Continual Graph Memory Agent for AI Research Replication

Anonymous ACL submission

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 rediscover 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. Evaluated on papers from PaperBench, Congram demonstrates significant improvements in token usage (in average, 7763 tokens less for prompt and 1724 tokens less for completion for Qwen3-Next backbone) for code generation compared to Zero-shot setting while preserving the comparable replication quality scores.

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 study (Siegel et al., 2024; Huang et al., 2024; Chan et al., 2025) this setting as a long-horizon ML engineering task, including agents that reproduce papers from existing repositories or solve end-to-end ML tasks. PaperBench (Starace et al., 2025) makes the task especially 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. 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 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. We find that continual knowledge graph memory improves reproduction quality over standard and iterative paper-to-code baselines: Congram uses 7763 less tokens for prompt and 1724 less tokens for completion with Qwen3-Next backbone. It also achieves quality comparable to constructing a separate memory for each paper from its original repository, while reducing the number of generated code tokens. These results show that reusable implementation experience stored in knowledge graph-based memory can make code generation more efficient and scalable. Our contributions are: (i) Congram reduces the code generation completion token budget compared to Zero-shot knowledge graph-

augmented code generation while preserving the comparable code quality according to PaperBench scores; (ii) code generated with graph memory-augmented agents both in Zero-shot and continual regimes outperforms plain paper text-based generation baselines (vanilla agent and iteratively improving one) for Qwen3-Next backbone.

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.

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 submission, 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 in Appendix C.

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.

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.

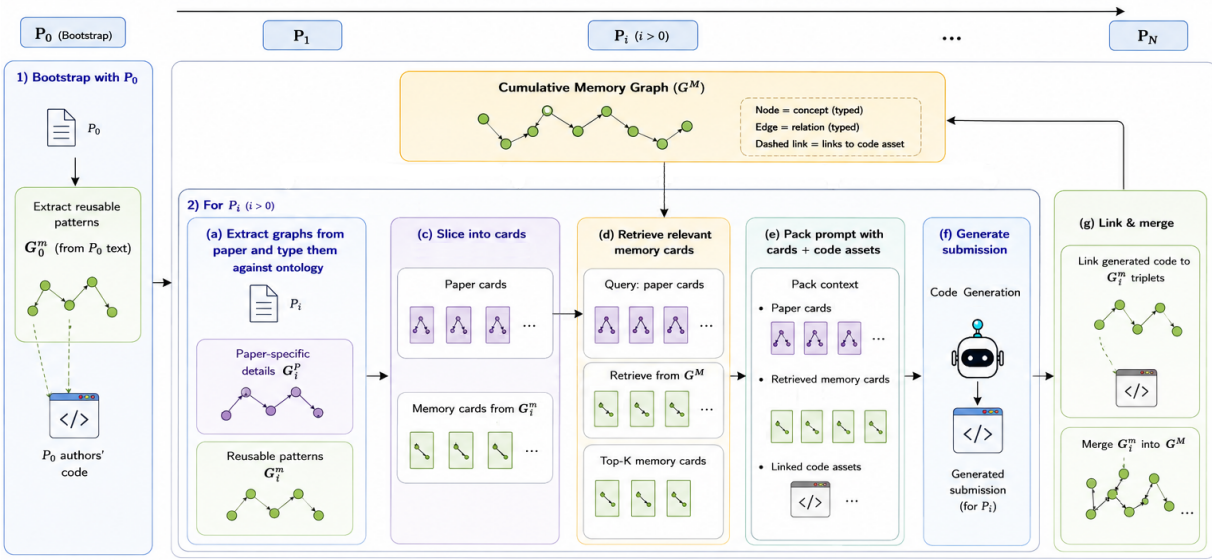

Figure 1: Congram code generation pipeline.

Before retrieval, each raw triplet list is materialized into a typed graph: 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.

For retrieval 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.

After generation, G^M is updated from the agent's own submission in two steps. First, the generated code is chunked and run through the 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 is the only point at which authors' code influences G^M.

