Executable Knowledge Graphs for Replicating AI Research

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 modular and pluggable knowledge base that automatically integrates technical insights, code snippets, and domain-specific knowledge 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 replication¹.

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 back-

ground materials to reproduce results, enabling machines to perform the same workflow reliably remains an open challenge (Chen et al., 2025).

Why Executable Knowledge Graphs? Existing attempts (Tang et al., 2025; Ou et al., 2025) to convert papers into knowledge bases show promising signs but often stop at shallow scaffolding rather than delivering rigorous, reproducible implementations. Three key issues limit agent-driven reproduction: (1) most approaches fail to extract deeper technical insights hidden in cited references and background literature; (2) they overlook practical signals embedded in concrete code implementations; and (3) the lack of a structured, unified representation prevents effective retrieval, composition, and reuse of scientific concepts and their executable components (Hua et al., 2025).

To address these gaps, we propose the Executable Knowledge Graph (xKG), a novel knowledge representation that fuses textual paper knowledge with its corresponding executable code snippets. Unlike conventional KG, XKG captures both conceptual relations and runnable components, enabling agents to retrieve, reason about, and assemble the precise artifacts needed for faithful reproduction. We evaluate xKG by integrating it into three distinct agent frameworks—BasicAgent, IterativeAgent, and PaperCoder. Our experiments on PaperBench (Starace et al., 2025) demonstrate consistent and significant performance gains. The design of xKG is modular and extensible, facilitating its adoption and expansion across diverse research domains.

2 Executable Knowledge Graphs

2.1 Preliminary

We define the paper reproduction task as generating an executable code repository R from a paper P, modeled as $R = \mathcal{A}(P)$, where \mathcal{A} is an agent. The primary benchmark for this task evaluates the

^{*} Equal Contribution.

[†] Corresponding Authors.

https://github.com/zjunlp/xKG.

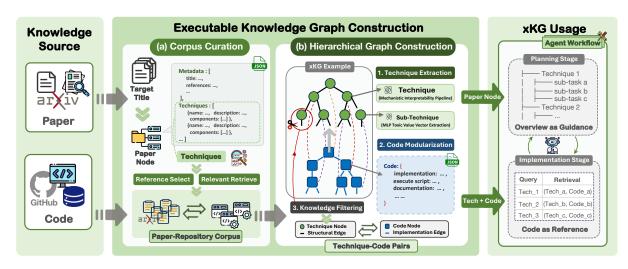


Figure 1: xKG is constructed automatically from arXiv papers and GitHub repositories (Examples at Appendix D).

functional correctness of R against an evaluation rubric \mathcal{T} . A final Replication Score, $S = \mathcal{E}(R, \mathcal{T})$, quantifies the weighted proportion of criteria met.

2.2 Design Formulation

We model xKG as a hierarchical, multi-relational graph $xKG = (\mathcal{N}, \mathcal{E})$, which is composed of various node types and edge types defined as:

$$\mathcal{N} = \mathcal{N}_P \cup \mathcal{N}_T \cup \mathcal{N}_C \tag{1}$$

$$\mathcal{E} = \mathcal{E}_{\text{struct}} \cup \mathcal{E}_{\text{impl}} \tag{2}$$

We first define the 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 $\{n_c\}_j$.
- Technique Node (n_t) : A self-contained academic concept $n_t = (D_t, \{n_t'\}_k)$ with its definition D_t and sub-nodes $\{n_t'\}_k$, ranging from a complete framework to a 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 then linked by the following two primary types of edges:

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

Through the above design, it becomes possible to link specific techniques (e.g., "Contrast-Consistent

Search" in Burns et al. (2022), see Figure 5) from papers, as well as associate these techniques with code snippets, yielding more precise knowledge.

