What Makes AI Research Replicable?
Executable Knowledge Graphs as Scientific Knowledge Representations

Yujie Luo[♠♡*] , **Zhuoyun Yu**[♠*] , **Xuehai Wang**[♠], **Yuqi Zhu**[♠♡],
Ningyu Zhang[♠♡†] , **Lanning Wei**[♣♡], **Lun Du**[♣♡], **Da Zheng**[♣♡†] , **Huajun Chen**[♠♡†]
[♠]Zhejiang University [♣]Ant Group
[♡]Zhejiang University - Ant Group Joint Laboratory of Knowledge Graph
{luo.yj,zhangningyu}@zju.edu.cn

Abstract

Replicating AI research is a crucial yet challenging task for large language model (LLM) agents. Existing approaches often struggle to generate executable code, primarily due to insufficient background knowledge and the limitations of retrieval-augmented generation (RAG) methods, which fail to capture latent technical details hidden in referenced papers. Furthermore, previous approaches tend to overlook valuable implementation-level code signals and lack structured knowledge representations that support multi-granular retrieval and reuse. To overcome these challenges, we propose **Executable Knowledge Graphs (xKG)**, a pluggable, paper-centric knowledge base that automatically integrates code snippets and technical insights extracted from scientific literature. When integrated into three agent frameworks with two different LLMs, xKG shows substantial performance gains (10.9% with o3-mini) on PaperBench, demonstrating its effectiveness as a general and extensible solution for automated AI research replication[1].

1 Introduction

The rapid advancement of AI has dramatically accelerated scientific progress, producing thousands of new publications each year (Zhao et al., 2023). However, reproducing these results remains a major bottleneck: many papers omit critical implementation details, code repositories are incomplete or unavailable, and essential background knowledge is scattered across diverse sources (Zhao et al., 2025; Seo et al., 2025; Zhou et al., 2025; Edwards et al., 2025; Zhu et al., 2025; Huang et al., 2025; Zhu et al., 2025; Kon et al., 2025; Yan et al., 2025). While humans perform the tedious pipeline of reading papers, inspecting code, and collecting background materials to reproduce results, enabling machines to perform the same workflow reliably remains an open challenge (Chen et al., 2025).

Why Executable Knowledge Graphs? AI research is hard to replicate and reuse because its knowledge is implicit and fragmented across text, code, and configuration. To address this challenge, we propose the **Executable Knowledge Graph (xKG)**, a novel, paper-centric knowledge base designed to externalize latent scientific knowledge into a verifiable, executable representation.

Our xKG overcomes the limitations of existing attempts (Tang et al., 2025; Ou et al., 2025), which often stop at coarse-grained knowledge reuse, by modeling scientific literature as hierarchical graphs that ground academic concepts in executable code. Unlike conventional KGs, xKG captures both conceptual relations and executable components, enabling agents to assemble the precise artifacts needed for faithful reproduction. We evaluate xKG by integrating it into three distinct agent frameworks: BasicAgent, IterativeAgent, and PaperCoder. Experiments on PaperBench (Starace et al., 2025) demonstrate consistent and significant performance gains. Our xKG is built on a fully automated paper-aware pipeline, updatable to stay current with advancing research.

2 Executable Knowledge Graphs

2.1 Design Formulation

We model xKG as a hierarchical, multi-relational graph $\text{xKG} = (\mathcal{N}, \mathcal{E})$, specifically:

$$\mathcal{N} = \mathcal{N}_P \cup \mathcal{N}_T \cup \mathcal{N}_C \quad (1)$$

$$\mathcal{E} = \mathcal{E}_{\text{struct}} \cup \mathcal{E}_{\text{impl}} \quad (2)$$

We define three types of nodes to capture knowledge at different granularities:

- **Paper Node (n_p):** Represents a paper as a tuple $n_p = (M_p, \{n_t\}_i, \{n_c\}_j)$, containing metadata M_p (e.g., abstracts, references, etc.), technique nodes $\{n_t\}_i$, and code nodes $\{n_c\}_j$.

[*] Equal Contribution.
[†] Corresponding Authors.
[1] https://github.com/zjunlp/xKG.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), pages 841–861
July 2-7, 2026

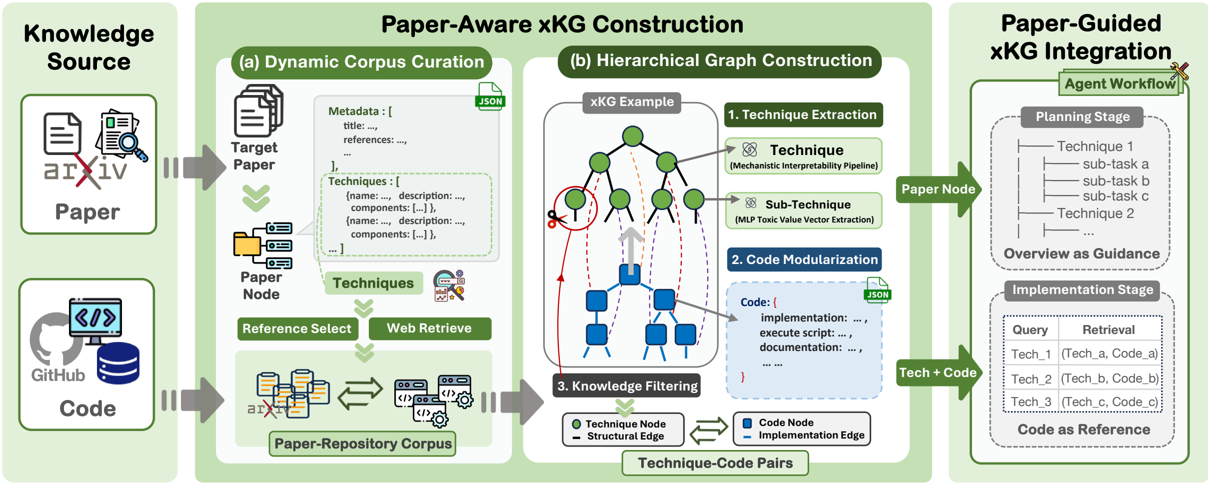

Figure 1: **The Paper-Centric xKG Pipeline. (1) Construction**: A paper-aware pipeline first curates a corpus of relevant papers and repositories, then extracts, modularizes, and filters it into executable technique-code pairs. **(2) Integration**: Our xKG guides agents by providing high-level technique overviews and low-level code references.

- **Technique Node (n_t):** A self-contained academic concept $n_t = (D_t, \{n'_t\}_k)$ with definition D_t and optional sub-nodes $\{n'_t\}_k$, ranging from complete framework to reusable component.
- **Code Node (n_c):** An executable unit $n_c = (\sigma, \tau, \delta)$ comprising code implementation σ, a test script τ, and documentation δ.

These nodes are linked by the following edges:

- **Structural Edge (e_{struct}):** An edge $(n_{t,i}, n_{t,j})$ indicates an architectural dependency between technique nodes.
- **Implementation Edge (e_{impl}):** An edge (n_t, n_c) linking a technique to its code implementation.

Our xKG directly links scientific concepts to executable code, yielding more comprehensive knowledge (example in Figure 1(b), Appendix D).

