tree – the product of sibling-normalized weights along its path to the root. The unmet requirements (the ones having the score below 0.5) are ranked by that share times their score shortfall. The repair prompt is comprised of paper cards, retrieved memory cards, and code assets from the initial generation, the top unmet requirements (we use at most 8 unmet requirements per iteration) with the reviewer's explanation, and the satisfied requirements. Only the changed files' content is returned to merge the repair into the working tree. The refinement cycle repeats until no requirement is left unmet or a five-iteration budget is exhausted. The best-scoring submission is kept. The examples of LLM reviewer feedback before and after refinement are presented in Figure 3.

Memory update After the generation stage, G^M is updated from the agent's own submission in two steps. First, generated code is chunked and run through memory-code linking; only triplets not

already in G^M are admitted to avoid triplet duplicates. To create the bootstrap memory graph, we run the same stages but link against the original authors' repository for P_0, and it is the only point at which authors' code influences G^M. Prompts for triplet extraction, code generation, refinement, and linking are listed in Appendix D.

4 Experimental Results

For experiments, we use PaperBench benchmark (Starace et al., 2025), all 23 papers available in PaperBench to maximize the length of the continual code generation sequence (20 papers from the main benchmark set and 3 papers used by authors for human versus agent comparison). Each paper has a benchmark harness that returns a scalar score in $[0, 1]$ for a submission.

As baselines, we use (i) two paper-text-only generation baselines (following the benchmark paper): BasicAgent generates code directly from paper text, and IterativeAgent applies a self-refine 5-iteration loop using compile checks; and (ii) NoMem baseline that follows Congram per-paper graphs extraction, slicing, retrieval, and submission refinement stages but does not use cumulative memory or any code assets. It generates code for each paper independently, based on its paper cards and retrieved memory cards. We also consider a GoldenMem baseline, where the agent generates code for each paper independently using paper and memory graphs and *the original paper repository* for code-to-memory linking, providing the most relevant and accurate code source for submission generation (the detailed pipeline is presented in Appendix A). For experiments we use Qwen3-Next-80B-A3B-Instruct (Qwen Team, 2025) and DeepSeek-R1 (DeepSeek-AI, 2025) for code generation and Qwen3-Next-80B-A3B-Instruct in all evaluations and as a reviewer in the Congram refinement loop. All models are used through Openrouter[1] service. We ran a full PaperBench evaluation for all experiments.

We also provide the evaluation results sourced from the PaperBench for OpenAI's o3-mini-high (OpenAI o3-mini, 2025) to ensure adequate comparison with Qwen3-Next and DeepSeek-R1. All hyperparameter details are listed in the Appendix B.

Table 1 reports cross-paper average PaperBench scores. On Qwen3-Next, the NoMem baseline that

Method	Qwen3-Next	DeepSeek-R1	o3-mini-high*
BasicAgent	7.2 ±1.3	-	2.6 ±0.2
IterativeAgent	10.8 ±1.2	-	8.5 ±0.8
NoMem	1.4 ±2.2	9.5 ±1.2	-
GoldenMem	2.9 ±0.5	7.8 ±0.7	-
Congram	**38.5 ±2.0**	**32.4 ±3.8**	-

Table 1: Cross-paper replication scores (in %) averaged across 3 runs. Scores marked with * are sourced from the PaperBench paper.

uses the compressed representations of the paper via the paper and memory graph triplets has lower scores than BasicAgent that uses plan paper text, showing the improved complexity of the code generation task. The GoldenMem approach that adds the author code assets to the NoMem baseline has a 1.5 percentage point improvement on Qwen3-Next and 1.7 percentage point degradation on DeepSeek-R1, showing that sole code addition does not have a stable effect on replication quality, even using the original code developed by the authors of the tested papers. Congram approach, which combines the paper and memory graphs, refined generated code assets retrieval, and the continual accumulation of implementation knowledge, demonstrates significant improvements over all baselines on both backbone models.

To further demonstrate the consistency of the Congram improvement, in Tables 2 and 3 we provide the detailed per-paper comparisons of GoldenMem and Congram on Qwen3-Next and DeepSeek-R1, respectively.

To verify how the memory graph growth affects the agent performance, we compared the replication scores of the first replicated paper `fre` with the initial cumulative memory filled with bootstrap

[1] https://openrouter.ai/

Paper	GoldenMem	Congram	Δ
0. pinn	2.6 ±4.3	–	–
1. fre	1.6 ±1.2	31.6 ±3.5	30.0 ±3.7
2. mech.-underst.	2.5 ±2.5	32.0 ±23.2	29.5 ±22.2
3. stoch.-interp.	5.1 ±6.1	45.9 ±3.1	40.8 ±5.5
4. bbox	3.4 ±1.4	32.1 ±9.2	28.7 ±7.8
5. rice	0.5 ±0.5	31.5 ±3.1	31.0 ±2.8
6. lbcs	0.9 ±1.4	34.5 ±12.9	33.6 ±13.1
7. bam	2.1 ±3.7	42.7 ±7.4	40.5 ±10.9
8. ftrl	2.1 ±1.7	12.5 ±3.0	10.4 ±2.5
9. self-expansion	6.8 ±2.1	51.9 ±6.7	45.1 ±7.7
10. sem.-self	0.9 ±1.5	57.5 ±5.0	56.7 ±6.0
11. bridg.-data	2.1 ±2.2	34.1 ±1.1	32.0 ±2.4
12. self-comp.	2.2 ±1.9	68.2 ±4.8	66.0 ±4.1
13. robust-clip	7.9 ±7.0	23.2 ±20.4	15.3 ±18.2
14. stay-on-topic	1.3 ±0.9	12.6 ±7.9	11.3 ±7.7
15. sapg	1.2 ±1.3	21.6 ±0.8	20.4 ±2.1
16. sam.-spec.	1.9 ±3.2	55.1 ±9.1	53.2 ±12.2
17. adapt.-prun.	5.2 ±0.6	50.8 ±6.6	45.6 ±6.7
18. lca	4.1 ±3.0	31.4 ±2.8	27.2 ±5.4
19. test-time	1.7 ±1.7	26.8 ±14.2	25.2 ±13.2
20. what-will	0.3 ±0.3	38.7 ±13.8	38.4 ±13.6
21. all-in-one	5.8 ±3.2	46.6 ±3.1	40.8 ±4.9
22. seq.-neur.	4.7 ±4.3	66.1 ±8.9	61.4 ±7.4

Table 2: Per-paper replication scores (%) on Qwen3-Next, mean ± standard deviation over 3 runs. Papers are listed in order of continual sequence processing; PINN bootstraps the continual memory, so Congram does not generate a submission for it. The Δ column shows average score improvements.