2.3 Executable Knowledge Graph Construction

2.3.1 Corpus Curation

Our corpus curation strategy is a fully automated, paper-centric pipeline designed for scalability. For each paper targeted for reproduction (papers in PaperBench (Starace et al., 2025)), we employ o4mini to identify its core techniques, which drive a two-pronged collection process. Note that we strictly do NOT use the GitHub repositories or third-party reproduction repositories listed in PaperBench's blacklist to avoid any risk of data leakage. We first perform reference-based selection, expanding the corpus by filtering each paper's references and retaining the top five most valuable works, ranked by their technical contribution and overlap. Next, we conduct technique-based retrieval, using the top-contributing techniques as keywords to retrieve additional papers from the web. All retrieved papers are processed to fetch their LATEX sources from arXiv and then identify the associated GitHub. A rule-based filter is applied to retain papers with official repositories, resulting in the curated corpus of paper-repository pairs.

2.3.2 Hierarchical Graph Construction

Based on the corpus obtained above, we then proceed to construct the xKG, including the following three automated steps:

• **Step 1: Technique Extraction.** We first use o4-mini to deconstruct the paper's methodology

Method	Model					One-SBI							Average	
		vanilla	+xKG	vanilla	+xKG	vanilla	+xKG	vanilla	+xKG	vanilla	+xKG	vanilla	+xKG	
	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	$24.57_{+6.68}$	
DasicAgeiii	DS-R1	33.05	$39.14_{\pm 6.09}$	40.55	$39.14_{-1.41}$	17.22	$24.49_{+7.27}$	31.56	$33.97_{\pm 2.41}$	17.08	$21.38_{\pm 4.30}$	27.89	$31.62_{\pm 3.73}$	
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	$31.91_{+7.31}$	
neranveAgem	DS-R1	16.20	$47.40_{+31.20}$	31.19	$31.78_{\pm 0.59}$	31.09	$26.57_{-4.52}$	35.30	$38.44_{\pm 3.14}$	21.32	$31.89_{\pm 10.57}$	27.02	$35.22_{\pm 8.20}$	
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	$53.21_{+10.90}$	
rapercoder	DS-R1	43.24	$49.26_{\pm 6.02}$	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	$60.34_{\pm 8.11}$	

Table 1: **Main results on PaperBench Code-Dev.** We evaluate on the official lite subset of PaperBench, consisting of five papers: MU-DPO, TTA-FP, One-SBI, CFG, and FRE (details in Table 3). Results are reported using the *Replication Score* (%) metric with o3-mini as evaluator. All scores are shown as best@3 to mitigate task stochasticity and tool-related failures.

into a preliminary hierarchical tree of Technique Nodes \mathcal{N}_T linked by Structural Edges e_{struct} . Subsequently, we utilize RAG² (treating the paper as a document) to enrich each node by retrieving relevant text from the paper, which is then synthesized into a comprehensive definition D_t . This step yields a set of detailed yet unverified techniques that may contain noise.

- Step 2: Code Modularization. For each technique n_t , its definition is used as a query to retrieve relevant code snippets, following the similar RAG-based procedure (treating the code as a document) as in Step 1. We then employ o4-mini to synthesize these snippets into a candidate Code Node n_c , which includes the implementation σ , a test script τ , and accompanying documentation δ . This candidate node is then organized in a modular fashion and subjected to an iterative self-debugging loop to verify the executability of each module, ultimately producing a set of fully executable Code Nodes \mathcal{N}_c along with their associated Implementation Edges e_{impl} .
- 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. We aim to automate

this process to enable knowledge scaling.

2.4 Using Executable Knowledge Graphs

In a practical reproduction workflow, a LLM agent can use xKG at two critical stages. For high-level planning, the agent fetches the target paper's Paper Node (without all Code Nodes) to grasp its core techniques and overall structure. During low**level implementation**, the agent queries XKG for semantically relevant (Technique, Code) pairs to aid in specific functionalities. These two steps can be supplied either as callable tools for ReActstyle agents or as pluggable components of fixedworkflow agents. Crucially, to combat knowledge noise, all retrieved candidates are processed by a final LLM-based Verifier (o4-mini). This verifier acts as a critical quality gate, filtering, reranking, and refining the results to ensure that the retrieved knowledge is highly relevant and implementable.

3 Experiments

3.1 Settings

We evaluate xKG on the lite collection of Paper-Bench Code-Dev using a structured rubric (Starace et al., 2025), a weighted tree of binary criteria whose leaves are aggregated by an o3-mini-based evaluator into a single score. 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 repository-level reproduction agent with a fixed workflow). 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

²We employ text-embedding-3-small throughout all stages of xKG construction.

