HiRAS: A Hierarchical Multi-Agent Framework for Paper-to-Code Generation and Execution

Hanhua Hong[1,2], Yizhi Li[1], Jiaoyan Chen[1],
Sophia Ananiadou[1,3], Xiaoli Li[4], Jung-jae Kim[2], Chenghua Lin[1]
[1]The University of Manchester, [2]Institute for Infocomm Research (I²R), A*STAR
[3]ELLIS Manchester, [4]Singapore University of Technology and Design
hanhua.hong@postgrad.manchester.ac.uk
xiaoli_li@sutd.edu.sg, kim_jung_jae@a-star.edu.sg
{yizhi.li-2,jiaoyan.chen,sophia.ananiadou,chenghua.lin}@manchester.ac.uk

Abstract

Recent advances in large language models have highlighted their potential to automate computational research, particularly reproducing experimental results. However, existing approaches still use fixed sequential agent pipelines with weak global coordination, which limits their robustness and overall performance. In this work, we propose Hierarchical Research Agent System (HiRAS), a hierarchical multi-agent framework for end-to-end experiment reproduction that employs supervisory manager agents to coordinate specialised agents across fine-grained stages. We also identify limitations in the reference-free evaluation of the Paper2Code benchmark and introduce Paper2Code-Extra (P2C-Ex), a refined protocol that incorporates repository-level information and better aligns with the original reference-based metric. We conduct extensive evaluation, validating the effectiveness and robustness of our proposed methods, and observing improvements, including >10% relative performance gain beyond the previous state-of-the-art using open-source backbone models and significantly reduced hallucination in evaluation. Our work is available on GitHub: https://github.com/KOU-199024/HiRAS.

1 Introduction

Recent advances in artificial intelligence (AI), particularly large language models (LLMs), have demonstrated remarkable capability in code generation and software development (Dehaerne et al., 2022; Yetiştiren et al., 2023; OpenAI, 2024). These developments have motivated a growing amount of research exploring the use of AI systems to assist scientific research, especially within Computer Science (Chen et al., 2025; Eger et al., 2025). AI Scientist (Lu et al., 2024) is one of the earliest attempts, which aims to automate the entire scientific workflow. Beyond such all-in-one pipelines, a substantial number of studies focus on individual tasks in the research lifecycle, including idea generation (Wang et al., 2024; Li et al., 2025; Garikaparthi et al., 2025), paper review (Gao et al., 2024; James et al., 2024; Zhu et al., 2025), and experiment conduction (Guo et al., 2024a; Schmidgall et al., 2025).

Among these tasks, reproduction of experimental pipelines and results described in published papers is of particular importance, since cumulative scientific progress fundamentally depends on the reproducibility of prior works (Resnik and Shamoo, 2017; Pineau et al., 2021; James et al., 2026). However, the exponential growth in the number of research papers, combined with the fact that only around 20% of papers release complete and usable code repositories on average, has made exhaustive reproduction increasingly impractical for researchers (Lin et al., 2022; Magnusson et al., 2023; Seo et al., 2025). This limitation naturally motivates the development of LLM-based agents to automate experiment reproduction. To evaluate how well existing systems can reproduce experimental pipelines and results directly from research papers, several benchmarks, such as PaperBench (Starace et al., 2025) and Paper2Code (Seo et al., 2025), have been proposed. Generally, they use LLM-as-a-judge evaluation metrics due to the complexity of the task. Results on these benchmarks indicate that, despite strong general-purpose coding ability, directly prompted LLMs achieve only limited success (Xiang et al., 2025; Kim et al., 2025).