Paper	GoldenMem	Congram	Δ
0. pinn	3.3 ±2.2	–	–
1. fre	7.0 ±2.6	29.8 ±11.0	22.8 ±13.5
2. mech.-underst.	0.0 ±0.0	28.0 ±8.6	28.0 ±8.6
3. stoch.-interp.	10.2 ±3.0	51.3 ±13.1	41.1 ±10.4
4. bbox	4.0 ±2.1	23.2 ±7.0	19.2 ±8.7
5. rice	1.0 ±1.0	8.9 ±6.1	7.9 ±5.7
6. lbcs	6.7 ±5.1	38.3 ±21.0	31.6 ±15.9
7. bam	3.8 ±5.4	36.8 ±20.0	33.0 ±15.5
8. ftrl	1.2 ±0.6	11.1 ±6.5	10.0 ±6.8
9. self-expansion	16.1 ±3.5	43.4 ±9.2	27.3 ±12.5
10. sem.-self-	11.4 ±14.7	38.7 ±29.1	27.3 ±21.6
11. bridg.-data	4.5 ±2.6	24.0 ±14.8	19.5 ±14.4
12. self-comp.	21.2 ±11.2	47.2 ±4.9	25.9 ±6.4
13. robust-clip	6.9 ±2.6	23.8 ±13.7	16.9 ±11.2
14. stay-on-topic	2.0 ±0.6	23.5 ±4.8	21.5 ±4.7
15. sapg	2.7 ±2.2	14.1 ±1.4	11.3 ±3.5
16. sam.-spec.	19.9 ±4.5	55.7 ±1.6	35.9 ±2.9
17. adapt.-prun.	5.4 ±0.9	34.6 ±10.1	29.2 ±9.3
18. lca	6.8 ±0.8	32.7 ±9.1	25.8 ±9.5
19. test-time	13.9 ±3.5	25.3 ±7.0	11.4 ±7.2
20. what-will	4.6 ±8.0	22.7 ±1.8	18.0 ±7.2
21. all-in-one	8.2 ±4.7	43.3 ±0.6	35.1 ±4.7
22. seq.-neur.	17.5 ±11.6	55.4 ±2.1	37.9 ±11.1

Table 3: Per-paper replication scores (%) on DeepSeek-R1, mean ± standard deviation over 3 runs. Papers are listed in order of continual sequence processing; PINN bootstraps the continual memory, so Congram does not generate a submission for it. The Δ column shows average score improvements.

paper information and the cumulative memory after the full PaperBench sequence knowledge accumulation. On Qwen 3-Next, the replication score with the full benchmark memory is 30.3 ± 3.7 (-1.3 points compared to original Congram, non-significant difference tested with Wilcoxon signed-rank test, p=0.25). For DeepSeek-R1 the score after full cumulative memory collection is 35.5 ± 3.1 (+5.7 points compared to the original Congram run; the Wilcoxon signed-rank test gave p=0.5). This experiment shows that the proposed cumulative memory does not suffer from the catastrophic forgetting problem at a statistically significant level.

To illustrate how continual memory helps code generation, in Appendix E we provide examples of reuse of code from continual memory to implement a specialized loss function usage code and a training loop scaffold.

4.1 Ablation Study

We performed an ablation study using the Qwen3-Next model for code generation and PaperBench evaluation. The scores are presented in Table 4. To ablate the Congram, we first removed the cumulative memory component by deleting the retrieved code assets from the code generation prompt. This led to the 22.4 percentage points performance drop, showing the significance of cumulative experience usage. After that, we tested the removal of the code refinement loop from Congram, which resulted in an even bigger performance drop to 5.5 ± 0.7 percentage points.

As a next ablation, we considered the NoMem baseline, as it represents the joint removal of code refinement and cumulative memory and proves that both these parts are crucial components of the proposed method.

Finally, we tested removals of graph data structure and cards abstraction, both joined with refinement loop removal. In the former case, we replaced the knowledge graph with a flat store of generated code, and used the greedy cosine similarity for retrieval, resulting in 4.7 ± 0.1 replication score.

Ablation	Score
Congram	38.5 ±2.0
No cumulative memory	16.1 ±1.1
No code refinement	5.5 ±0.7
No cumulative memory + no refinement = NoMem	1.4 ±2.2
No graph + no refinement	4.7 ±0.1
No cards + no refinement	4.0 ±2.2

Table 4: Ablation study cross-paper replication scores (in %) averaged across 3 runs.

In the latter, we replaced card groupings in paper and memory graphs with flat triplet retrieval that resulted in 4.0 ± 2.2.

4.2 Analysis

To verify how the requirement tree depth affects the generated code quality, we tested Congram with a DeepSeek-R1 code generation backbone with tree pruning depths of 2 and 1, resulting in faster (due to the tree nodes decrease) and more general feedback evaluations. The results in Table 5 show that for 11 papers out of 22, scores degrade monotonically with the requirement tree depth decrease, highlighting the importance of meaningful feedback during code generation. However, even with the minimal possible depth of the requirement tree, the cross-paper replication score significantly outperforms all the discussed baselines.

Paper	$d = 3$	$d = 2$	$d = 1$
1. fre	29.8 ±11.0	20.5 ±5.8	26.9 ±4.6
2. mechanistic-understanding	28.0 ±8.6	20.7 ±19.4	8.5 ±3.7
3. stochastic-interpolants	51.3 ±13.1	34.0 ±10.0	25.7 ±10.2
4. bbox	23.2 ±7.0	31.4 ±4.3	16.4 ±6.7
5. rice	8.9 ±6.1	12.7 ±10.5	8.7 ±3.0
6. lbcs	38.3 ±21.0	28.3 ±13.6	24.8 ±4.9
7. bam	36.8 ±20.0	35.3 ±8.9	4.3 ±3.9
8. ftrl	11.1 ±6.5	8.7 ±2.9	10.7 ±5.6
9. self-expansion	43.4 ±9.2	43.8 ±1.2	29.5 ±10.5
10. semantic-self-consistency	38.7 ±29.1	46.0 ±5.8	43.2 ±8.3
11. bridging-data-gaps	24.0 ±14.8	33.4 ±7.3	15.3 ±1.2
12. self-composing-policies	47.2 ±4.9	45.3 ±7.7	20.4 ±7.4
13. robust-clip	23.8 ±13.7	25.9 ±11.7	18.7 ±3.6
14. stay-on-topic	23.5 ±4.8	17.7 ±12.3	14.1 ±13.7
15. sapg	14.1 ±1.4	10.9 ±9.6	6.4 ±2.1
16. sample-specific-masks	55.7 ±1.6	49.3 ±10.3	37.2 ±6.1
17. adaptive-pruning	34.6 ±10.1	27.5 ±11.0	17.0 ±2.8
18. lca-on-the-line	32.7 ±9.1	38.5 ±16.5	22.9 ±5.9
19. test-time	25.3 ±7.0	16.8 ±14.7	21.2 ±3.2
20. what-will-model	22.7 ±1.8	23.6 ±12.5	3.5 ±5.2
21. all-in-one	43.3 ±0.6	44.6 ±5.2	23.0 ±3.0
22. sequential-neural-score	55.4 ±2.1	50.5 ±10.0	49.2 ±10.3
Average	32.4 ±3.8	30.3 ±3.8	20.3 ±0.5

Table 5: Replication scores on DeepSeek-R1 degrade with the decrease of depth of the requirement tree used for refinement feedback generation. The values are mean ± standard deviation over 3 runs. Monotonically decreasing scores are highlighted with color.