2.2 Paper-Aware xKG Construction

2.2.1 Dynamic Corpus Curation

Our corpus curation is a paper-aware automated pipeline engineered for continuous scalability and cross-domain extensibility, as shown in Figure 1(a). For each paper targeted for reproduction (papers in PaperBench (Starace et al., 2025)), xKG automates the collection of all prerequisite resources, beginning by employing o4-mini to identify its core *techniques*. Centered on these techniques, we first cull its top-ranked references via LLM, and then execute an automated technique-based web retrieval, culminating in a curated set of ten relevant papers for each. **Note that we strictly do NOT use any GitHub or third-party reproduction repositories listed in PaperBench's blacklist to avoid any risk of data leakage.** All retrieved papers are processed to fetch their LaTeX sources from arXiv and then identify the associated GitHub. Papers without official repositories are automatically filtered out, resulting in a corpus of **paper-repository pairs**.

2.2.2 Hierarchical KG Construction

Based on the corpus, we then construct the xKG with the automated steps shown in Figure 1(b):

- **Step 1: Technique Extraction.** We first use o4-mini to deconstruct the paper's decomposable methodology into a preliminary hierarchical tree of Technique Nodes $\mathcal{N}_T$ linked by Structural Edges e_{struct}(details in Appendix B.3). Subsequently, we utilize RAG[2] to enrich each node by retrieving relevant text from the paper to form a comprehensive definition D_t. This step yields a set of detailed techniques yet may contain noise.

- **Step 2: Code Modularization.** For each technique n_t, its definition is used as a query to retrieve code snippets from the repository via embedding similarity. We then employ o4-mini to synthesize these snippets into candidate Code Nodes n_c, each includes the implementation σ, test script τ, and documentation δ. These nodes then undergo an iterative *self-debugging* loop to ensure executability, ultimately producing a set of **executable Code Nodes** $\mathcal{N}_c$ with associated technique-linking Implementation Edges e_{impl}.

[2] We employ text-embedding-3-small for embedding similarity throughout all stages of xKG construction.

Method	Model	MU-DPO		TTA-FP		One-SBI		CFG		FRE		Average	
		vanilla	+xKG	vanilla	+xKG	vanilla	+xKG	vanilla	+xKG	vanilla	+xKG	vanilla	+xKG
BasicAgent	o3-mini	12.96	37.22$_{+24.26}$	22.63	27.26$_{+4.63}$	18.24	20.82$_{+2.58}$	20.82	22.86$_{+2.04}$	14.82	14.67$_{-0.15}$	17.89	24.57 ↑**6.68**
	DS-R1	33.05	39.14$_{+6.09}$	40.55	39.14$_{-1.41}$	17.22	24.49$_{+7.27}$	31.56	33.97$_{+2.41}$	17.08	21.38$_{+4.30}$	27.89	31.62 ↑**3.73**
IterativeAgent	o3-mini	22.22	43.70$_{+21.48}$	21.38	36.28$_{+14.90}$	28.77	23.91$_{-4.86}$	31.28	29.15$_{-2.13}$	19.35	26.50$_{+7.15}$	24.60	31.91 ↑**7.31**
	DS-R1	16.20	47.40$_{+31.20}$	31.19	31.78$_{+0.59}$	31.09	26.57$_{-4.52}$	35.30	38.44$_{+3.14}$	21.32	31.89$_{+10.57}$	27.02	35.22 ↑**8.20**
PaperCoder	o3-mini	23.15	46.48$_{+23.33}$	45.70	53.99$_{+8.29}$	52.48	52.08$_{-0.40}$	50.37	63.13$_{+12.76}$	39.84	50.36$_{+10.52}$	42.31	53.21 ↑**10.90**
	DS-R1	43.24	49.26$_{+6.02}$	43.26	59.19$_{+15.93}$	51.18	73.03$_{+21.85}$	61.12	60.68$_{-0.44}$	62.37	59.53$_{-2.84}$	52.23	60.34 ↑**8.11**

Table 1: **Main results on PaperBench Code-Dev.** We evaluate on the official `lite` subset of PaperBench (details in Table 9). All results are the `best@3` *Replication Score* (%) to mitigate task stochasticity and potential tool failures.

- **Step 3: Knowledge Filtering.** We formalize a simple yet powerful verification principle: a technique n_t is considered valuable only if it can be grounded in executable code. Therefore, any technique for which **Step 2** failed to retrieve relevant code snippets is pruned from the xKG. This filtering process ensures that only techniques with proven, practical value populate the final xKG, eliminating the noise and overly granular nodes introduced in **Step 1**.

Finally, we construct xKG from 42 curated papers, totaling 591,145 tokens. **Our xKG construction is built upon a paper-aware pipeline designed for continuous evolution as new literature emerges**, as detailed in Appendix B.1.

2.3 Paper-Guided xKG Integration

xKG can be seamlessly integrated into a practical reproduction workflow, where LLM agents utilize it at two stages (Figure 1, right). For **high-level planning**, the agent fetches the target paper's Paper Node (without Code Nodes) to grasp its core techniques and overall structure. During **low-level implementation**, the agent queries xKG for (Technique, Code) pairs directly relevant to the target paper. These two steps can be supplied either as callable tools for ReAct agents or as pluggable components of fixed-workflow agents. Crucially, all retrieved candidates are processed by a final **LLM-based Verifier** (o4-mini), acting as a final quality gate to ensure the retrieved knowledge is both technically relevant and implementable.

3 Experiments

3.1 Settings

We evaluate xKG on the lite collection of PaperBench Code-Dev (Starace et al., 2025), a benchmark for repository-level paper reproduction from scratch, scored by an o3-mini-based evaluator using a weighted, tree-structured rubric. We integrate xKG into BasicAgent (a ReAct-style agent), IterativeAgent (adds a self-improvement loop), both with a one-hour runtime limit, and PaperCoder(a powerful agent tailored for repository-level paper reproduction.). See Appendix A for more details.

3.2 Main Results

As shown in Table 1, xKG achieves substantial performance gains across diverse agent frameworks and LLM backbones. On the general ReAct-style IterativeAgent with DeepSeek-R1, xKG delivers a performance improvement of ↑8.20%. The effectiveness of xKG is further highlighted by the ↑10.90% improvement achieved with PaperCoder powered with o3-mini, underscoring its broad applicability from simpler agents to more advanced ones. Notably, the impact of xKG is also highly paper-dependent. While BasicAgent with o3-mini achieves a remarkable 24.26% performance gain on MU-DPO, the same configuration yields only a 2.58% improvement on One-SBI and even a 0.15% drop on the FRE task, revealing a critical dependency on the target paper (details in Appendix C).

3.3 Further Analysis

Method	Score (%)	Drop (▽)
xKG(Full)	53.21	-
w/o Paper Node	51.08	2.13
w/o Code Node	48.65	4.56
w/o Technique Node	52.16	1.05

Table 2: Ablation study on xKG node types.