To address this gap, recent works have adopted multi-agent frameworks that decompose the reproduction process into multiple stages. For example, PaperCoder (Seo et al., 2025) structures reproduction into planning, analysis, and coding phases, while AutoReproduce (Zhao et al., 2025) combines search agents and coding agents to retrieve relevant studies and implement experimental code. Despite these advances, existing systems largely rely on fixed, sequential agent pipelines implemented via

Findings of the Association for Computational Linguistics: ACL 2026, pages 7624–7652

fixed prompt invocations with limited global supervision. Consequently, errors introduced by agents early in the pipeline can easily propagate, resulting in stalled workflows and incomplete repositories. This lack of adaptive coordination fundamentally limits robustness, fault tolerance, and scalability in complex experimental environments (Xie et al., 2025; Wei et al., 2025). In contrast to fixed sequential pipelines, hierarchical agent architectures have been shown to be effective across domains such as embodied intelligence (Lallement et al., 2014), reinforcement learning (Wang et al., 2020), and energy systems (Dragomir and Dragomir, 2025). By introducing explicit manager roles, hierarchical organisation enables effective communication across the system, thereby improving cooperation and scalability in complex multi-agent systems (Feng et al., 2024; Moore, 2025).

In this work, we propose **Hi**erarchical **R**esearch **A**gent **S**ystem (**HIRAS**), a novel multi-agent framework that introduces dedicated manager agents to supervise and control the global experiment reproduction process. Unlike prior hierarchical multi-agent systems, where manager agents passively deliver messages (Ahilan and Dayan, 2019; Wu et al., 2024), the managers in HIRAS function as proactive supervisors, which actively inspect task progress and dynamically invoke specialised subordinate agents to produce artefacts and correct errors. For example, a global manager may invoke a coding agent to implement experimental code and re-invoke it when the subsequent execution agent reports errors in the program. In addition, the reproduction workflow is decomposed into finer-grained phases for these specialised agents, which are equipped with appropriate tools to interact with a shared workspace. Together, these design choices enable effective collaboration and information exchange across the system. To the best of our knowledge, HIRAS is the first framework to incorporate global, supervisory manager agents in a hierarchical multi-agent architecture for automated experiment reproduction.

For evaluation, we conduct comprehensive experiments on the PaperBench and Paper2Code benchmarks using two popular backbone models, `Qwen3-Coder-480B` (Yang et al., 2025) and `DeepSeek-v3.1-Terminus` (DeepSeek-AI et al., 2025). Our approach consistently outperforms baselines across all settings when using the same backbone model. Specifically, on PaperBench, HIRAS even outperforms the current state-of-the-art methods which use proprietary LLMs by a substantial margin. On Paper2Code, our analysis further identifies fundamental limitations of the reference-free prompt-based evaluation protocol introduced in the original work (Seo et al., 2025), particularly its susceptibility to evaluator hallucination. To address this issue, we introduce **Paper2Code-Extra (P2C-Ex)**, a refined reference-free evaluation paradigm that incorporates explicit repository-level information. Empirical results show that P2C-Ex substantially improves alignment with reference-based evaluation, offering a more reliable framework for assessing paper-to-code consistency in the absence of gold repositories.

To summarise, the contributions of our work are three-fold:

- We propose **HIRAS**, a novel hierarchical multi-agent framework for automated experiment reproduction from research papers, which integrates specialised agents with supervisory manager agents for improved coordination.
- We evaluate our framework on two experiment reproduction benchmarks using three LLM backbones, demonstrating consistent and substantial improvements over prior approaches and achieving state-of-the-art performance.
- We introduce **P2C-Ex**, a reference-free evaluation protocol for the Paper2Code benchmark that incorporates repository-level information and yields stronger alignment with reference-based metrics than the original protocol of Paper2Code.

2 Related Work

2.1 AI for Scientific Research

The rapid advancement of LLMs has resulted in their increasing application in scientific research, particularly within the computer science domain (Gottweis et al., 2025; Xie et al., 2025; Si et al., 2025; Wu et al., 2026). Some studies explore the use of LLMs to perform the entire scientific research lifecycle, including the AI Scientist (Lu et al., 2024; Yamada et al., 2025), DOLPHIN (Yuan et al., 2025), and Zochi (Intology, 2025), which produced papers accepted at major conferences.