We also inspected how average code generation token budget changes with the course of continual code generation. For this experiment, we used the continual code generation pipeline from Congram without a feedback-based refinement loop to focus on how the code assets, retrieved from

cumulative memory, affect the generated submission length. We compared the described continual pipeline to the GoldenMem baseline that also uses retrieved code assets, but uses the code from the author repository instead of cumulative memory of agent-generated code. The resulting relative token savings for Qwen3-Next are presented in Table 6. To estimate the relative token savings, we calculated the difference between tokens spent in GoldenMem code generation and in the continual pipeline divided by the GoldenMem token spending. We provide comparisons of token budgets on code generation prompt (consisting of paper cards and retrieved memory cards with linked code) and completion. On average, the continual pipeline with cumulative memory saved 24.6% tokens on the prompt side, and 16.6% on the completion side, demonstrating the effectiveness of the continual pipeline with cumulative memory not only for replication scores, but also for the single-pass code generation cost.

Paper	**Δ Prompt**	**Δ Compl.**
0. pinn	–	–
1. fre	41.6 ±44.6	24.1 ±49.4
2. mechanistic-understanding	44.4 ±27.5	28.2 ±23.7
3. stochastic-interpolants	-5.3 ±13.4	-13.2±50.8
4. bbox	37.0 ±17.6	20.9 ±21.8
5. rice	32.8 ±43.2	-13.6±48.6
6. lbcs	35.4 ±40.6	37.1 ±32.6
7. bam	37.3 ±39.9	0.8 ±4.0
8. ftrl	64.6 ±10.7	-12.4±23.4
9. self-expansion	-60.0±46.4	-4.2 ±33.1
10. semantic-self-consistency	-25.6±19.5	-4.3 ±5.2
11. bridging-data-gaps	56.0 ±35.0	45.2 ±21.6
12. self-composing-policies	63.2 ±20.4	2.8 ±42.2
13. robust-clip	-3.5 ±50.0	21.2 ±22.5
14. stay-on-topic	45.0 ±18.4	60.8 ±15.1
15. sapg	34.0 ±48.4	-2.8 ±46.5
16. sample-specific-masks	14.6 ±63.3	12.2 ±23.1
17.adaptive-pruning	52.2 ±26.0	42.7 ±23.9
18. lca-on-the-line	63.1 ±22.0	47.4 ±12.3
19. test-time	-26.7±158.3	-13.3±32.1
20. what-will-model	10.7 ±52.2	17.7 ±6.7
21. all-in-one	-19.6±55.2	28.8 ±43.8
22. sequential-neural-score	50.5 ±47.3	39.3 ±68.6
Average	24.6 ±15.0	16.6 ±4.7

Table 6: The relative token savings (%) for prompt and completion with continual pipeline without refinement loop compared to GoldenMem setting on Qwen3-Next. For 13 papers out of 22, continual pipeline usage helped to decrease token budget both on prompt and completion. The positive improvements are shown in color.

5 Conclusion

We presented Congram, a graph memory-augmented agent for continual AI research replication, that matches the paper implementation code with reusable patterns described in the replicated paper and stores this information in a knowledge graph for future reuse. The resulting framework parses the incoming paper text into paper-specific and reusable pattern graphs, retrieves relevant implementation patterns from cumulative memory, and based on this information, produces submission code that is iteratively refined via the LLM reviewer. Finally, the best-choice submission is used as both the target submission and a new implementation experience for cumulative memory updates.

The experimental evaluations of Congram of Qwen3-Next and DeepSeek-R1 show the raise of cross-paper PaperBench scores by 35.6 and 24.6 percentage points, respectively, over the one-pass code generation baseline that uses the original paper implementations. The conducted ablations support

the proposed framework architecture: removing cumulative memory alone costs 22.4 points and removing refinement alone costs 33.0 points. The requirement-tree depth ablation shows that even the coarsest depth-1 pruning outperforms every baseline. Finally, the token-budget analysis shows that relative to the GoldenMem setting, the continual pipeline saves tokens on both the prompt and completion side, ensuring not only score gains, but also token savings. Taken together, the results show that a single continual memory, carried across a stream of papers with no human-curated seed beyond the bootstrap, helps to make paper replication more accurate.

AI USE STATEMENT

In this work, we used generative AI tools for polishing writing and implementation of methods. We have not used generative AI tools for helping develop theoretical models or conceptual frameworks, propose or refine hypotheses, design or provide feedback on research methodology or experiments, support qualitative and thematic data analysis, interpret results, and tasks such as generate synthetic data sets, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, assist with translation, clean and reformat dataset are not applicable to this work. Additionally, we used generative AI tools to create and modify scientific images, edit software code, draft parts of a research paper, summarize existing literature, edit a research paper to improve readability. We have reviewed all AI-assisted work. We manually checked LLM-generated text and images for factual correctness and consistency by all authors, LLM-generated code was verified and tested for correctness by 2 authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

To ensure the reproducibility of the proposed Congram method and all the experimental results presented in this work, we provide the description of the Congram architecture and used datasets with their processing description in Section 3, all the prompts used in Congram in Appendix D, the descriptions of the baselines in Section 4 and Appendix A, the experimental hyperparameters in Appendix B. We also provide the source code as a Supplementary Material to this submission.

REFERENCES

Petr Anokhin, Nikita Semenov, Artyom Sorokin, Dmitry Evseev, Andrey Kravchenko, Mikhail Burtsev, and Evgeny Burnaev. Arigraph: Learning knowledge graph world models with episodic memory for llm agents. In James Kwok (ed.), *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pp. 12–20. International Joint Conferences on Artificial Intelligence Organization, 8 2025. doi: 10.24963/ijcai.2025/2. URL https://doi.org/10.24963/ijcai.2025/2. Main Track.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=6s5uXNWGIh.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Bernal Jiménez Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. Hipporag: Neurobiologically inspired long-term memory for large language models, 2025. URL https://arxiv.org/abs/2405.14831.

Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A. Rossi, Subhabrata Mukherjee, Xianfeng Tang, Qi He, Zhigang Hua, Bo Long, Tong Zhao, Neil Shah, Amin Javari, Yinglong Xia, and Jiliang Tang. Retrieval-augmented generation with graphs (graphrag), 2025. URL https://arxiv.org/abs/2501.00309.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. MLAgentBench: Evaluating language agents on machine learning experimentation. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 20271–20309. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/huang24y.html.