Code-based structured knowledge aids AI research replication. As shown in Table 2, our ablation study conducted on PaperCoder with o3-mini

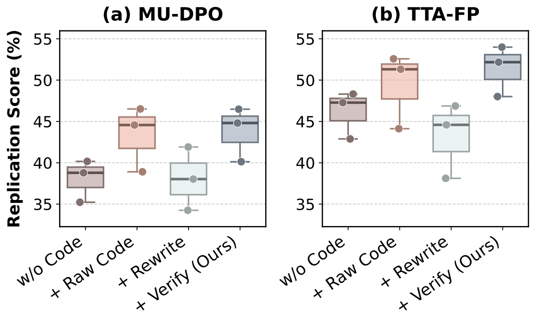

Figure 2: Further study on Code Node quality.

	Techs.	Codes	Tech-Code Pairs
Valid Rate (%)	89.44	100.00	74.51

Table 3: Human evaluation of xKG quality.

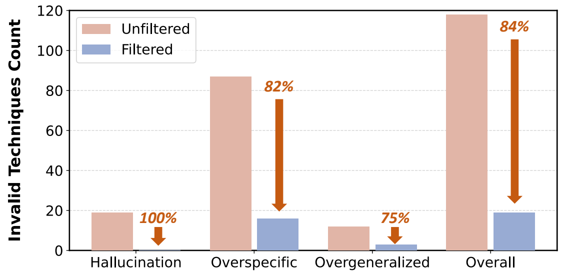

Figure 3: Comparison of invalid Technique Nodes in xKG before and after knowledge filtering.

reveals that removing any component degrades performance. The most significant drop occurs when removing **Code Nodes**, decreasing the score by 4.56% (53.21% → 48.65%), suggesting that LLM agents benefit immensely from fine-grained knowledge, with executable code being the most critical component. Ablating **Paper Nodes** yields a substantial degradation of 2.13%, highlighting the value of a structural overview of the target paper. In contrast, omitting **Technique Nodes** results in a modest 1.05% drop, since the function of each technique is already captured by the Code Nodes, rendering the explicit description redundant.

Successful reproduction hinges on retrieved code quality. Building on the above, we conduct a further analysis of how Code Nodes influence performance. We analyze the impact of code quality on PaperCoder (o3-mini), comparing four settings with 3 runs each: w/o Code (no code), + Raw Code (raw snippets), + Rewrite (LLM-rewritten but unverified), and + Verify (LLM-rewritten & verified).

As illustrated in Figure 2, our full method excels in score and stability. Notably, even incorporating raw code snippets (+ *Raw Code*) significantly

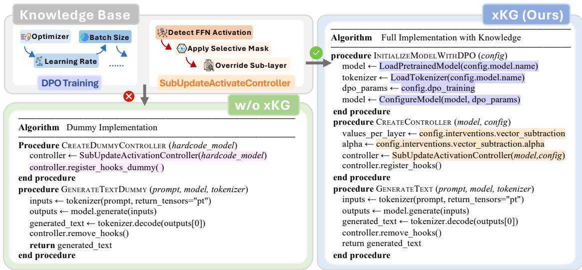

Figure 4: Case Study on MU-DPO. Highlights illustrating agent implementation contrast with/without xKG.

improves performance, validating that our method effectively localizes necessary code. The failure of + *Rewrite* ablation reveals a key insight: **agents are misled by well-formatted, semantically similar yet technically irrelevant knowledge**, a phenomenon consistent with previous findings (Wu et al., 2024). Our LLM Verifier mitigates this by prioritizing technical relevance over semantic similarity, filtering out distracting irrelevant information, thus boosting the score improvement varying from 5.75 to 8.20 percentage points.

Automated Construction architects robust xKG. Human evaluation (Table 3) confirms the high quality of xKG with 89.44% of Techniques representing self-contained academic concepts, all Code Nodes executable, and 74.51% of Tech-Code pairs exactly matched. During xKG construction, the Knowledge Filtering step mitigates LLM errors (e.g., overspecificity, hallucinations) by pruning techniques with no code retrieved (Figure 3). Concurrently, a self-debugging loop achieves 100% code executability, up from an initial 52.38%. Tech-Code mismatches primarily stem from broader code snippets with little impact on application. See Appendix B for more details of constructed xKG.

xKG Transforms Agents from Scaffolding to Implementation. To understand the mechanism behind the performance gains, we conduct a case study on the MU-DPO paper (Figure 4). We notice that xKG enriches information granularity, allowing agents to generate critical details accurately, and improves modular implementation capability, enabling agents to reuse code for functionally correct implementations, as highlighted in Figure 4. This case reveals that xKG transforms agents from dummy scaffolders to substantive implementers, arming them with both the precise method details for accurate planning and the verified, modular reference code for robust implementation.

4 Conclusion

We introduce Executable Knowledge Graphs (xKG) to make implicit research knowledge modular and executable, boosting agent replication performance. Looking forward, we envision xKG serving not only as a dynamic knowledge base, but also as a flexible instrument to accelerate AI Research with improved efficiency and verifiability.

Limitations

This work has several limitations. First, the PaperBench task exhibits high variance and is costly to evaluate. Although we report results across multiple papers and conduct experiments, due to funding constraints, we only perform experiments on the `lite` collection of PaperBench Code-Dev. Second, for emerging domains, there may be no available reference papers at all, which limits the applicability of our approach to scenarios where some baseline references exist. Finally, while the code-based knowledge organization we propose may have the potential to transfer to similar tasks, exploring this remains future work (Nathani et al., 2025; Chan et al., 2024; Toledo et al., 2025; Jia et al., 2025; Miao et al., 2025).

During our work, we found another project with a similar name, ExeKG (Zheng et al., 2022b,a; Zhou et al., 2022). However, our approach differs fundamentally in the organization of the knowledge base — we adopt a much simpler structure of nodes and edges. Moreover, the problems addressed are entirely distinct: our focus is on paper replication tasks. We hold deep respect for the pioneering efforts of the ExeKG authors.

Acknowledgement

We would like to express sincere gratitude to the reviewers for their thoughtful and constructive feedback. This work was supported by the National Natural Science Foundation of China (No. 62576307, No. NSFCU23B2055, No. NSFCU19B2027), the Fundamental Research Funds for the Central Universities (226-2023-00138), Yongjiang Talent Introduction Programme (2021A-156-G), and Information Technology Center and State Key Lab of CAD&CG, Zhejiang University. This work was supported by Ant Group and Zhejiang University - Ant Group Joint Laboratory of Knowledge Graph.

References

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. 2024. Mle-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*.

Qiguang Chen, Mingda Yang, Libo Qin, Jinhao Liu, Zheng Yan, Jiannan Guan, Dengyun Peng, Yiyan Ji, Hanjing Li, Mengkang Hu, et al. 2025. Ai4research: A survey of artificial intelligence for scientific research. *arXiv preprint arXiv:2507.01903*.

Nicholas Edwards, Yukyung Lee, Yujun Audrey Mao, Yulu Qin, Sebastian Schuster, and Najoung Kim. 2025. Rexbench: Can coding agents autonomously implement ai research extensions? *arXiv preprint arXiv:2506.22598*.

Kevin Frans, Seohong Park, Pieter Abbeel, and Sergey Levine. 2024. Unsupervised zero-shot reinforcement learning via functional reward encodings. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Manuel Glöckler, Michael Deistler, Christian Dietrich Weilbach, Frank Wood, and Jakob H. Macke. 2024. All-in-one simulation-based inference. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Yuxuan Huang, Yihang Chen, Haozheng Zhang, Kang Li, Huichi Zhou, Meng Fang, Linyi Yang, Xiaoguang Li, Lifeng Shang, Songcen Xu, et al. 2025. Deep research agents: A systematic examination and roadmap. *arXiv preprint arXiv:2506.18096*.