In addition to such holistic frameworks, other work has focused on individual components of the research process. For example, SciMON (Wang et al., 2024) and Chain-of-Ideas (Li et al., 2025)

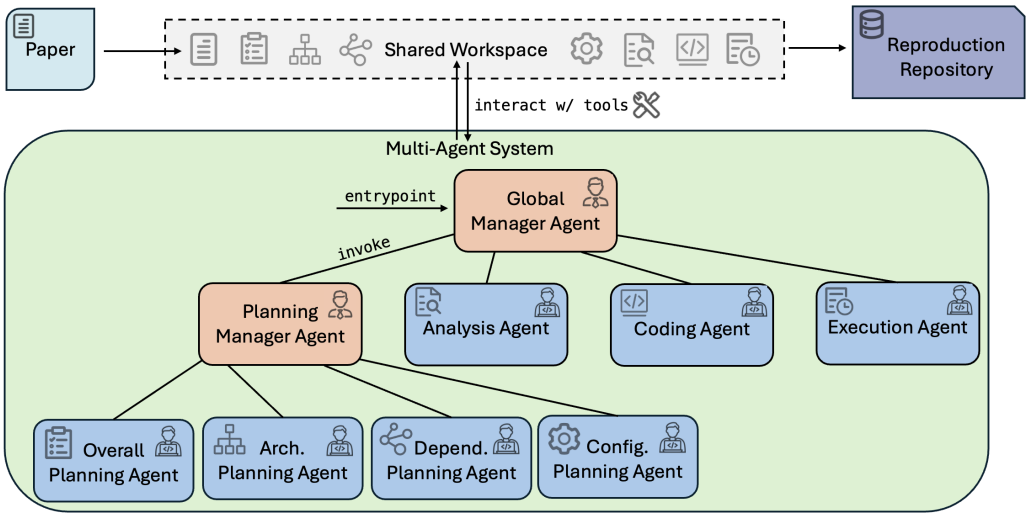


Figure 1: Overview of the HIRAS framework. The reproduction workflow is decomposed into fine-grained phases, each handled by a specialised agent (blue) equipped with appropriate tools to operate within a shared workspace throughout the process. To enhance coordination, HIRAS introduces hierarchical manager agents (orange) that inspect the workspace to supervise progress and dynamically invoke subordinate agents to perform tasks and correct errors according to the feedback results.

improve the generation of AI-driven ideas, whilst other frameworks have been developed to identify and assess the scientific rigour described in research papers (James et al., 2024). Another key direction is experiment reproduction, in which models reproduce experimental results from published studies. As benchmarks, SciReplicateBench (Xiang et al., 2025) evaluates reproducibility by requiring agents to complete missing code segments, and PaperBench (Starace et al., 2025) provides detailed, paper-specific rubrics for assessing end-to-end reproduction fidelity. To support reproduction, PaperCoder (Seo et al., 2025) reconstructs experimental codebases from published papers using multi-stage task decomposition, and AutoReproduce (Zhao et al., 2025) further enhances reproduction by automatically retrieving relevant literature and associated code repositories.

However, prior work on experiment reproduction has largely relied on fixed, sequential workflows, in which errors introduced by early-stage agents propagate unchecked. In contrast, our approach introduces a hierarchical management architecture that provides explicit supervisory control: manager agents actively inspect intermediate artefacts, diagnose failures, and re-invoke stage-specific agents to correct errors throughout the reproduction process.

2.2 Multi-Agent Systems

Multi-agent systems have gained increasing attention due to the widespread adoption of AI agents across diverse applications (Guo et al., 2024b; Li et al., 2024). A key distinction among these systems is their coordination strategies, which determine how agents communicate and collaborate (Sun et al., 2025a,b). While some adopt flat, decentralised schemes (Du et al., 2024; Sun et al., 2025b), hierarchical organisation has proven more effective for managing complexity and enhancing scalability (Moore, 2025). Examples include FMH (Ahilan and Dayan, 2019), where a manager delegates sub-goals to workers; Liang et al. (2024), which uses a judge to summarise agent debates; and AutoGen (Wu et al., 2024), where managers mediate and disseminate messages among agents.

Unlike prior works, where manager agents primarily act as passive communication intermediaries, our framework assigns them an active supervisory role. Managers oversee progress, inspect artefacts, and control the workflow by invoking phase-specific specialised agents. This enables effective global coordination throughout end-to-end experiment reproduction.