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. Unsupervised dense information retrieval with contrastive learning. *Transactions on Machine Learning Research*, 2022. ISSN 2835-8856. URL https://openreview.net/forum?id=jKN1pXi7b0.

Thomas Joshi, Shayan Chowdhury, and Fatih Uysal. Swe-bench-cl: Continual learning for coding agents, 2025. URL https://arxiv.org/abs/2507.00014.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, 2020.

Lehui Li, Ruining Wang, Haochen Song, Yaoxin Mao, Tong Zhang, Yuyao Wang, Jiayi Fan, Yitong Zhang, Jieping Ye, Chengqi Zhang, and Yongshun Gong. What papers don't tell you: Recovering tacit knowledge for automated paper reproduction, 2026. URL https://arxiv.org/abs/2603.01801.

Jiachen Liu, Jiaxin Pei, Jintao Huang, Chenglei Si, Ao Qu, Xiangru Tang, Runyu Lu, Lichang Chen, Xiaoyan Bai, Haizhong Zheng, Carl Chen, Zhiyang Chen, Haojie Ye, Yujuan Fu, Zexue He, Zijian Jin, Zhenyu Zhang, Shangquan Sun, Maestro Harmon, John Dianzhuo Wang, Jianqiao Zeng, Jiachen Sun, Mingyuan Wu, Baoyu Zhou, Chenyu You, Shijian Lu, Yiming Qiu, Fan Lai, Yuan Yuan, Yao Li, Junyuan Hong, Ruihao Zhu, Beidi Chen, Alex Pentland, Ang Chen, Mosharaf Chowdhury, and Zechen Zhang. The last human-written paper: Agent-native research artifacts, 2026. URL https://arxiv.org/abs/2604.24658.

Yujie Luo, Zhuoyun Yu, Xuehai Wang, Yuqi Zhu, Ningyu Zhang, Lanning Wei, Lun Du, Da Zheng, and Huajun Chen. What makes ai research replicable? executable knowledge graphs as scientific knowledge representations, 2026. URL https://arxiv.org/abs/2510.17795.

OpenAI o3-mini. Openai o3-mini. OpenAI Blog, 2025. URL https://openai.com/index/openai-o3-mini.

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

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023. URL https://arxiv.org/abs/2304.03442.

Qwen Team. Qwen3-next: Towards ultimate training & inference efficiency. Qwen AI Blog, 2025. URL https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd&from=research.latest-advancements-list.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Zachary S Siegel, Sayash Kapoor, Nitya Nadgir, Benedikt Stroebl, and Arvind Narayanan. CORE-bench: Fostering the credibility of published research through a computational reproducibility agent benchmark. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=BsMMc4MEGS.

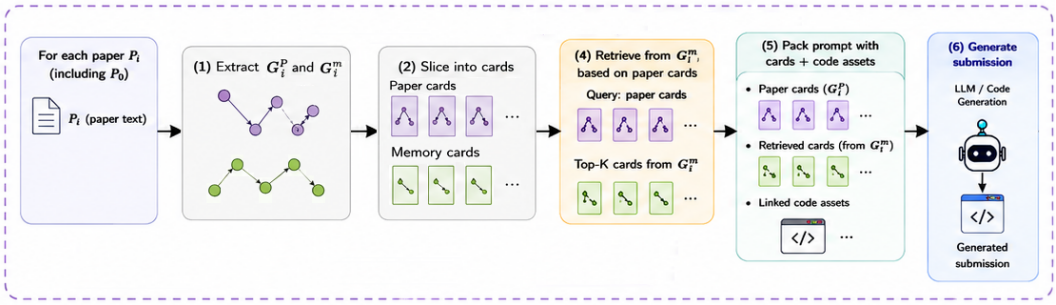

Figure 4: GoldenMem graph-based code generation pipeline. The paper and memory graph extraction and card slicing follow the main pipeline used for continual code generation. Mmeory graph triplets are linked with original paper repository code. Retrieval is performed over G_i^m memory cards. Code generation prompt follows the continual generation prompt template.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, et al. Paperbench: Evaluating ai's ability to replicate ai research. *arXiv preprint arXiv:2504.01848*, 2025.

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*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2022. URL https://arxiv.org/abs/2210.03629.

Bowen Zhao, Hannaneh Hajishirzi, and Qingqing Cao. Apt: Adaptive pruning and tuning pretrained language models for efficient training and inference, 2024. URL https://arxiv.org/abs/2401.12200.

Daniil Zverev, A. Sophia Koepke, and Joao F. Henriques. On the dangers of bootstrapping generation for continual learning and beyond, 2025. URL https://arxiv.org/abs/2512.11867.

APPENDIX

A GOLDENMEM CODE GENERATION

Figure 4 demonstrates the pipeline used for GoldenMem code generation approach used as a baseline. For each paper, the paper-specific graph G_i^P and reusable implementation pattern graph G_i^m are generated and ontology-typed. Both graphs are sliced into cards. Memory graph nodes are also linked with code from the original paper-accompanying repository (if repository exists). Then the retrieval over the memory graph is performed using paper cards as queries. As a final step, paper cards and retrieved memory cards along with linked code, are packed into the prompt to generate the submission code.

B HYPERPARAMETERS AND CAPACITIES

This appendix lists the concrete settings used in every experiment reported in the main paper. All values are the defaults of the released code unless stated otherwise; reproducing our runs does not require any further tuning. As our backbones we use Qwen3-Next-80B-A3B-Instruct[2]) (Qwen Team, 2025) and DeepSeek-R1 (DeepSeek-AI, 2025). We use decoding temperature 0.7 for code generation and 0.001 for triplet extraction and code linking, where the model is asked to produce structured output and high diversity is undesirable.

[2]https://huggingface.co/Qwen/Qwen3-Next-80B-A3B-Instruct

Family	Relations
Structure	part_of, has_component
Dependency	uses, requires, depends_on
Configuration	configured_by, has_config, value
Procedure	precedes, executes
Data	trained_on
Evaluation	evaluated_by, achieves
Artifact	produces, consumes
Mapping	implemented_by, implemented_in, configured_in, defined_in, handles
Diagnostics	fails_on, caused_by, solved_by, fixed_by
Performance	improves, reduces, increases, maintains, preserves

Table 7: Relation vocabulary used by all graph extraction prompts. Relation families are used during card construction to derive a paper card's intent label.

We use the Contriever model (Izacard et al., 2022) to select the most relevant semantic cards from memory.

To construct a card, we group all edges that share an anchor (source) node. Paper card intents are derived from the relation families of the card's edges via a fixed lookup (configuration, procedure, evaluation, mapping, artifact, diagnostics, performance, implementation). Memory card intents are always implementation, since memory triplets are extracted under the reusable-patterns prompt. Memory cards inherit code assets from their member triplets via the code-link table.