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. 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. This striking contrast reveals a critical dependency on the target paper (details in Appendix B).

3.3 Further Analysis

Method	Score (%)	$\textbf{Drop}(\nabla)$	
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 the core components of xKG. Performance is averaged over all 5 papers.

Code-based structured knowledge aids AI research replication. As shown in Table 2, our ablation study conducted on PaperCoder framework with o3-mini setup, reveals that removing any component degrades performance. The most significant drop occurs when removing Code Nodes, decreasing the score by 4.56% (53.21% $\to 48.65\%$), 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 high-level structural overview of the target task. In contrast, omitting **Technique Nodes** results in a modest 1.05% drop, since the function of each technique is already implicitly captured by the Code Nodes, rendering the explicit description redundant.

Successful reproduction hinges on retrieved code quality. Building on the above findings, we conduct a further analysis into how the quality of Code Nodes within xKG influences performance. Using PaperCoder with o3-mini on two high-gain papers, MU-DPO and TTA-FP, we compare xKG with four configurations, each repeated three times to mitigate stochasticity (Figure 2): w/o Code, without access to any code nodes; + Raw

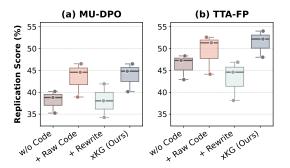


Figure 2: Further study on Code Node quality.

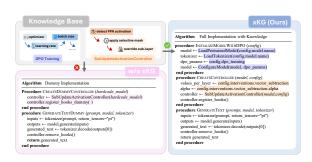


Figure 3: Case Study on MU-DPO. A comparison of performance with and without xKG on IterativeAgent.

Code, which incorporates code nodes with raw retrieved snippets; **+ Rewrite**, using LLM-rewritten executable nodes but omitting the verification step.

As illustrated in Figure 2, our full approach not only achieves the highest mean score but also exhibits low variance. Notably, even incorporating raw code snippets (+ Raw Code) improves performance, validating that our method effectively localizes necessary code. A critical insight emerges from the + Rewrite setting, which underperforms even the raw snippet baseline. We attribute this to a misleading guidance phenomenon: well-formatted but contextually irrelevant code can cause the agent to blindly adopt the retrieved snippets, deviating from the target paper's specific implementation.

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 3). We notice that xKG enriches information granularity, allowing agents to generate critical details accurately, and improves modular implementation capability, enabling agents to reuse verified code for functionally correct implementations, as illustrated by the case colors in Figure 3.

4 Conclusion

We introduce XKG, which improves AI research replication. We aim for xKG to serve as an AI-for-Research knowledge base, reducing noise from web retrieval and improving efficiency.

Limitations

This work has several limitations. First, the Paper-Bench 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.

References

- Collin Burns, Haotian Ye, Dan Klein, and Jacob Steinhardt. 2022. Discovering latent knowledge in language models without supervision. *arXiv* preprint *arXiv*:2212.03827.
- 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. Open-Review.net.
- Manuel Glöckler, Michael Deistler, Christian Dietrich Weilbach, Frank Wood, and Jakob H. Macke. 2024. All-in-one simulation-based inference. In Fortyfirst International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024. Open-Review.net.
- Tianyu Hua, Harper Hua, Violet Xiang, Benjamin Klieger, Sang T Truong, Weixin Liang, Fan-Yun Sun, and Nick Haber. 2025. Researchcodebench: Benchmarking Ilms on implementing novel machine learning research code. *arXiv preprint arXiv:2506.02314*.
- 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. Open-Review.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*.
- Shuo Yan, Ruochen Li, Ziming Luo, Zimu Wang, Daoyang Li, Liqiang Jing, Kaiyu He, Peilin Wu, George Michalopoulos, Yue Zhang, et al. 2025. Lmrbench: 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 Setup

A.1 Benchmarks

The original PaperBench benchmark (Starace et al., 2025) is designed to evaluate the ability of AI agents to reproduce state-of-the-art AI research from scratch. The full benchmark includes 20 recent papers from top-tier machine learning conferences (e.g., ICML 2024), where agents must understand each paper, develop a complete codebase, and replicate its empirical results.