3 Methodology

3.1 Problem Definition

In this work, we use research agents for experiment reproduction. A **research agent** is defined as a tuple $\mathcal{A} = (\mathrm{LLM}, \mathrm{Mem}, T, E, K)$, comprising a backbone model LLM, an individual memory context Mem (including prompts), and access to a suite of tools T that enable interaction with a workspace environment E. The agents follow the

ReAct paradigm (Yao et al., 2023), performing up to K reasoning-action iterations, while optionally calling `end_task` to submit outputs early.

We use a system composed of multiple research agents defined above to deal with this task, denoted as $\mathcal{S} = \{\mathcal{A}_1, \mathcal{A}_2, \dots\}$. Given a research paper P, the objective of the multi-agent system $\mathcal{S}$ is to generate the full reproduction output, expressed as $R = \mathcal{S}(P)$.

3.2 Method Overview

The workflow of HIRAS is decomposed into fine-grained phases, executed by specialised agents, while dedicated manager agents supervise progress and dynamically invoke their subordinate specialised agents. An overview of HIRAS is shown in Figure 1.

3.3 Specialised Agents

In decomposing the reproduction workflow, we keep the three primary phases: *planning*, *analysis*, and *coding*, following PaperCoder (Seo et al., 2025), and introduce an additional *execution* phase to validate code executability. The planning phase is further decomposed into sub-phases: overall planning, architecture design, dependency modelling, and configuration generation. Unlike PaperCoder, which employs only one agent for planning, we assign each sub-phase to a specialised agent: $\mathcal{A}_{\text{overall}}, \mathcal{A}_{\text{arch}}, \mathcal{A}_{\text{dep}}, \mathcal{A}_{\text{config}}$, to sequentially generate the plans. Then, we instantiate a specialised agent $\mathcal{A}_t$ for each remaining phase, yielding $\mathcal{A}_{\text{analysis}}, \mathcal{A}_{\text{code}}, \mathcal{A}_{\text{exec}}$. Every agent is initialised with its initial memory context Mem_t.

All agents operate within a shared workspace E, initialised with only the paper $E = \{P\}$. To interact with this workspace, agents are equipped with a common file system toolset:

$$T_{\text{file}} = \{\texttt{list\_dir}, \texttt{read\_file}, \texttt{write\_file}\}, \quad (1)$$

enabling them to store outputs and access both the paper and artefacts generated by other agents. The execution agent has an extended toolset: $T_{\text{file}} \cup \{\texttt{bash}\}$, enabling direct interaction with the system console for program execution.

Specifically, during the planning phase, agents sequentially generate a set of structured plans: an overall plan that summarises the experiments described in the paper; an architecture design that specifies the components required for implementation and their structure; a dependency plan that models the function-level relationships among these components; and a configuration plan that details the experimental parameters. Each plan is produced by a dedicated sub-phase agent, collectively forming the plan set: $\text{Pln} = \{\text{Pln}_{\text{overall}}, \text{Pln}_{\text{arch}}, \text{Pln}_{\text{dep}}, \text{Pln}_{\text{config}}\}$.

The analysis agent $\mathcal{A}_{\text{analysis}}$ subsequently generates detailed implementation analysis for each experimental component specified in the generated plan, such as `dataloader.py`, `trainer.py`, etc. Together, the analysis files are denoted as Analysis.

Based on these analyses, the coding agent $\mathcal{A}_{\text{code}}$ generates the codebase C required for reproduction, which is then executed by the execution agent $\mathcal{A}_{\text{exec}}$ to produce logs and results L. The final workspace comprises: $E = \{P, \text{Pln}, \text{Analysis}, C, L\}$, satisfying the requirement of a complete reproduction R. Collectively, these specialised agents constitute the foundation of our framework.