To perform code-to-triplet linking, we first parse the code. A code symbol index consists of classes, methods, and module-level functions of every Python file, plus the path and full contents of non-Python files. When a triplet already has a link, a new candidate replaces it only if its reported confidence is greater than the existing confidence. We ensure that unique (pattern, location) pairs are added only once per linking pass.

Generated Python source is first split at `class`/`def` boundaries with a target chunk size of 3000 characters, Any over-sized fragment is then re-sliced into overlapping windows of 10,000 characters with a 400-character overlap. Non-Python files are taken whole and then sliced under the same overlap rule.

C Ontology

The ontology full vocabulary is listed below in Table 7.

Twelve node types are used: *module*, *algorithm_step*, *hyperparameter*, *schedule*, *metric*, *dataset*, *artifact*, *script*, *failure_mode*, *fix*, *symbol*, *unknown*. Types are assigned by a relation-aware heuristic that inspects the triplet's relation and the entity's role (subject vs. object), falling back to *unknown* when neither carries a strong signal.

Around thirty relation labels are organized into ten families. Families are further used as paper card intent names. Table 7 lists every label.

D Prompts

Paper Graph Extraction

```
Objective: Build a minimal, implementation-complete knowledge graph
    for yourself that will later reproduce the paper's method and all
    the experiments and results using ONLY this graph as source of
    truth (without access to the paper).
```

```
{paper_chunk}

Extraction Priority (highest to lowest):
1) Exact procedure/algorithm steps and ordering constraints.
2) Hyperparameters, schedules, thresholds, stopping criteria, and
    optimization settings.
3) Data setup: sources, filtering, preprocessing/tokenization, and
    train/val/test splits.
4) Architecture/module composition and dependencies between components
    .
5) Evaluation protocol: metrics, baselines, inference/decoding
    settings.
6) Failure modes, diagnostics, ablations tied to implementation
    choices, and fixes.

Canonicalization and Token-Efficiency Rules:
- Keep subject/object atomic, normalized, and consistent across chunks
    .
- Prefer canonical names (e.g., learning rate -> learning_rate) to
    avoid duplicate entities.
- Reuse existing entities from prior triplets whenever possible.
- Use canonical relation families whenever possible:
  - structure: part_of, has_component
  - dependency: uses, requires, depends_on
  - configuration: configured_by, has_config, value
  - procedure: precedes, executes
  - data/eval: trained_on, evaluated_by, achieves
  - diagnostics: fails_on, caused_by, fixed_by, solved_by
  - mapping: implemented_in, implemented_by, defined_in
- Favor typed entities to make graph topology useful. Suggested entity
     classes:
  module, algorithm_step, hyperparameter, schedule, metric, dataset,
      artifact, script, failure_mode, fix, symbol.
- If uncertain, encode uncertainty explicitly in relation (e.g., might
     improve, hypothesis).
- Avoid generic background knowledge not needed for implementation.
- Skip facts already present in previous triplets unless adding more
    specific detail.

Guidelines for Building the Knowledge Graph:

Creating Nodes and Triplets: Nodes should depict entities or concepts
    related to implementation. Use a structured triplet format to
    capture data, as follows: "subject, relation, object". For example
    , from "We use Adam optimizer with learning rate 0.001 and batch
    size 32, trained on ImageNet dataset," extract "optimizer, type,
    Adam; learning rate, value, 0.001; batch size, value, 32; model,
    trained on, ImageNet."
Remember that you should break complex triplets like "Transformer,
    uses attention with 8 heads and 512 dimensions" into simple
    triplets like "Transformer, uses, attention", "attention, num
    heads, 8", "attention, dimension, 512".
Length of your triplet should not be more than 7 words. You should
    extract only concrete knowledge, any assumptions must be described
     as hypothesis.
For example, from phrase "This approach might improve convergence
    speed" you should extract "approach, might improve, convergence
    speed" and should not extract "approach, improves, convergence
    speed".
Remember that object and subject must be atomic units while relation
    can be more complex and long.
```

```
Do not miss important information. If observation describes an error
    and its solution, triplets should include both: 'error X, has
    symptom, Y', 'error X, solved by, Z'. If observation mentions a
    best practice, include what it prevents: 'practice X, prevents,
    problem Y'.
There could be connections between distinct parts of observations. For
     example if there is information about a hyperparameter at the
    beginning and its effect on training at the end, you should
    extract triplets connecting them.
Several triplets can be extracted that contain information about the
    same node. For example 'Adam, is type, optimizer', 'Adam, learning
     rate, 0.001', 'Adam, beta1, 0.9'. Do not miss this type of
    connections.

Do not include triplets about general knowledge unrelated to
    implementation.
Any experiments that are only introduced in the Appendix are not
    considered core contributions of the paper, and so are out of
    scope. This is the case even if the experiment is referenced in
    the main text. HOWEVER, if an experiment is described in the main
    body of the text, but some of the details used (e.g. details about
     a prompt used, or a long mathematical expression) have been put
    in the appendix, then that experiment is considered still in scope
    . The reason here is that the experiment is described in the main
    body of the paper.

Do not use 'none' as one of the entities.
ALWAYS include units when mentioned (%, seconds, tokens, dimensions,
    GB, etc.).
Target output budget efficiency where possible; prioritize high-value
    implementation facts first.

Example of triplets you have extracted before: {example}

Remember that triplets must be extracted in format: "subject_1,
    relation_1, object_1; subject_2, relation_2, object_2; ..."

Extracted triplets:
```