4 Experimental Results

For experiments we use PaperBench benchmark (Starace et al., 2025). Each of 23 papers has a benchmark harness that returns a scalar score in $[0, 1]$ for a submission. As baselines we use two paper-text only generation baselines (following the benchmark paper): BasicAgent generates code directly from paper text, and IterativeAgent applies self-refine 5-iteration loop using compile checks. As a graph memory-augmented baseline and a natural ablation of Congram we also consider the strong Zero-shot code generation pipeline where agent operates paper and memory graphs the same way as Congram except code linking stage where for linking the original paper repository is used, providing the most relevant and accurate code source for final submission generation. For code generation we use Qwen3-Next-80B-A3B-Instruct (Qwen Team, 2025) for all our baselines and Congram, and Gemini 2.0 Flash (Google Deepmind, 2024) for Zero-shot and Congram comparison. All hyperparameter details are listed in Appendix B.

Table 1 reports cross-paper average PaperBench scores. All methods have comparable performance on Qwen3-Next. We also provide the evaluation results sourced fron the PaperBench for Gemini 2.0 Flash (Google Deepmind, 2024) and and Ope-

Method	Qwen3-Next	Gemini 2.0 Flash	o3-mini*
BasicAgent	0.08 ± 0.01	0.03 ± 0.00 *	0.03 ± 0.02
IterativeAgent	0.13 ± 0.01	- *	0.08 ± 0.09
Zero-shot	0.08 ± 0.07	0.04 ± 0.00	-
Congram	0.05 ± 0.04	0.02 ± 0.00	-

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

nAI's o3-mini (OpenAI o3-mini, 2025) to ensure the adequate comparison with Qwen3-Next. As a result, Qwen3-Next with 3B activated parameters and 262k tokens context window in average performs better than Gemini 2.0 Flash with maximal sequence length of 1M tokens.

We demonstrate how average code generation token budget changes with the course of continual code generation with Congram compared to Zero-shot generation in tables 2 (for Qwen3-Next) and 3 (for Gemini 2.0 Flash). To estimate the token budget improvement we calculated the difference between token spent in Zero-shot code generation and in continual regime (Congram). We provide comparisons of token budgets on code generation prompt (consisting of paper cards and retrieved memory cards with linked code) and completion. The results on Qwen3-Next show that for a half of

PAPER	Δ PROMPT	Δ COMPL.
0. PINN	-	-
1. FRE	10632 ± 11470	4671 ± 8359
2. MECH.-UNDERST.	10890 ± 7279	3325 ± 2887
3. STOCH.-INTERP.	-147 ± 409	-21 ± 2790
4. BBOX	13893 ± 7123	2506 ± 2762
5. RICE	13922 ± 12839	-635 ± 2810
6. LBCS	7571 ± 9259	5853 ± 5159
7. BAM	11690 ± 12350	65 ± 313
8. FTRL	6876 ± 11067	-2682 ± 2444
9. SELF-EXPANSION	-7743 ± 7479	-447 ± 2902
10. SEM.-SELF-CONS.	-5318 ± 7044	-1387 ± 2232
11. BRIDG.-DATA-GAPS	16212 ± 8012	7176 ± 4301
12. SELF-COMP.-POL.	10719 ± 5802	-155 ± 6786
13. ROBUST-CLIP	715 ± 9301	139 ± 7996
14. STAY-ON-TOPIC	8742 ± 13485	9088 ± 12263
15. SAPG	7587 ± 7848	155 ± 2722
16. SAM.-SPEC.-MASK.	15136 ± 30306	1036 ± 1843
17. ADAPT.-PRUN.	12954 ± 11170	6335 ± 14316
18. LCA-ON-THE-LINE	21111 ± 16865	4628 ± 1941
19. TEST-TIME	10076 ± 18291	-1210 ± 3040
20. WHAT-WILL-MODEL	-5471 ± 3307	-3906 ± 5537
21. ALL-IN-ONE	-3134 ± 5472	792 ± 6584
22. SEQ.-NEUR.-SCORE	13875 ± 14318	2595 ± 7054
AVERAGE	7763 ± 2595	1724 ± 568

Table 2: Average token budget improvement for prompt and competion with Congram compared to Zero-shot setting on Qwen3-Next. Papers are listed in order of processing. PINN paper is used to bootstrap continual memory and was not used for code generation. Numbers averaged across 3 runs. The positive improvements are shown in color.

the papers Codegram spent less tokens on coding. For prompt, Codegram used in average 7763 ± 2595 tokens less than Zero-shot generation, and for completion 1724 ± 568 less.