Algorithm 1 Algorithm for Function `invoke`

Input: Research Agent $\mathcal{A}_t = \{\text{LLM}, \text{Mem}_t, T_t, E, K\}$; Instruction Prompt p;
Output: Ending Report Report; Updated Workspace E;

$\text{Mem}_t \leftarrow \text{Mem}_t \cup \{p\}$.
for $i \leftarrow 1$ to K **do**
  $\text{Reasoning}, \text{Action} \leftarrow \text{LLM}(\text{Mem}_t, T_t)$.
  $\text{Mem}_t \leftarrow \text{Mem}_t \cup \{(\text{Reasoning}, \text{Action})\}$.
  if $\text{Action} \in T_t$ **then**
    $\text{Result}, E \leftarrow$ `system.call(`Action`)`.
    $\text{Mem}_t \leftarrow \text{Mem}_t \cup \{\text{Result}\}$.
  else if Action **matches** `"(`$\mathcal{A}_{\text{sub}}, p_{\text{sub}}$`)"` **and** $\mathcal{A}_{\text{sub}} \in \text{Sub}(\mathcal{A}_t)$ **then**
    $\text{Result}, E \leftarrow$ `invoke(`$\mathcal{A}_{\text{sub}}, p_{\text{sub}}$`)`.
    $\text{Mem}_t \leftarrow \text{Mem}_t \cup \{\text{Result}\}$.
  else if Action **matches** `"end_task(Report)"` **then**
    return Report, E.
  end if
end for
$\text{Report} \leftarrow \text{Reasoning}$
return Report, E.

3.4 Hierarchical Orchestration

HIRAS is built on the principle that reliable research experiment reproduction requires continuous supervision rather than one-time task decomposition. *Manager agents* are therefore granted both global visibility into the shared workspace and authority to actively inspect intermediate artefacts, diagnose failures, and re-invoke subordinate agents with corrective instructions. We introduce two manager agents into the framework. The first one is the *planning manager* $\mathcal{A}_{\text{plan}}$, which oversees the entire planning phase by managing all sub-phase agents:

$$\mathrm{Sub}(\mathcal{A}_{\text{plan}}) = \{\mathcal{A}_{\text{overall}}, \mathcal{A}_{\text{arch}}, \mathcal{A}_{\text{dep}}, \mathcal{A}_{\text{config}}\}. \quad (2)$$

The second, and the most important, is the *global manager*, which supervises the overall reproduction process at the global phase level, managing the planning agent, analysis agent, coding agent, and execution agent:

$$\mathrm{Sub}(\mathcal{A}_{\text{global}}) = \{\mathcal{A}_{\text{plan}}, \mathcal{A}_{\text{analysis}}, \mathcal{A}_{\text{code}}, \mathcal{A}_{\text{exec}}\}. \quad (3)$$

Like the specialised agents, the manager agents are also initialised with the model LLM, their own initial context Mem_t, file system toolset T, shared environment E, the maximum iteration step K. Crucially, manager agents differ in that they are granted supervisory authority over the workflow: depending on the current stage of the reproduction process, manager agents can invoke the appropriate subordinate agents with an instruction prompt p:

$$\texttt{invoke}(\mathcal{A}_{\text{sub}}, p), \mathcal{A}_{\text{sub}} \in \mathrm{Sub}(\mathcal{A}_t). \quad (4)$$

When an invoked agent is a manager agent, it delegates tasks to the appropriate sub-agents through further invocations. In contrast, a specialised agent directly generates or refines the required artefacts, thereby advancing the reproduction process. The entire reproduction workflow is initiated by explicitly invoking the global manager with an initial prompt p_0, i.e., $\texttt{invoke}(\mathcal{A}_{\text{global}}, p_0)$, which serves as the system entry point. Algorithm 1 presents a detailed description of the core `invoke` function.

After each invocation, the manager agents will inspect the outputs in the workspace via the file system tools T_{file}. If results are incomplete, lack details, or exhibit signs of hallucination, the manager will instruct the corresponding subordinate agents to re-execute their tasks. This inspection step enables explicit detection and correction of intermediate errors.

Beyond quality control, manager agents also facilitate coordination across subordinate agents. When an agent encounters errors caused by another agent's outputs, it will report the issue to the manager, who then instructs the responsible agent for correction. This supervision ensures consistency and robust execution of the full reproduction process composed of multiple phases.

This design establishes a tree-structured hierarchical orchestration architecture that ensures steady progress, and explicit supervision over intermediate artefacts leading to high-quality outcomes throughout the full reproduction workflow. The initial context prompts for all agents are given in Appendix E.