Hangyi Jia, Yuxi Qian, Hanwen Tong, Xinhui Wu, Lin Chen, and Feng Wei. 2025. Towards adaptive ml benchmarks: Web-agent-driven construction, domain expansion, and metric optimization. *arXiv preprint arXiv:2509.09321*.

Patrick Tser Jern Kon, Jiachen Liu, Xinyi Zhu, Qiuyi Ding, Jingjia Peng, Jiarong Xing, Yibo Huang, Yiming Qiu, Jayanth Srinivasa, Myungjin Lee, et al. 2025. Exp-bench: Can ai conduct ai research experiments? *arXiv preprint arXiv:2505.24785*.

Andrew Lee, Xiaoyan Bai, Itamar Pres, Martin Wattenberg, Jonathan K. Kummerfeld, and Rada Mihalcea. 2024. A mechanistic understanding of alignment algorithms: A case study on DPO and toxicity. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Jiacheng Miao, Joe R Davis, Jonathan K Pritchard, and James Zou. 2025. Paper2agent: Reimagining research papers as interactive and reliable ai agents. *arXiv preprint arXiv:2509.06917*.

Deepak Nathani, Lovish Madaan, Nicholas Roberts, Nikolay Bashlykov, Ajay Menon, Vincent Moens, Amar Budhiraja, Despoina Magka, Vladislav Vorotilov, Gaurav Chaurasia, et al. 2025. Mlgym: A new framework and benchmark for advancing ai research agents. *arXiv preprint arXiv:2502.14499*.

Shuaicheng Niu, Chunyan Miao, Guohao Chen, Pengcheng Wu, and Peilin Zhao. 2024. Test-time model adaptation with only forward passes. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Yixin Ou, Yujie Luo, Jingsheng Zheng, Lanning Wei, Shuofei Qiao, Jintian Zhang, Da Zheng, Huajun Chen, and Ningyu Zhang. 2025. Automind: Adaptive knowledgeable agent for automated data science. *arXiv preprint arXiv:2506.10974*.

Guillaume Sanchez, Alexander Spangher, Honglu Fan, Elad Levi, and Stella Biderman. 2024. Stay on topic with classifier-free guidance. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. 2025. Paper2code: Automating code generation from scientific papers in machine learning. *arXiv preprint arXiv:2504.17192*.

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

Jiabin Tang, Lianghao Xia, Zhonghang Li, and Chao Huang. 2025. Ai-researcher: Autonomous scientific innovation. *CoRR*, abs/2505.18705.

Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Rishi Hazra, Nicolas Baldwin, Alexis Audran-Reiss, Michael Kuchnik, Despoina Magka, Minqi Jiang, Alisia Maria Lupidi, et al. 2025. Ai research agents for machine learning: Search, exploration, and generalization in mle-bench. *arXiv preprint arXiv:2507.02554*.

Siye Wu, Jian Xie, Jiangjie Chen, Tinghui Zhu, Kai Zhang, and Yanghua Xiao. 2024. How easily do irrelevant inputs skew the responses of large language models? *CoRR*, abs/2404.03302.

Shuo Yan, Ruochen Li, Ziming Luo, Zimu Wang, Daoyang Li, Liqiang Jing, Kaiyu He, Peilin Wu, George Michalopoulos, Yue Zhang, et al. 2025. Lmr-bench: Evaluating llm agent's ability on reproducing language modeling research. *arXiv preprint arXiv:2506.17335*.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2).

Xuanle Zhao, Zilin Sang, Yuxuan Li, Qi Shi, Weilun Zhao, Shuo Wang, Duzhen Zhang, Xu Han, Zhiyuan Liu, and Maosong Sun. 2025. Autoreproduce: Automatic ai experiment reproduction with paper lineage. *arXiv preprint arXiv:2505.20662*.

Zhuoxun Zheng, Baifan Zhou, Dongzhuoran Zhou, Ahmet Soylu, and Evgeny Kharlamov. 2022a. Executable knowledge graph for transparent machine learning in welding monitoring at bosch. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 5102–5103.

Zhuoxun Zheng, Baifan Zhou, Dongzhuoran Zhou, Ahmet Soylu, and Evgeny Kharlamov. 2022b. Exekg: Executable knowledge graph system for user-friendly data analytics. In *Proceedings of the 31st ACM international conference on information & knowledge management*, pages 5064–5068.

Dongzhuoran Zhou, Baifan Zhou, Zhuoxun Zheng, Zhipeng Tan, Egor V Kostylev, and Evgeny Kharlamov. 2022. Towards executable knowledge graph translation. In *ISWC (Posters/Demos/Industry)*.

Mingyang Zhou, Quanming Yao, Lun Du, Lanning Wei, and Da Zheng. 2025. Reflective paper-to-code reproduction enabled by fine-grained verification. *arXiv preprint arXiv:2508.16671*.

Minjun Zhu, Qiujie Xie, Yixuan Weng, Jian Wu, Zhen Lin, Linyi Yang, and Yue Zhang. 2025. Ai scientists fail without strong implementation capability. *arXiv preprint arXiv:2506.01372*.

A Experimental Details

A.1 Benchmarks

The original PaperBench benchmark (Starace et al., 2025), featuring publicly available tasks and evaluation rubrics, is designed to evaluate the ability of AI agents to reproduce AI research from scratch.

As full-scale evaluation is both computationally expensive and time-consuming, the authors introduced a lightweight variant, PaperBench Code-Dev, which focuses solely on code development—assessing implementation correctness without requiring code execution or result verification.

In our study, we adopt the pre-defined lite subset of PaperBench Code-Dev provided in the official repository, spanning diverse AI Research domains such as machine learning, reinforcement learning, and natural language processing. We further analyze and categorize the research domains and related techniques involved in the target papers, as detailed in Table 8.

Evaluation follows a structured hierarchical rubric co-developed with the original authors, and an LLM-based evaluator (`o3-mini`) aggregates the final scores using a weighted binary criteria tree. Specific details about the papers and their evaluation nodes are listed in Table 9.

Furthermore, since PaperBench shows that BASICAGENT and ITERATIVEAGENT achieve little performance improvement beyond one hour, we cap their execution time at one hour for efficiency and cost control.

A.2 Configuration

The configuration of our xKG framework comprises both hyperparameters and prompts. The hyperparameters are managed via a central `config.yaml` file, which is organized into modules for Code-RAG, Paper-RAG, and Knowledge Graph Retrieval. We summarize the key parameters for each module in Tables 11-14. In addition, the specific prompts designed in our system are detailed in Appendix E.

A.3 Cost Evaluation

Our xKG construction pipeline leverages OpenAI's o4-mini via third-party API, chosen for its strong cost-effectiveness and robust text/code capabilities. We quantify the cost of this process in Table 10.