The results for Gemini-2.0-Flash demonstrate that even if Congram more frequently generates longer prompts that Zero-shot setting, it improves the completion token budget on more than half of the papers (12 of 22), spending on average 915 ± 2027 fewer completion tokens than Zero-shot generation. And for 7 out of 16 papers where Congram prompt took more tokens that Zero-shot, the completion token usage improved – which is important when using proprietary LLMs that usually have higher completion token cost rather than prompt.

PAPER	Δ PROMPT	Δ COMPL.
0. PINN	-	-
1. FRE	4972 ± 6382	3600 ± 1621
2. MECH.-UNDERST.	5009 ± 3617	4501 ± 766
3. STOCH.-INTERP.	-727 ± 875	90 ± 807
4. BBOX	6940 ± 3930	2709 ± 1962
5. RICE	-7486 ± 3569	87 ± 2022
6. LBCS	-3874 ± 2002	-1432 ± 2567
7. BAM	-9169 ± 7056	-502 ± 1040
8. FTRL	-5058 ± 1688	420 ± 1178
9. SELF-EXPANSION	-3622 ± 2100	1130 ± 734
10. SEM.-SELF-CONS.	-4037 ± 733	-267 ± 228
11. BRIDG.-DATA-GAPS	2969 ± 4323	4990 ± 1957
12. SELF-COMP.-POL.	-7596 ± 7191	3139 ± 2219
13. ROBUST-CLIP	-2697 ± 93	-1036 ± 974
14. STAY-ON-TOPIC	-2654 ± 2850	-737 ± 510
15. SAPG	1291 ± 2268	-1358 ± 4097
16. SAM.-SPEC.-MASK.	-2153 ± 6175	-244 ± 641
17. ADAPT.-PRUN.	10111 ± 12630	3192 ± 1559
18. LCA-ON-THE-LINE	-6357 ± 3124	-82 ± 2173
19. TEST-TIME	-4897 ± 3413	2757 ± 1902
20. WHAT-WILL-MODEL	-1290 ± 978	-869 ± 2436
21. ALL-IN-ONE	-6021 ± 3793	-1296 ± 1301
22. SEQ.-NEUR.-SCORE	-4208 ± 3900	1347 ± 1217
AVERAGE	-1843 ± 5086	915 ± 2027

Table 3: Average token budget improvement (for prompt and completion) with Congram compared to Zero-shot setting on Gemini-2.0-Flash. Papers are listed in order of processing during Congram experiment. PINN paper is used to bootstrap continual memory and was not used for code generation. Numbers averaged across 3 runs.The positive improvements are shown in color.

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

5 Conclusion

Our results support the thesis that replication agents benefit from continual structured memory, and also highlight he open problems of robust entity resolution across heterogeneous papers, improving retrieval faithfulness, and integrating execution feedback loops with structured memory updates.

Limitations

Our current evaluation is limited by (i) a small continual sequence, (ii) high research domain variety across tested papers, and (iii) the absence of end-to-end execution feedback in our graph-based pipeline comparable to IterativeAgent checks.

References

Petr Anokhin, Nikita Semenov, Artyom Sorokin, Dmitry Evseev, Andrey Kravchenko, Mikhail Burtsev, and Evgeny Burnaev. 2025. Arigraph: Learning knowledge graph world models with episodic memory for llm agents. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pages 12–20. International Joint Conferences on Artificial Intelligence Organization. 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. 2025. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*.

Google Deepmind. 2024. Introducing gemini 2.0: our new ai model for the agentic era. Google Deepmind Blog.

Bernal Jiménez Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. 2025. Hipporag: Neurobiologically inspired long-term memory for large language models. *Preprint*, arXiv: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. 2025. Retrieval-augmented generation with graphs (graphrag). *Preprint*, arXiv:2501.00309.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2024. MLAgentBench: Evaluating language agents on machine learning experimentation. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 20271–20309. PMLR.

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. 2022. Unsupervised dense information retrieval with contrastive learning. *Transactions on Machine Learning Research*.