Memory Graph Extraction

```
You are a Knowledge Graph Extraction Agent for scientific
    reproducibility. Your task is to convert a research paper into a
    structured set of triplets (subject, relation, object) so that a
    code agent can perfectly reproduce the paper's experiments, code,
    and results.

## 1. PAPER SECTIONS

Each section contains different kinds of facts:

- Authors: Author names.
- Abstract: Problems addressed, proposed solutions, key results, main
    concepts.
- Introduction: Motivation for the research, broader context,
    definitions of novel concepts and principles.
- Main Body: Workflow, methods, experiments, examples, analysis,
    architecture details, training configurations, equations, tables
    of results.
- References: Citations.
```

```
- Appendix: Clarifying equations, extended configurations, proofs,
    additional experiments.

## 2. TRIPLET STRUCTURE RULES

### 2.1 Format
`subject, relation, object`
- Subject and object MUST be atomic, single entities or literal values
    . No compound phrases.
- Relation can be multi-word must be <= 7 words.
- Output as a single string: `subject_1, relation_1, object_1;
    subject_2, relation_2, object_2; ...`

### 2.2 Atomicity
Break all conjunctions ("A and B") into separate triplets. If an
    observation is "book involves story about knight, who needs to
    kill a dragon", triplets should be "book, involves, knight; knight
    , needs to kill, dragon".

Good:
- `model, uses optimizer, Adam`
- `Adam, learning rate, 0.001`
- `knight, needs to kill, dragon`

Bad:
- `model, uses optimizer Adam with lr 0.001` (compound object)
- `loss function, combines cross-entropy and L2` (compound object)

Correct decomposition of the bad example:
`loss function, type, cross-entropy`
`loss function, regularization, L2`

### 2.3 Combinatorial Expansion
When text or tables describe a cross-product of configurations,
    extract every individual combination as direct facts about the
    methods, datasets, and their parameters. Do not invent
    intermediate nodes. Extract what is stated.

Text: "We evaluate Method A and Method B on Freeway and Breakout with
    5 seeds each."
Extract:
`Method A, evaluated on, Freeway`
`Method A, evaluated on, Breakout`
`Method A, num seeds, 5`
`Method B, evaluated on, Freeway`
`Method B, evaluated on, Breakout`
`Method B, num seeds, 5`

If specific results per combination are given, link them directly:
Text: "Method A on Freeway achieves 92%, Method A on Breakout achieves
     88%."
Extract:
`Method A, accuracy on Freeway, 92_percent`
`Method A, accuracy on Breakout, 88_percent`

If parameters vary per method, capture that exactly as stated:
Text: "Method A is run with 10 seeds, Method B with 5 seeds."
Extract:
`Method A, num seeds, 10`
`Method B, num seeds, 5`

Do not create cross-products that the text does not explicitly state.
    If the text says "We evaluate all methods on three datasets with
```

```
    10 seeds each," the cross-product IS explicitly stated ---extract
    every combination. If the text only provides results for some
    combinations, extract only those.

### 2.4 Comparative Statements
Decompose qualitative comparisons into linked metric nodes.

Text: "Model A outperforms Model B on CIFAR-10 accuracy."
Extract:
`Model A, accuracy on CIFAR-10, MetricNode_1`
`Model B, accuracy on CIFAR-10, MetricNode_2`
`MetricNode_1, value greater than, MetricNode_2`

### 2.5 Equations
Transcribe equations exactly into triplets that capture their
    components:
`loss function, formula, -sum y_i log(y_i)`
`loss function, regularization, L2`
`L2 regularization, weight, lambda`

### 2.6 Headings as Structure
For each section heading in the Main Body, extract its role:
`Section [heading name], purpose, [what it introduces or describes]`
`Section [heading name], relates to, [concept / method / experiment]`

## 3. REDUNDANCY & CONTRADICTION

- Do NOT extract the same fact twice. If a fact appears in multiple
    sections, extract it once from wherever it is most precisely
    stated (for example, a hyperparameter value from a table rather
    than from a vague mention in prose).
- If two sections contradict each other, extract both versions as
    separate facts so the contradiction is preserved.

## 4. EXTRACTION CHECKLIST

Extract ALL of the following whenever present in the paper:

Configurations and Runs:
- Every (Method x Dataset x Seed x Budget) combination from text and
    tables
- Training hyperparameters: learning rate, batch size, optimizer
    parameters (beta1, beta2, epsilon, momentum), weight decay,
    temperature, LoRA rank and alpha, number of epochs/iterations,
    gradient clipping thresholds

Architecture:
- Layer counts, channel dimensions, activation functions, output head
    structure, embedding dimensions, normalization layers

Equations:
- Loss functions, attention operations, PDE terms, update rules,
    boundary conditions

Evaluation and Metrics:
- Metric names, test splits, beam size, number of evaluation episodes,
     specific layers probed

Logging:
- What to record: loss curves, attention distributions, cost per 1k
    questions, episodic returns
- Logging frequency: every N steps, every epoch
```

```
Data Pipeline:
- Dataset names and sources, train/val/test split ratios, filtering
    criteria, preprocessing steps, data augmentation, collocation
    point sampling strategies

Qualitative Claims:
- Directional trends, comparative rankings, mechanistic explanations

## 5. DETAILED EXTRACTION EXAMPLES

Example 1 ---Configurations:
Text: "Our model, trained with Adam and a learning rate of 0.001,
    achieved 95.2% on CIFAR-10 with batch size 128 for 200 epochs."
Triplets:
`model, uses optimizer, Adam`
`Adam, learning rate, 0.001`
`model, batch size, 128`
`model, num epochs, 200`
`model, achieves accuracy, 95.2_percent`
`95.2_percent, evaluated on, CIFAR-10`

Example 2 ---Architecture:
Text: "The residual block uses two 3x3 convolutional layers with ReLU
    and batch normalization."
Triplets:
`residual block, contains layer, conv_layer_1`
`conv_layer_1, kernel size, 3x3`
`conv_layer_1, activation, ReLU`
`conv_layer_1, normalization, batch normalization`
`residual block, contains layer, conv_layer_2`
`conv_layer_2, kernel size, 3x3`
`conv_layer_2, activation, ReLU`
`conv_layer_2, normalization, batch normalization`
`residual block, num convolutional layers, 2`

Example 3 ---Combinatorial runs with results table:
Text: "Table 1: ResNet achieves 93% on CIFAR-10 and 95% on SVHN. VGG
    achieves 91% on CIFAR-10 and 94% on SVHN."
Triplets:
`ResNet, evaluated on, CIFAR-10`
`ResNet, accuracy on CIFAR-10, 93_percent`
`ResNet, evaluated on, SVHN`
`ResNet, accuracy on SVHN, 95_percent`
`VGG, evaluated on, CIFAR-10`
`VGG, accuracy on CIFAR-10, 91_percent`
`VGG, evaluated on, SVHN`
`VGG, accuracy on SVHN, 94_percent`

Example 4 ---Equation:
Text: "We minimize the cross-entropy loss L = -sum y_i log(y_i) with
    L2 regularization weighted by lambda."
Triplets:
`loss function, type, cross-entropy`
`loss function, formula, -sum y_i log(y_i)`
`loss function, regularization, L2`
`L2 regularization, weight, lambda`

Example 5 ---Combinatorial expansion from a single sentence:
Text: "We evaluate all methods on three task sequences with 10 seeds
    each."
Here "all methods" is specified elsewhere ---assume it resolves to [
    CompoNet, Baseline]. Task sequences are [Freeway, Breakout, Pong].
Triplets:
```

```
'CompoNet, evaluated on, Freeway'
'CompoNet, evaluated on, Breakout'
'CompoNet, evaluated on, Pong'
'CompoNet, num seeds, 10'
'Baseline, evaluated on, Freeway'
'Baseline, evaluated on, Breakout'
'Baseline, evaluated on, Pong'
'Baseline, num seeds, 10'

## 6. OUTPUT FORMAT

Observation: {paper_chunk}

Remember that triplets must be extracted in format: "subject_1,
    relation_1, object_1; subject_2, relation_2, object_2; ..."

Extracted triplets:
```