As shown, for an average cost of about \$0.73 per paper, our pipeline transforms a scientific publication into an executable knowledge graph. The most significant cost driver is the iterative *self-debugging* loop of the Code Modularization stage, simultaneously serving as the critical quality-assurance mechanism for our code nodes. The key advantage of xKG lies in this one-time investment: **each processed paper is converted into a durable and reusable knowledge resource**.

B Analysis of constructed xKG

B.1 Self-Evolution of xKG

xKG is not a static knowledge base but a flexible framework that automates the collection, extraction, and validation of knowledge in a paper-aware way. To illustrate this, we select two additional target papers from PaperBench, *bridging-data-gaps* and *sample-specific-masks*, to demonstrate the dynamic evolution of xKG.

The expansion from a 42-paper corpus (Table 4) to a 56-paper version (Table 5), curated from publicly available research, highlights the autonomous evolution of xKG. From the specific replication scores on these tasks (Table 6), we can further observe that xKG maintains sustained performance gains on newly introduced paper replication tasks.

When new target papers are introduced, our Corpus Curation module automatically gathers relevant literature and completes the full construction and verification cycle. This adaptability substantiates our claim that xKG is an adaptive system, effectively handling scenarios like fetching updated research or leveraging broader domain knowledge.

B.2 Comparison to Human-Constructed KG

To further validate the quality of our automated xKG paper deconstruction, we manually construct KGs for several papers and compared them against the nodes generated by xKG. Recognizing that manual annotation is a time-consuming process requiring significant domain expertise, we randomly selected three papers from the xKG corpus and invited expert PhD candidates to annotate them.

As shown in Table 7, the high weighted F1-scores demonstrate that our automated KG construction closely aligns with human annotation. Crucially, manually annotating a single paper takes 30-60 minutes, which highlights the significant efficiency gains of our automated pipeline.

B.3 Analysis of Paper Represent Granularity

Our hierarchical graph design (Section 2.1) is intentionally flexible, built to preserve a paper's natural methodological structure rather than forcing a uniform method decomposition. While most papers (71%) in xKG are decomposed into 2-5 techniques, a significant portion of papers are represented as single, atomic nodes (12%) or complex works with 6+ nodes (17%), ensuring that potentially useful information from high-level methodologies to fine-grained details is captured at its appropriate level.

C Analysis on Target Paper

As illustrated in Figure 5, the effectiveness of xKG is highly contingent on the target paper, with performance occasionally degrading. Bad cases stem from two primary failure modes: (1) **Over-reliance on retrieved code**, where the agent prioritizes generic snippets over the paper's unique implementation details; and (2) **Over-focus on core components**, where excelling at core techniques highlighted by xKG leads to the neglect of secondary objectives.

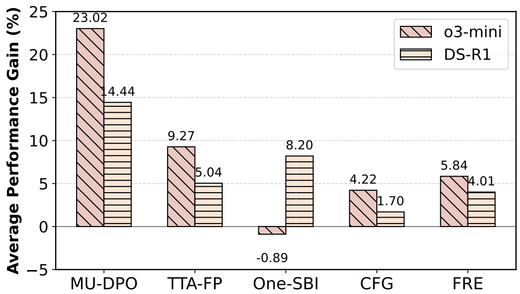

Figure 5: Average performance gain per paper.

More fundamentally, this performance disparity is tied to the paper's research archetype. *analytical papers*, such as MU-DPO(Lee et al., 2024), which synthesize and refine existing techniques, benefit substantially as their components are well-represented in xKG. Conversely, *methodological papers* like One-SBI(Glöckler et al., 2024), which introduce fundamentally novel architectures, find less directly applicable knowledge, as their core innovations have limited precedent in the corpus. This outcome is logical, as the performance bottleneck shifts from *knowledge augmentation* to the intrinsic *innovative capacity* of the base LLM itself.

Domain	Subfield	Count
Generative AI	Diffusion Models, Distribution Modeling, Controllable Generation	14
AI Safety & Robustness	Test-Time Adaptation, Continual Learning, Toxicity Detection	13
Simulation-Based Inference	Inverse Inference, Bayesian Inference, Scientific Computing	8
Reinforcement Learning	Exploration Strategies, Agent Decision-making, Policy Optimization	7
Mechanistic Interpretability	Circuit Analysis, Feature Visualization, Geometric Analysis	7

Table 4: Domains of the original 42 papers in xKG. The sum of counts exceeds the total number of papers due to interdisciplinary classifications.

Domain	Subfield	Count
Generative AI	Diffusion Models, Distribution Modeling, Controllable Generation	18
AI Safety & Robustness	Test-Time Adaptation, Continual Learning, Toxicity Detection	15
Simulation-Based Inference	Inverse Inference, Bayesian Inference, Scientific Computing	8
Mechanistic Interpretability	Circuit Analysis, Feature Visualization, Geometric Analysis	8
Reinforcement Learning	Exploration Strategies, Agent Decision-making, Policy Optimization	7
Model Adaptation & Efficiency	Transfer Learning, Visual Prompting, Model Reprogramming	4

Table 5: Updated domains of the 56 papers in the expanded xKG+, where some papers span multiple domains.

Setting	Method	Model	bridging-data-gaps	sample-specific-masks
w/o xKG	BasicAgent	`o3-mini`	11.55	24.09
xKG (42 papers)	BasicAgent	`o3-mini`	13.93	31.73
xKG+ (56 papers)	BasicAgent	`o3-mini`	44.64	42.47

Table 6: Replication scores (%) under different settings on the *bridging-data-gaps* and *sample-specific-masks* tasks.

Paper	HumanKG Nodes	xKG Nodes	Precision (w)	Recall (w)	F1-Score (w)
CFDG	4	3	1.00	0.90	0.95
RND	11	7	0.92	0.86	0.89
TENT	4	4	1.00	1.00	1.00

Table 7: Comparison of automatically-constructed xKG and human-constructed KG.

Paper	Domain	Subfield
FRE	Reinforcement Learning	Zero-Shot RL, Unsupervised Pre-training, Functional Reward Encoding, Transformer-VAE, Offline RL
TTA-FP	Efficient Machine Learning	Test-Time Adaptation, Backpropagation-Free Optimization, CMA-ES, Prompt Tuning, Quantized Model Deployment
MU-DPO	AI Safety & Alignment	Mechanistic Interpretability, Direct Preference Optimization, Toxicity Reduction, Activation Intervention, SVD Analysis
One-SBI	Scientific Machine Learning	Simulation-Based Inference, Amortized Bayesian Inference, Diffusion Models, Transformer with Structured Attention, Score-Matching
CFG	Natural Language Processing	Controllable Text Generation

Table 8: Domains and subfields of the PaperBench tasks evaluated in this work.

Our Abbr.	PaperBench Name	CodeDev Nodes
FRE (Frans et al., 2024)	*fre*	306
TTA-FP (Niu et al., 2024)	*test-time-model-adaptation*	86
MU-DPO (Lee et al., 2024)	*mechanistic-understanding*	36
One-SBI (Glöckler et al., 2024)	*all-in-one*	92
CFG (Sanchez et al., 2024)	*stay-on-topic-with-classifier-free-guidance*	70

Table 9: Abbreviations for the PaperBench tasks evaluated in this work.