Framework	Model	Score (%)
AutoReproduce	`o3-mini`	48.5
PaperCoder	`Claude-Sonnet`	51.1
HiRAS **(Ours)**		**64.1**
PaperCoder*		36.9
AutoReproduce*	`Qwen3-Coder-480B`	31.8
HiRAS **(Ours)**		45.7
PaperCoder*		40.8
AutoReproduce*	`DeepSeek-v3.1`	41.5
HiRAS **(Ours)**	`-Terminus`	**57.4**

Table 1: Main results on the PaperBench–CodeDev subset. An asterisk (*) indicates results reproduced by us. All evaluations in this table are conducted using `o3-mini-high` as the evaluator.

4 Experimental Setup

Backbone Models. We mainly implement our framework based on two open-source LLMs: (1) `Qwen3-Coder-480B` (Yang et al., 2025), an MoE model designed for agentic coding and tool use, making it well-suited for experiment reproduction tasks; and (2) `DeepSeek-v3.1-Terminus` (DeepSeek-AI et al., 2025), which is optimised for high-performance agentic behaviour in both search and code generation. Using two distinct backbones allows us to disentangle architectural contributions from model-specific effects. For simplicity, we will also name them as `Qwen3` and `DeepSeek-v3.1`, respectively.

To further assess generalisability with proprietary models, we additionally employ `Claude-Sonnet` as the backbone in the PaperBench-CodeDev experiment.

Benchmarks. To evaluate the effectiveness of our framework, we conduct experiments on two experiment reproduction benchmarks. First, **PaperBench** (Starace et al., 2025) comprises 20 machine learning papers from ICML 2024, each accompanied by a manually constructed evaluation rubric. Second, **Paper2Code** (Seo et al., 2025) contains 90 papers collected from ICML 2024, NeurIPS 2024, and ICLR 2024. These benchmarks jointly assess end-to-end reproduction fidelity across planning, implementation, and execution stages.

Evaluation Protocol. For PaperBench, we adopt the original tree-structured evaluation rubrics provided with each paper, which assess generated repositories along three categories of requirements: code development, execution, and result match-

Framework	PaperBench-CodeDev (%)				PaperBench (%)	
	o3-mini-high		ChatGPT-4o-mini		ChatGPT-4o-mini	
	Qwen3	DeepSeek-v3.1	Qwen3	DeepSeek-v3.1	Qwen3	DeepSeek-v3.1
PaperCoder	36.9	40.8	26.2	34.9	11.5	16.9
AutoReproduce	31.7	41.5	26.3	35.8	11.2	17.1
Specialised Agents	37.5	47.5	29.9	37.9	15.1	18.7
+*Refinement*	40.5	51.2	31.8	39.6	18.2	19.5
HiRAS (**Ours**)	**45.7**	**57.4**	**34.3**	**43.1**	**19.1**	**21.5**

Table 2: Extended robustness validation and ablation study on PaperBench-CodeDev and the full PaperBench benchmark. PaperBench-CodeDev results are evaluated using both ChatGPT-4o-mini and o3-mini-high, while full PaperBench results are evaluated using ChatGPT-4o-mini only.

```
- README.md
- config.yaml
- datasets/
      - data_loader.py
- code/
      - model.py
- main.py
```

Figure 2: An example of the repository structure

ing, using percentage scores to indicate comprehensiveness of the reproduction. Following previous work (Seo et al., 2025; Zhao et al., 2025), we first evaluate our framework on PaperBench-CodeDev subset with o3-mini-high (Zhang et al., 2025) as the evaluator to showcase the overall performance. We further use ChatGPT-4o-mini (OpenAI, 2024) as an alternative evaluator to assess robustness across evaluators, first validating consistency trends on the PaperBench–CodeDev subset and then extending to the full PaperBench benchmark owing to its cost-efficiency for large-scale evaluation.

For the Paper2Code benchmark, evaluation is conducted on a 1–5 scoring scale under two settings: *reference-free* and *reference-based*. The reference-free metric directly evaluates the generated code repository against the paper, whereas the reference-based metric additionally provides the gold-standard repository in the prompt.