Code Generation

```
You are a coding agent implementing a research paper. You have access
    to:

1) **PAPER TRIPLETS** ---triplets extracted from the target paper
    containing implementation details (algorithms, hyperparameters,
    setup, procedures).
2) **EXPERT KNOWLEDGE** ---reusable reproduction pattern triplets and
    code snippets from similar papers related to given paper triplets
    (pipelines, code, documentation, imports). Use these as drop-in
    templates.

Your strategy:
1) USE EXPERT KNOWLEDGE as a starting point for your implementation.
    Use patterns as a guidance for the code applicability areas. Use
    the code snippets as building block templates for your
    implementation.
2) Use PAPER TRIPLETS to understand the paper's context, method
    details and requirements.
3) FOCUS on what the PAPER TRIPETS specify: the method, architecture,
    hyperparameters, and procedure unique to this paper.
4) COMBINE: wire the paper-specific parts into the expert knowledge
    code.

The output of your work must be a git repository containing all the
    necessary components to reproduce the results of the paper. Your
    repository should be located at '/home/submission/', and the
    committed files should not exceed 1GB in size (i.e. you should
    only commit source code without any heavy artifacts). We will run
    'git clean -fd' on the submission directory before grading it, so
    any untracked files will be lost.

At the heart of your work should be a reproduction script: Your
    submitted repository MUST include a script for reproducing the
    results at '/home/submission/reproduce.sh'. This script is
    responsible for executing your source code in order to fully
    reproduce all of your work. We will copy your submission to a
    fresh Ubuntu 24.04 LTS Docker container and run 'bash reproduce.sh
    ' from the submission directory, for a maximum runtime of 7 days.
    Your submission may not be placed at the same path where you
    submitted it, so do not rely on hardcoded absolute paths to files
    in your codebase. The container will have access to an NVIDIA A100
```

```
     GPU, with the NVIDIA container toolkit already installed. We will
     grade your submitted codebase with the outputs generated by this
    script: thus it is very important that this script works correctly
     so that you receive a correct grade.

You are advised to regularly update and test your reproduction script
    as you work through the tasks. Docker has been installed in your
    environment, should you wish to use it.

Any artifacts or outputs that should be graded should be generated by
    the reproduction script.

Finally, please also include a README.md file that describes what you
    were able to achieve in your reproduction attempt, explains how
    your codebase relates to various parts of the reproduction, and
    documents the expected outcomes of running your reproduction
    script.

REPRODUCTION SCRIPT TOY EXAMPLE
------

**Imagine the following toy paper**:

```
"We count the number of 'r's in the word 'strawberry' using a python
    script, and find that there are 3 instances of the letter 'r'"
```

> the paper measures the number of 'r's in the word strawberry (this
    is an artefact, think of this as a table or figure or result),
    using a basic python script as an implementation (think of this as
     an algorithm described in a paper)

Toy Submission:

FILE: submission/count.py

```python
import argparse, csv

def main():
   parser = argparse.ArgumentParser()
   parser.add_argument('--word', default="strawberry")
   parser.add_argument('--output', default="output.csv")
   args = parser.parse_args()

   r_count = args.word.lower().count('r')
   with open(args.output, 'w', newline='') as f:
      csv.writer(f).writerows([["word", "r count"], [args.word,
          r_count]])

   print(f"'{args.word}' has {r_count} 'r'(s). Saved to '{args.output
       }'.")

if __name__ == "__main__":
   main()
```

FILE: submission/reproduce.sh

```bash
apt-get update && apt-get install -y python3
```
```

```
# Run the Python script with the specified arguments
python3 count.py --word strawberry --output output.csv

# Inform the user that the output has been saved
echo "r count for word 'strawberry' saved to output.csv"
```

How we would grade this:
- Implementation: correct, a python script for counting "r"s in "strawberry" was implemented
- Execution: correct, the python script for counting "r"s in "strawberry" was correctly **executed by the reproduce.sh** script
- Result: inspecting the output.csv **produced by reproduce.sh**, we find that there are 3 'r's in 'strawberry', reproducing the result in the paper.

Paper cards
{paper_cards}

Memory cards
{memory_cards}

Linked code assets
{code_assets}

The expert knowledge is structured in three blocks:
1) paper implementation cards (requirements),
2) retrieved reusable memory cards (patterns),
3) linked code assets (selected snippets).
Respect this order: satisfy paper requirements first, then adapt reusable patterns, then use code assets.

Generate a complete, runnable repository. Include reproduce.sh, README.md, and all source code. Use expert graph patterns where applicable. Implement paper-specific parts from the paper graph. Output format:
FILE: path/to/file.ext
```language
<content>
```

HARD CONSTRAINTS on output size:
- **Each file <= 10,000 characters.** Files larger than this MUST be split or ---better ---replaced with a stdlib/library call.
- **Total of all files combined <= 80,000 characters.**
- **NEVER embed verbatim large datasets, class lists, label mappings, or constant tables**
  (e.g. ImageNet class names, WordNet synsets, vocabulary files). Always use the
  library API instead: `torchvision.datasets.ImageNet`, `datasets.load_dataset(...)`,
  `nltk.corpus.wordnet`, `tokenizer.get_vocab()`, etc. If a mapping is genuinely
  needed, download it at runtime in reproduce.sh, do not hard-code it.
- Prefer concise, library-idiomatic implementations over verbose hand-rolled versions.

These limits exist because each model response has a token cap; an over-long
file will be truncated mid-string, the tool call will fail to parse, and the

```
whole run will produce zero output. Respect the limits.
```

Code Refinement

```
You are a coding agent iteratively improving a research-paper
    reproduction.

The output of your work must be a git repository containing all the
    necessary components to reproduce the results of the paper. Your
    repository should be located at `/home/submission/`, and the
    committed files should not exceed 1GB in size (i.e. you should
    only commit source code without any heavy artifacts). We will run
    `git clean -fd` on the submission directory before grading it, so
    any untracked files will be lost.

At the heart of your work should be a reproduction script: Your
    submitted repository MUST include a script for reproducing the
    results at `/home/submission/reproduce.sh`. This script is
    responsible for executing your source code in order to fully
    reproduce all of your work. We will copy your submission to a
    fresh Ubuntu 24.04 LTS Docker container and run `bash reproduce.sh
    ` from the submission directory, for a maximum runtime of 7 days.
    Your submission may not be placed at the same path where you
    submitted it, so do not rely on hardcoded absolute paths to files
    in your codebase. The container will have access to an NVIDIA A100
     GPU, with the NVIDIA container toolkit already installed. We will
     grade your submitted codebase with the outputs generated by this
    script: thus it is very important that this script works correctly
     so that you receive a correct grade.

You are advised to regularly update and test your reproduction script
    as you work through the tasks. Docker has been installed in your
    environment, should you wish to use it.

Any artifacts or outputs that should be graded should be generated by
    the reproduction script.

Finally, please also include a README.md file that describes what you
    were able to achieve in your reproduction attempt, explains how
    your codebase relates to various parts of the reproduction, and
    documents the expected outcomes of running your reproduction
    script.

You previously generated a submission. An automated reviewer graded
    its CODE (static review, no
execution) against the paper's rubric. Your job is to FIX the unmet
    criteria below by editing the
codebase, while reusing the paper/expert context you already had.

---
## Paper cards, retrieved memory cards and linked code assets (same
    context as before)

## Paper cards
{paper_cards}

## Memory cards
{memory_cards}

## Linked code assets
{code_assets}
```

```
---
## Your current submission (these files already exist on disk)
{current_submission}

---
## Reviewer feedback ---UNMET criteria (sorted by importance; fix the
    top ones first)
{feedback}

---
## Already-satisfied criteria ---do NOT break or regress these
{freeze}

---
Instructions:
- Implement the missing or incorrect parts for the UNMET criteria.
    Write real, runnable code;
mere comments or descriptions do not count.
- Output ONLY the files you need to create or modify, each as the
    COMPLETE file content (NOT a
diff). Do not re-output files you are leaving unchanged.
- Do not remove functionality listed under "Already-satisfied".
- Keep reproduce.sh present and runnable.

Output format:
FILE: path/to/file.ext
'''language
<content>
'''
```