Stage	LLM	API Calls (avg)	Input Tokens (avg)	Output Tokens (avg)	Cost (avg, $)
Technique Extraction	o4-mini	8.21	53,142.36	11,523.43	0.1092
Code Modularization	o4-mini	33.36	202,625.21	91,426.14	0.6252
Knowledge Filtering	—	—	—	—	—
Total	o4-mini	41.57	255,767.57	102,949.57	0.7344

Table 10: Average cost analysis for constructing the xKG for a single paper.

Hyperparameter	Value	Description
Code-RAG Module		
`code.embedder.model`	`text-embedding-3-small`	The embedding model used for code chunk vectorization.
`code.text_splitter.chunk_size`	350	The size of each text chunk when splitting code files.
`code.text_splitter.chunk_overlap`	100	The number of overlapping characters between adjacent chunks.
`code.retriever.faiss.top_k`	10	Number of initial candidate chunks retrieved via FAISS vector search.
`code.retriever.llm.top_files`	5	Number of top files selected by the LLM reranker for detailed analysis.
`code.exec_check_code`	False	A boolean flag to enable or disable the execution-based verification of generated code.

Table 11: Hyperparameters for the Code-RAG module in xKG.

Hyperparameter	Value	Description
Paper-RAG Module		
`paper.rag`	True	A boolean flag to enable or disable the entire Paper-RAG process.
`paper.embedder.model`	`text-embedding-3-small`	The embedding model used for paper text vectorization.
`paper.text_splitter.chunk_size`	350	The size of each text chunk when splitting the paper content.
`paper.retriever.faiss.top_k`	5	Number of relevant text excerpts retrieved from the paper via FAISS.

Table 12: Hyperparameters for the Paper-RAG module in xKG.

Hyperparameter	Value	Description
Knowledge Graph Retrieval		
`retrieve.embedding_model`	`all-MiniLM-L6-v2`	The sentence-transformer model used for calculating similarity between techniques.
`retrieve.technique_similarity`	0.6	The minimum similarity score required for a technique to be retrieved from the KG.
`retrieve.paper_similarity`	0.6	The minimum similarity score required for a paper to be retrieved from the KG.

Table 13: Hyperparameters for Knowledge Graph retrieval.

Hyperparameter	Value	Description
Global & Model Profile Configuration		
`log_level`	DEBUG	Sets the verbosity of logging.
`kg_path`	`storage/kg`	The directory where the constructed Knowledge Graph is stored.
`max_prompt_code_bytes`	52100	The maximum size in bytes for code content included in a prompt to the LLM.
`model`	`DeepSeek-V3`	The primary foundation model for the agent's base tasks.
`paper_model`	`o4-mini`	A specialized model used specifically for extracting and rewriting techniques from papers.
`code_model`	`o4-mini`	A specialized model used for rewriting and debugging code.

Table 14: Common global settings and an example model profile (`basic-deepseek-v3`). Specific models can be defined for different sub-tasks, allowing for flexible and optimized model selection.

D Running Examples of xKG


```json
{
    "paper_title": "Discovering Latent Knowledge in Language Models Without Supervision",
    "paper_abstract": "Existing techniques for training language models can be misaligned with the truth: if we train mod
    "paper_references": [
      "A general language assistant as a laboratory for alignment",
      "... (Omit the rest)"
    ],
    "findings": [
      "CCS Outperforms Zero-Shot Prompting: Empirically, CCS exceeds the performance of strong zero-shot prompting baseli
      "... (Omit the rest)"
    ],
    "techniques": [
      {
        "name": "Contrast-Consistent Search (CCS)",
        "description": "Contrast-Consistent Search (CCS) is an end-to-end, unsupervised procedure for extracting a latent
        "code": {
          "implementation": "import numpy as np\nimport torch\nimport torch.nn as nn\nfrom transformers import AutoModel,
          "test": "if __name__ == \"__main__\":\n    # Synthetic demonstration of CCS on random features\n    np.random.s
          "documentation": "This module implements Contrast-Consistent Search (CCS), an unsupervised method to extract a
          "package": [
            "numpy",
            "torch",
            "transformers"
          ]
        },
        "components": [
          {
            "name": "Construction of Contrast Pairs",
            "description": "For each yes-no question, we construct a contrast pair (x+, x-) by formatting the question in
            "code": {
              "implementation": "import torch\nfrom transformers import AutoTokenizer\n\nclass ContrastPairConstructor:\n
              "test": "if __name__ == \"__main__\":\n    # Example usage with T5-small tokenizer\n    tokenizer = AutoTok
              "documentation": "This module defines a ContrastPairConstructor that formats binary questions into positive
              "package": [
                "torch",
                "transformers"
              ]
            },
            "components": []
          },
          {
            "name": "Feature Extraction and Normalization",
            "description": "In this approach, we compute the hidden representations of each contrast example using a pret
            "code": {
              "implementation": "import numpy as np\nimport torch\nfrom transformers import AutoModel, AutoTokenizer\n\nd
              "test": "if __name__ == \"__main__\":\n    # Example usage with BERT encoder-only model\n    model_name = \
              "documentation": "This module implements the unsupervised feature extraction and normalization technique fo
              "package": [
                "numpy",
                "torch",
                "transformers"
              ]
            },
            "components": []
          },
          // ... (Omit)
        ]
      }
    ]
  }
```


Figure 6: An example of structural xKG data storage. Paper Nodes are stored as JSON files, with technique and code nodes embedded as structured dictionaries, where key-value pairs are used to create a one-to-one mapping representing the implementation relationship.

E Prompts

In this section, we showcase some of the key prompts used in the full pipeline of our system, which serve as a reference. The prompts are organized by their functional role in the pipeline: paper parsing, code repository parsing, and knowledge graph construction.

E.1 Paper Parsing

Prompt for Extracting References from .bbl File

Task
You are provided with a .bbl file {bbl}. Please extract the titles of all the references in the .bbl file.

Output
1. Output the extracted reference titles in the form of a string list.
2. If no reference is available, please return None.

Please wrap your final answer between two ``` in the end.

Prompt for Extracting Paper Contributions

Task
You are provided with the paper titled {title}. Here are the main sections of the paper: {sections}. Furthermore, key equations from the paper are provided to help you understand its specific algorithms: {equations}. Your task is to analyze the provided research paper and identify its **Core Components**. For each Component, you must provide a clear, concise, and implementable definition.

INSTRUCTIONS
1. **Identify Core Components**: Read the paper to identify its primary components. A componnet is not limited to a single algorithm; it can be a novel methodology, reusable techniques, key insight/finding, open-source datasets/benchmarks, etc.
2. **Categorize Each Component**: Assign one of the following types to each component you identify:

- `Methodology`: A novel, end-to-end procedure proposed by the paper for solving a problem. This can be an entire algorithm or model architecture design that addresses a specific research challenge. It must correspond to a systematic and complete end-to-end code implementation. When composed of multiple atomic sub-techniques, represent using the "components" field. Ensure the methodology can be implemented standalone, instead of a generic theoretical definition or a high-level outline of a framework.

- `Technique`: A self-contained and algorithmically implementable component, applied within the paper's Methodology or Experiment Process. It is either a novel module from this work, or a traceable technique from prior research. When composed of multiple atomic sub-techniques, represent using the "components" field. Ensure each technique can be implemented standalone, requiring NO integration with other modules to constitute a single code module. Exclude theoretical points and experimental tricks not directly tied to code implementation. Move them to the "Finding" category.