However, our initial studies reveal that reference-free scores based on the original protocol by Paper2Code are systematically overestimated, making them unreliable. To remedy this, we introduce **Paper2Code-Extra (P2C-Ex)**, a refined reference-free evaluation protocol that enhances the evaluator prompt with repository-level context and additional instructions. Specifically, the prompt includes (i) the total file count in the repository, (ii) a hierarchical repository structure illustration, and (iii) improved instructions to mitigate hallucination. The full prompts are presented in Appendix D. An example of repository structure is displayed in Figure 2. To quantify the consistency between different evaluation protocols, we report the Pearson correlation coefficient (r) between the reference-based setting and each reference-free setting. All Paper2Code evaluations are conducted using o3-mini-high as the evaluator.

Baselines. We evaluate our framework against two baselines for experiment reproduction. **PaperCoder** (Seo et al., 2025) employs a sequential multi-stage pipeline with a dedicated agent for each stage to generate codebases from papers. **AutoReproduce** (Zhao et al., 2025) is a multi-agent system with search and coding agents that improve reproduction quality by leveraging information from related papers. For a fair comparison, we reproduced both baselines using the Qwen3 and DeepSeek-v3.1 backbones.

For the Paper2Code benchmark, we additionally include three naïve baselines to demonstrate the intrinsic limitation of the original reference-free evaluation: (i) an empty repository, (ii) repositories containing only configuration or documentation files (e.g., .md or .yaml) extracted from the gold reference repository, and (iii) the gold reference repositories themselves.

Environment. All experiments are conducted with eight NVIDIA H200. We use smolagents (Roucher et al., 2025) as our agent scaffold.

5 Experimental Results

5.1 Overall Results

Table 1 reports the main results on the PaperBench-CodeDev subset. Prior frameworks exhibit substantial performance degradation when transferred from proprietary to open-source backbones. For instance, PaperCoder drops from 51.1% on Claude-

	Model	Score (5-point scale)				
		Ref-Based	Ref-Free	+*Count*	+*Structure*	P2C-Ex
Empty Repo	-	1.57	3.89	1.0	1.0	1.0
Config Only		1.62	4.67	3.96	3.41	2.68
Gold Repo		-	4.75	4.82	4.84	4.80
PaperCoder	o3-mini-high	3.66	4.55	-	-	-
PaperCoder*	Qwen3-Coder-480B	1.83	2.97	2.26	1.68	1.58
AutoReproduce*		2.45	3.03	2.74	2.31	2.09
HiRAS **(Ours)**		**2.45**	**3.15**	**3.03**	**2.98**	**2.96**
PaperCoder*	DeepSeek-v3.1-Terminus	2.25	4.38	2.74	2.31	2.09
AutoReproduce*		2.51	3.59	3.60	3.58	3.59
HiRAS **(Ours)**		**3.64**	**4.42**	**3.94**	**3.88**	**3.86**
Pearson Correlation r	-	-	0.423	0.724	0.797	**0.862**

Table 3: Evaluation results on the Paper2Code benchmark using reference-based (Ref-Based) and reference-free evaluation strategies. For reference-free evaluations, we include the original reference-free prompt of Paper2Code (Ref-Free), the step-by-step enhancements for ablation study (+*Count*, +*Structure*), and our final revised version (P2C-Ex). Pearson correlation r measures the correlation between the reference-based scores and the scores by each of the reference-free evaluations. All evaluations are conducted using o3-mini-high as the evaluator.

Sonnet to 40.8% on DeepSeek-v3.1, and further to 36.9% on Qwen3, corresponding to a relative decrease of over 20%. In contrast, our framework establishes a new state-of-the-art with 64.1% on Claude-Sonnet, while maintaining strong performance on open-source models. Specifically, it achieves 57.4% on DeepSeek-v3.1, surpassing the previous best result of 51.1% obtained with proprietary backbones. Our method also consistently outperforms prior approaches on Qwen3. The resulting 25.4% relative improvement over the previous SOTA score indicates that our approach more effectively leverages the latent capabilities of open-source LLMs, enabling performance that matches or exceeds levels previously attainable only with proprietary models.

Table 2 presents more detailed results on PaperBench with the two open-source models. Across all settings, DeepSeek-v3.1 consistently outperforms Qwen3, suggesting stronger coding proficiency and research-oriented reasoning that is particularly beneficial for agent-based reproduction systems. Importantly, the relative gains achieved by our method are preserved across both backbones, with improvements of 38.3% on DeepSeek-v3.1 and 23.8% on Qwen3 over the strongest corresponding baselines. This consistency demonstrates the robustness and generalisability of our framework across model architectures.