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 domains including machine learning (ML), reinforcement learning (RL), and natural language processing (NLP). Furthermore, since PaperBench shows that BasicAgent and IterativeAgent gain little performance improvement after one hour, we cap their execution time at one hour for efficiency and cost reasons.

Evaluation follows a structured hierarchical rubric co-developed with the original authors, and an LLM-based(o3-mini) evaluator aggregates final scores using a weighted binary criteria tree. More specific details about the papers and their evaluation nodes are listed in Table 3.

A.2 Configuration Details

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 4-7. In addition, the specific prompts designed in our system are detailed in Appendix C.

B Further Analysis on Target Paper

As illustrated in Figure 4, 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 com-**

ponents, where excelling at core techniques highlighted by XKG leads to the neglect of secondary objectives.

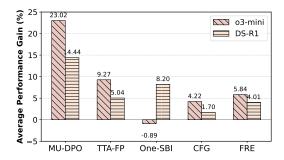


Figure 4: Average performance gain per paper.

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 argumentation* to the intrinsic *innovative capacity* of the base LLM itself.

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 3: Abbreviations and statistics for the PaperBench tasks evaluated in this work. We assign a brief abbreviation to each paper for easier reference. The "CodeDev Nodes" column specifies the number of nodes to evaluation for each reproduction task.

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

Table 4: 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.		
<pre>paper.text_splitter.chunk_size</pre>	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 5: Hyperparameters for the Paper-RAG module in xKG.

Hyperparameter	Value	Description	
Knowledge Graph Retrieval	-11 MiniM IC2	The contains the reference and allowed for rela-	
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 6: Hyperparameters for Knowledge Graph retrieval.

Hyperparameter	Value	Description				
Global & Model Profile Configura	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 core reasoning 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 7: 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.

C 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.

C.1 Paper Parsing Prompts

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.

C.2 Code Repository Parsing Prompts

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.

C.3 Knowledge Graph Construction Prompts

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 configuration 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 nitrogen 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 nitrogen 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.

D Running Examples of xKG

```
"paper_title": "Discovering Latent Knowledge in Language Models Without Supervision",
 3
         "paper_abstract": "Existing techniques for training language models can be misaligned with the truth: if we train mod
         "paper_references": [
 4 -
           "A general language assistant as a laboratory for alignment",
 8 -
         "findings": [
           "CCS Outperforms Zero-Shot Prompting: Empirically, CCS exceeds the performance of strong zero-shot prompting baseli
10
          "... (Omit the rest)
11
12 -
         "techniques": [
13 🕶
          {
            "name": "Contrast-Consistent Search (CCS)",
15
            "description": "Contrast-Consistent Search (CCS) is an end-to-end, unsupervised procedure for extracting a latent
16 -
            "code": {
              "implementation": "import numpy as np\nimport torch\nimport torch.nn as nn\nfrom transformers import AutoModel,
17
                                                      # Synthetic demonstration of CCS on random features\n
              "test": "if __name__ == \"__main__\":\n
19
              "documentation": "This module implements Contrast-Consistent Search (CCS), an unsupervised method to extract a
20 -
              "package": [
                 "numpy",
21
                "torch",
22
                "transformers"
24
              ]
25
             "components": [
27 🕶
                "name": "Construction of Contrast Pairs",
28
                "description": "For each yes—no question, we construct a contrast pair (x^*, x^*) by formatting the question in
29
30 -
                  32
                  "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
33
                  "package": [
36
                    "transformers"
                  1
37
38
                "components": []
40
41 -
                "name": "Feature Extraction and Normalization",
42
                "description": "In this approach, we compute the hidden representations of each contrast example using a pret
43
45
                  "implementation": "import numpy as np\nimport torch\nfrom transformers import AutoModel, AutoTokenizer\n\nd
                                     == \"__main__\":\n
                                                          # Example usage with BERT encoder-only model\n
46
                  "documentation": "This module implements the unsupervised feature extraction and normalization technique fo
                  "package": [
48 🕶
                    "numpy",
49
50
                    "torch"
51
                    "transformers"
53
54
                "components": []
              // ... (Omit)
            ]
          }
58
        ]
59
```

Figure 5: An example of 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.