- `Finding`: A significant empirical or theoretical insight which can refer to an intriguing experimental finding, a powerful theoretical proof, or a promising research direction.

- `Resource`: A PUBLICLY available dataset or benchmark originally constructed in this paper.

3. **Define and Detail**: For each component, provide a detailed definition adhering to the following rules:

- **Fidelity**: All definitions must originate strictly from the provided paper. Do not invent details.
- **Atomicity & Modularity**: Each component, whether high-level or a component, should be defined as a distinct, self-contained unit. Explain its inputs, core logic, and outputs.
- **Reproducibility**: Retain as much original detail as possible. The definition should be comprehensive enough for an engineer or researcher to understand and implement it.
- **Structure**: If a 'Methodology' or a 'Technique' is composed of smaller 'Technique's, represent this hierarchical relationship using nested bullet points. This is crucial for understanding how the parts form the whole. Don't list techniques individually if they're already part of a larger technique/methodology.

OUTPUT FORMAT
Organize the extracted techniques into a list of dictionaries, with the final answer wrapped between two ``` markers. The keys for each dictionary are described below:
1. name: str, the name of the component, expressed as a concise and standardized academic term, intended to precisely capture its core identity while facilitating efficient indexing and retrieval from other literature.
2. type: str, One of 'Methodology', 'Technique', 'Finding', or 'Resource'.
3. description: str, A detailed, self-contained explanation of the component, focusing on what it is, how it works, and its purpose. For implementable items, describe the whole process without missing any critical steps and implementation details. For insights, describe the core discovery. Maximize the retention of description and implementation details from the original text.
4. components: List[dict], Optional, If the component is a complex 'Methodology' or 'Techinque' composed of multiple smaller techniques, this field lists its key sub-techniques. Each sub-technique listed here must also be defined separately as a complete technique object following this same JSON schema (with 'name', 'type' and 'description' as dictionary keys), allowing for hierarchical and recursive decomposition. ATTENTION: Only 'Methodology' and 'Technique' can have 'Technique' as its components!!!

Now please think and reason carefully, and wrap your final answer between two ``` in the end.

E.2 Code Repository Parsing

Prompt for Generating Code Repository Overview

Task
Analyze this GitHub repository {name} and create a structured overview of it.

Input
1. The complete file tree of the project: {file_tree}
2. The README file of the project: {readme}

Output
Create a detailed overview of the project, including:
1.Overview (general information about the project)
2.System Architecture (how the system is designed)
3.Core Features (key functionality)

Organize the overview in a clear and structured markdown format.

Please wrap your final answer between two ``` in the end.

Prompt for Finding Associated Paper from Code

Task
Analyze this GitHub repository {name}, and determine whether this repository is directly associated with a specific academic paper.

Input
The README file of the project: {readme}

Output
1. If you can find clear evidence that this repository is the official or direct code implementation of a specific academic paper, return the full title of the paper as a string.
2. If there is no sufficient evidence to identify a directly corresponding paper (e.g., only general descriptions, multiple papers, or no paper mentioned), return None.

Please wrap your final answer between two ``` in the end.

E.3 Knowledge Graph Construction

Prompt for Rewriting a Technique's Description

Task
Your task is to refine and enhance the description of a technical concept extracted from a research paper {paper}. The goal is to produce a clear, concise, and comprehensive description that accurately captures the essence of the technique.

Input
1. Technical Concept from the paper {paper}: {technique}
2. Relevant Excerpt of this Technique: {excerpt}

Output
Return a precise and comprehensive description, presented as a single, continuous paragraph written in a comprehensive, academic style. Avoid using bullet points, numbered lists, or other form of itemization.
1. Ensure the technique precisely matches the original description. DO NOT alter, expand, or reduce the scope of the technique. Ignore other related techniques and only FOCUS ON this technique.
2. Strictly adhering to the original description, augment its implementation details based on the provided excerpts. All formulas, parameter configurations, and implementation details must be extracted from the given excerpts, ensuring strict adherence to them. Avoid any summarization, inference, or omission.
3. If the excerpts offer no new information, leave the description unchanged. Your response MUST be based solely on the original description and provided excerpts. The inclusion of ANY external information or fabricated details is strictly forbidden!!!
4. Ensure that the provided description is precise, complete, and possesses sufficient detail to correspond to a specific implementation.

Now please think and reason carefully, and wrap your final answer between two ``` in the end.

Prompt for Identifying Relevant Code Snippets

Task
Your task is to analyze a list of code files retrieved from a GitHub repository, and identify which files are directly relevant to the implementation of a specific technical concept defined in an academic paper {paper}.

Input
1. Technical Concept Definition from the paper {paper}: {technique}
2. Overview of the Code repository: {overview}
3. Relevant Code Files: {file_snippets}

Output
Return a list of filenames formatted as ["xx", "xx", ...], sorted in **descending** order of relevance of the technical concept.
1. Exclude any file not DIRECTLY correspond to the concrete implementation and configurarion of this technique (e.g., tests, documentation, other technique implementation).
2. Confirm that a direct implementation exists within your provided file list. If no such implementation can be found, return None.
3. Return the filename list even if there's only one file.

Now please think and reason carefully, and wrap your final answer between two ``` in the end.

Prompt for Reranking Retrieved Techniques

Task
Your task is to analyze a list of technique implementations retrieved from the knowledge base, and identify which techniques are directly relevant to the implementation of a specific technical concept.

Input
1. Technical Concept Definition: {technique}
2. Relevant Technique implementations: {relevant_techniques}

Output
Return a list of (technique_name, apply_guidance) tuples formatted as [("", ""), ("",""), ...], sorted in descending order of relevance to the technical concept. The guidance should be a short explanation of how the technique applies to the current scenario and what modifications are needed for adaptation. Use clear and definite wording, avoiding parentheses.
1. Exclude any techniques not relevant to the concrete implementation of this technique.
2. Ensure the returned technique name exactly matches the original one.
3. For technologies with identical core definitions, keep the one whose application is most relevant.
4. If no such technique can be found, return None.
5. Return the filename list even if there's only one relevant technique.

Now please think and reason carefully, and wrap your final answer between two ``` in the end.

Prompt for Rewriting Code for a Leaf Technique

Task
Your task is to transform a collection of disparate source code snippets, which are the official implementation of a technique component from a research paper {paper}, into a single, self-contained, and executable code block. The final code block must be clean, well-documented, and easy for others to understand and run.

Input
1. Abstract of the paper {paper}: {abstract}
2. Technical Concept Definition from the paper {paper}: {technique}
3. Relevant Code Files: {file_snippets}

Workflow
1. Analyze: Understand the technique's inputs, outputs and workflow from the paper. Focus ONLY on THIS technique, ignoring the mentioned context and related techniques.
2. Isolate & Extract: Based on the description of the technique, determine what is its PRECISE role and functionality, and extract ONLY the code you identified as belonging to {technique}. Other mentioned associated techniques **MUST BE IGNORED AND EXCLUDED**.
3. Refactor: Integrate the extracted code by removing hard-coded values, isolating the core algorithm, and standardizing it with proper documentation and type hints.
4. Assemble & Test: Build the final script and add an test block as a runnable example. Ensure accuracy and conciseness, avoiding unnecessary output.
5. Documentation: Write a brief and concise documentation of the code logic, configurable options, and usage in 5-10 sentences.