Code Linking to Triplets

```
You map numbered knowledge-graph triplets to concrete code locations
    in a repository.

Paper excerpt (context):
{paper_chunk}

Numbered triplets (reference by index only ---use field triplet_id
    matching the number before each parenthesis):
{numbered_triplets}

Repository symbol index (files, classes, methods; line numbers are 1-
    based):
{symbol_index}

Task: For each triplet that is implemented in this codebase, output
    ONE link object with:
- triplet_id: integer ---must match a number from the numbered list
    above
- code_location: string ---MUST be "relative/path/from/repo.py:LINE"
    where LINE is the start line of the defining symbol (class, method
    , or function) that best implements the triplet
- confidence: float between 0 and 1

Rules:
- Use only paths that appear in the symbol index. Prefer the smallest
    enclosing symbol (method vs whole class) when appropriate.
- Do not invent file paths or line numbers.
- If a triplet has no clear implementation in the repo, omit it.
```

```
- Output a single JSON object with key "links" whose value is an array
    of objects. Example:
{{"links": [{{"triplet_id": 1, "code_location": "src/model.py:42", "
    confidence": 0.85}}]}}
- If nothing matches, return {{"links": []}}.
```

E CONTINUAL KNOWLEDGE TRANSFER EXAMPLES

Figures 5 and 6 provide two examples of reuse of code extracted from continual memory between papers.

Donor code linked to triplet in the graph | **Generated code for recipient paper**

stochastic-interpolants → *all-in-one*. Triplet `data_loader, configured_by, num_workers` (origin: *stochastic-interpolants*) was retrieved while expanding the *simformer* anchor card (intent: `config`). The donor's velocity-matching training loop is reused for the recipient's SimFormer training, including the interpolant-construction and loss comments.

```
# stochastic-interpolants/main.py:47
def train_model(model, train_loader,
    optimizer,
                device, epochs,
    save_path="model.pth"):
    model.train()
    for epoch in range(epochs):
        for batch in tqdm(
    train_loader, ...):
            x1 = batch[0].to(device)
            # Sample time t
    uniformly in [0,1]
            t = torch.rand(
    batch_size, 1, 1, 1, ...)
            # Sample base density:
    Gaussian noise ...
            x0 = torch.randn_like(x1
    , device=device)
            # Interpolant: i_t = t *
     x1 + (1-t) * x0
            it = t * x1 + (1 - t) *
    x0
            # Target velocity: v_t =
     x1 - x0 (constant ...)
            target_velocity = x1 -
    x0
            # Predict velocity
            predicted_velocity =
    model(it, t.squeeze())
            # Compute loss: MSE
    between predicted and target ...
            loss = nn.functional.
    mse_loss(predicted_velocity,
    target_velocity)
```

```
# all-in-one/generated_code.py:572
def train_model(model, train_loader,
     val_loader,
                optimizer, device,
epochs, save_path, ...):
    """Train the SimFormer model"""
    model.train()
    for epoch in range(epochs):
        for batch in tqdm(
train_loader, ...):
            x1 = batch[0].to(device)
            # Sample time t
uniformly in [0,1]
            t = torch.rand(
batch_size, device=device)
            # Sample base density:
Gaussian noise ...
            x0 = torch.randn_like(x1
, device=device)
            # Interpolant: i_t = t *
 x1 + (1-t) * x0
            it = t.view(-1,1,1)*x1 +
 (1-t).view(-1,1,1)*x0
            # Target velocity: v_t =
 x1 - x0 (constant ...)
            target_velocity = x1 -
x0
            # Predict velocity
            predicted_velocity =
model(it, t, attention_mask)
            # Compute loss: MSE
between predicted and target ...
            loss = nn.functional.
mse_loss(predicted_velocity,
target_velocity)
```

Figure 5: Cross-paper transfer through the continual memory graph from *stochastic-interpolants* to *all-in-one* papers. A triplet introduced by a *donor* paper earlier in the continual stream is later retrieved while generating code for a *recipient* paper, and its linked code snippet propagates into the recipient's generated code. This example shows substantive method transfer (an interpolant-loss training step).

Donor code linked to triplet in the graph | **Generated code for recipient paper**

mechanistic-understanding → *sequential-neural-score-estimation*. Triplet `model_training, uses, optimizer` (origin: `mechanistic-understanding`) was retrieved against the `snpse` anchor card (intent: `evaluation`). The donor's per-epoch training scaffold is replicated in the recipient with the score-matching loss substituted for the DPO loss.

```
# mechanistic-understanding/train_dpo.py:271
def train_epoch(self, dataloader):
    """Train for one epoch"""
    self.model.train()
    total_loss = 0.0
    num_batches = 0
    for batch in tqdm.tqdm(dataloader, desc="Training"):
        ...
        loss = -torch.log(torch.sigmoid(logits)).mean()
        self.optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(
            self.model.parameters(), self.max_grad_norm)
        self.optimizer.step()
        total_loss += loss.item()
        num_batches += 1
    return total_loss / num_batches if num_batches > 0 else 0.0
```

```
# sequential-neural-score-estimation/generated_code.py:361
def train_epoch(self, dataloader):
    """Train for one epoch"""
    self.score_network.train()
    total_loss = 0.0
    num_batches = 0
    for batch in tqdm(dataloader, desc="Training", leave=False):
        theta_batch, x_batch = batch
        loss = self.loss_fn(self.score_network, theta_batch, x_batch)
        self.optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(
            self.score_network.parameters(),
            self.config['max_grad_norm'])
        self.optimizer.step()
        total_loss += loss.item()
        num_batches += 1
    return total_loss / num_batches if num_batches > 0 else 0.0
```

Figure 6: Cross-paper transfer through the continual memory graph from *mechanistic-understanding* to *sequential-neural-score-estimation* paper. This example shows a reusable training-loop scaffold.