Thomas Joshi, Shayan Chowdhury, and Fatih Uysal. 2025. Swe-bench-cl: Continual learning for coding agents. *Preprint*, arXiv: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. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*.

Lehui Li, Ruining Wang, Haochen Song, Yaoxin Mao, Tong Zhang, Yuyao Wang, Jiayi Fan, Yitong Zhang, Jieping Ye, Chengqi Zhang, and Yongshun Gong. 2026. What papers don't tell you: Recovering tacit knowledge for automated paper reproduction. *Preprint*, arXiv: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, and 18 others. 2026. The last human-written paper: Agent-native research artifacts. *Preprint*, arXiv:2604.24658.

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

OpenAI o3-mini. 2025. Openai o3-mini. OpenAI Blog.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Memgpt: Towards llms as operating systems. *Preprint*, arXiv:2310.08560.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. *Preprint*, arXiv:2304.03442.

Qwen Team. 2025. Qwen3-next: Towards ultimate training & inference efficiency. Qwen AI Blog.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Preprint*, arXiv:2303.11366.

Zachary S Siegel, Sayash Kapoor, Nitya Nadgir, Benedikt Stroebl, and Arvind Narayanan. 2024. CORE-bench: Fostering the credibility of published research through a computational reproducibility agent benchmark. *Transactions on Machine Learning Research*.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, and 1 others. 2025. Paperbench: Evaluating ai's ability to replicate ai research. *arXiv preprint arXiv:2504.01848*.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. A-mem: Agentic memory for llm agents. In *Advances in Neural Information Processing Systems*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

Daniil Zverev, A. Sophia Koepke, and Joao F. Henriques. 2025. On the dangers of bootstrapping generation for continual learning and beyond. *Preprint*, arXiv:2512.11867.

A Zero-shot code generation

Figure 2 demonstrates the pipeline used for Zero-shot 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[1]) (Qwen Team, 2025) and Gemini 2.0 Flash (Google Deepmind, 2024). 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.

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.

[1] 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 4: Relation vocabulary used by all graph extraction prompts. Relation families are used during card construction to derive a paper card's intent label.

C Ontology

The ontology full vocabulary is listed below in Table 4.

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 4 lists every label.

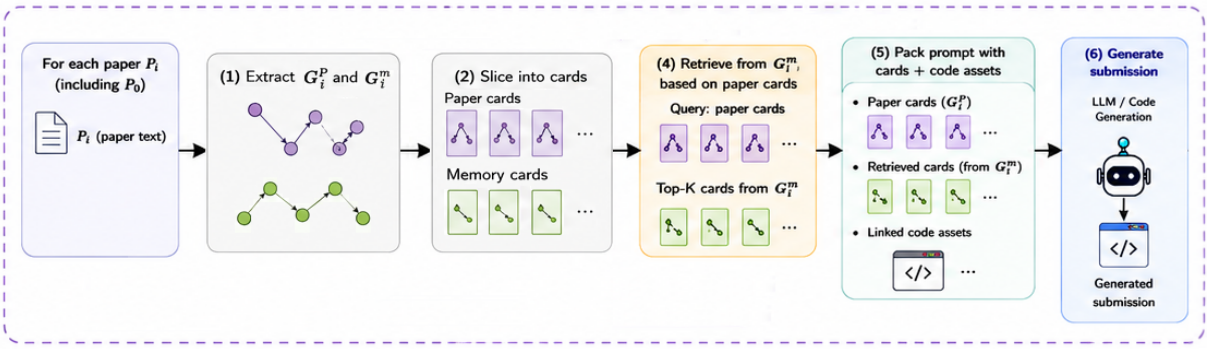

Figure 2: Zero-shot graph-based code generation pipeline. The paper and memory graph extraction and card slicing follow the main pipeline used for continual code generation. Mmeory graph triplets are linked with original paper repository code. Retrieval is performed over G_i^m memory cards. Code generation prompt follows the continual generation prompt template.

D Continual knowledge transfer examples

Figure 3 provides 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

(a) *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 near-verbatim 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)
```

(b) *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 3: Cross-paper transfer through the continual memory graph. For each pair, 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. (a) shows substantive method transfer (an interpolant-loss training step) and (b) a reusable training-loop scaffold.