Requirements
1. Dependency Management: Ensure all necessary imports and dependencies are included at the beginning of the code block.
2. Fidelity to the Original Technique: Strictly follow the description of the given technique to organize the code. ONLY focus on the implementation that DIRECTLY corresponds to THIS technique!!! (e.g., if the technique is a loss function definition, implement only the code for its calculation. Ignore all other parts of the algorithm's implementation, even if provided in the code snippets.)
3. Code Encapsulation and Documentation:

- Encapsulate the core logic of the technique into one or more functions/classes.
- Every function and class method must include a comprehensive docstring explaining its purpose, parameters, and return values.
- All function arguments and return values must have clear type hints.
- Preserve original parameters and comments from the source code.

4. Reproducibility and Testing:

- A main execution block, starting with the comment `# TEST BLOCK`, is required at the end of the file, which serves as a practical usage example and a test case.
- The test case should use parameters from the code repository or paper. If missing, create and state your own defaults.

5. Fidelity to the Original Logic:

- You must strictly adhere to the algorithmic logic present in the provided code snippets. Your role is to refactor and structure, not to re-implement or invent new logic.
- Minimal, necessary modifications are permitted (e.g., renaming variables for clarity, adapting function signatures for dependency injection), but the core computational steps must remain identical to the original author's implementation.

6. Documentation of Usage Scenarios: Provide a concise and fluent document of the code module's core logic, configurable options, and usage. Limit the description to 5-10 clear and coherent sentences.

Output
1. Implement the technique standalone without relying on external, undefined components. Return an executable code block and a corresponding documentation, each wrapped between two ``` .
Example:
[... Reasoning Steps ...]
```python
[... Core Implementation of the technique ...]
[... Ignore other relevant techniques ...]
# TEST BLOCK [... Example Usage ...]
```
The brief documentation of the code:
```
[...Brief Documentation ...]
```
2. Verify that the generated code does not exceed the scope of the technique's definition. If the technique requires integration with other modules to constitute a single code module, return None. If no direct implementation of the technique is found in the given code snippets, also return None.

Now, please proceed with the task, following the workflow and adhering to all requirements. Generate the final code block and documentation wrapped between two ``` separately at the end.

Prompt for Rewriting Code for a Composite Technique

Task
Your task is to transform a collection of disparate source code snippets, which are the official implementation of a technique component from a research paper paper, into a single, self-contained, and executable code block. The final code block must be clean, well-documented, and easy for others to understand and run.

Input
Abstract of the paper {paper}:
{abstract}
Technical Concept Definition from the paper {paper}:
{technique}
Sub-techniques and Associated Code:
{sub_techniques}
Relevant Code Files:
{file_snippets}

Workflow

Analyze: Understand the technique's inputs, outputs and workflow from the paper.
Locate: Fully reuse the code of the provided sub-techniques. For any uncovered parts, locate the relevant implementation logic from the given code snippets.
Refactor: Integrate the extracted code by removing hard-coded values, isolating the core algorithm, and standardizing it with proper documentation and type hints.
Assemble & Test: Build the final script and add an test block as a runnable example. Ensure accuracy and conciseness, avoiding unnecessary output.
Documentation: Write a brief and concise documentation of the code logic, configurable options, and usage in 5-10 sentences.

Requirements
Dependency Management: Ensure all necessary imports and dependencies are included at the beginning of the code block.
Fidelity to the Original Technique: Strictly follow the description of the given technique to organize the code. ONLY focus on the implementation that DIRECTLY corresponds to THIS technique!!! (e.g., if the technique is a loss function definition, implement only the code for its calculation. Ignore all other parts of the algorithm like model definition or training loop). Return None if no direct implementation is found.
Code Encapsulation and Documentation:

- Encapsulate the core logic of the technique into one or more functions/classes.
- Every function and class method must include a comprehensive docstring explaining its purpose, parameters, and return values.
- All function arguments and return values must have clear type hints.
- Preserve original parameters and comments from the source code.

Reproducibility and Testing:

- A main execution block, start with the comment `# TEST BLOCK`, is required at the end of the file, which serves as a practical usage example and a test case.
- The test case should use parameters from the code repository or paper. If missing, create and state your own defaults.

Fidelity to the Original Logic:

- You must strictly adhere to the algorithmic logic present in the provided code snippets. Your role is to refactor and structure, not to re-implement or invent new logic.
- Minimal, necessary modifications are permitted (e.g., renaming variables for clarity, adapting function signatures for dependency injection), but the core computational steps must remain identical to the original author's implementation.

Documentation of Usage Scenarios: Provide a concise and fluent document of the code module's core logic, configurable options, and usage. Limit the description to 5-10 clear and coherent sentences.

Output
1. Implement the technique standalone without relying on external, undefined components. Return an executable code block and a corresponding documentation, each wrapped between two ```.
Example:

[... Reasoning Steps ...]
```python
[... Core Implementation of the technique ...]
[... Ignore other relevant techniques ...]
# TEST BLOCK
[... Example Usage ...]
```
The brief documentation of the code:
```
[...Brief Documentation ...]
```
2. Verify that the generated code does not exceed the scope of the technique's definition. If the technique requires integration with other modules to constitute a single code module, return None. If no direct implementation of the technique is found in the given code snippets, also return None.

Now, please proceed with the task, following the workflow and adhering to all requirements. Generate the final code block and documentation wrapped between two ``` separately at the end.

Prompt for Verifying Rewritten Code

Task
Your task is to determine if the given code block strictly follows the provided technique description and relevant code files.

Input
Technical Concept Definition from the paper {paper}: {technique} Relevant Code Files: {file_snippets} Implemented Code Block: {code}

Output
1.Return False if the implementation is unrelated to the technique.
2.Return False if the implementation contains core logic cannot be located in the given relevant code files.
3.Return False if the implementation contains logics not covered in the technique description (e.g., the technique defines a submodule, but the code implements the full algorithm).
4.Return True if the code implements exactly what is specified in the technique description without adding any unnecessary features beyond the technical concept, and strictly follows the implementation in the given code files.

Now please think and reason carefully, provide a detailed analysis process for the above criteria, and wrap your final answer between two ``` in the end.

Prompt for Decomposing a Task into Techniques

Task
Your task is to decompose a complex academic task into its automic fundamental techniques based on its description.

Input
Academic Task Definition: {description}

Output
Return a list of (name, description) tuples in the format [("...", "..."), ("...", "...")], sorted by their importance to the task composition in descending order. Use clear and definite wording, avoiding parentheses. Each tuple must represent a distinct, fundamental academic concept that is reusable and traceable in other literature. Each tuple is explicitly mentioned or directly relevant to the target task. Avoid overly broad or vague techniques; each should have a clear, specific code implementation. Avoid trivial techniques like Cosine Similarity that require no literature review. If the task's implementation does not involve any specific academic concepts (e.g., purely engineering, configuration, or organizational task), simply return None.

Now please think and reason carefully, and wrap your final answer between two ``` in the end.