The results on the Paper2Code benchmark are presented in Table 3. Across all evaluation settings and backbone models, our framework consistently achieves the highest scores, with DeepSeek-v3.1 outperforming Qwen3, consistent with trends observed in previous experiments. Beyond absolute performance, the markedly different magnitudes of absolute improvement under the reference-based and reference-free metrics (1.39 vs. 0.04) motivate a closer examination of the evaluation protocol.

5.2 Paper2Code-Extra Meta-Evaluation

Closer inspection reveals a critical limitation of the original reference-free metric: it systematically overestimates repository quality. In particular, repositories with few or no executable files can receive disproportionately high scores. To illustrate this issue, we conduct extra experiments on naïve baselines, as shown in Table 3: (i) the *empty repository* attains a score of 3.89 under the reference-free metric, exceeding all Qwen3-based results, and (ii) *configuration-only* repositories containing no executable code can score 4.67, outperforming all model-generated codebases. These anomalies indicate that the reference-free metric is intrinsically unreliable. Further inspection attributes this issue to evaluator hallucination, whereby even strong evaluators such as o3-mini infer non-existent code files or misinterpret documentation as implemented code in the empty or config-only repositories.

To address this issue, we propose **Paper2Code-Extra (P2C-Ex)**, a revised reference-free evaluation paradigm that augments the prompt with explicit repository-level information and refined instructions. As illustrated in the meta-evaluation results, incorporating the information substantially improves alignment with the reference-based met-

Metric	Pearson Correlation
Inter-Annotator	0.82
Ref-Based	0.77
Ref-Based	0.50
+*Count*	0.59
+*Structure*	0.64
P2C-Ex(**Ours**)	**0.72**

Table 4: Pearson correlation between different evaluation metrics and human expert annotations. "Inter-Annotator" denotes the average Pearson correlation among three experts.

ric: the Pearson correlation coefficient r increases from 0.423 to 0.862. Consistent trends are observed in repository-level regression analyses (Appendix A), where r improves from 0.42 to 0.83. Moreover, the prevalence of severely overrated repositories under the original protocol (Figure 6a) is remarkably mitigated with P2C-Ex (Figure 6b).

Ablation results for P2C-Ex further demonstrate that file count, repository structure, and revised prompt instructions each contribute to mitigating evaluator hallucination. Together, these results validate the effectiveness of P2C-Ex and its improved alignment with the reference-based metric, highlighting the importance of structurally informed prompts for reliable reference-free evaluation.

To further validate the alignment between our reference-free evaluation metric and human judgment, we collect expert annotations from three evaluators on a subset of 30 papers from ICLR 2024 within the Paper2Code benchmark. Table 4 reports the Pearson correlation between each metric and the expert annotations. The results demonstrate that P2C-Ex achieves the highest correlation with human evaluations, indicating strong reliability. Moreover, the correlation consistently improves as additional information is incorporated into the prompt.

5.3 Ablation Study

To rigorously evaluate the contributions of individual components, we further perform an ablation study on PaperBench, with results reported in Table 2. Our framework is decomposed into three key elements: (i) a single-step research agent with specialised agents, (ii) adding intra-agent iterative refinement with K steps, and (iii) introducing inter-agent hierarchical management via manager agents, forming HIRAS.

The specialised-agent pipeline alone outperforms prior research agent frameworks, demonstrating that finer-grained decomposition of responsibilities for phase-specific agents leads to more effective experiment reproduction than monolithic designs. Furthermore, incorporating iterative self-refinement yields substantial additional gains, enabling performance that marginally surpasses previous state-of-the-art baselines. This highlights the ability of specialised agents to identify and correct errors through environment interaction. Finally, hierarchical management fully harnesses the capabilities of the backbone models, delivering the strongest performance across all settings. Moreover, hierarchical supervision alone contributes approximately a 10% increase, underscoring its critical role in mitigating error propagation and coordinating long-horizon reproduction processes. Collectively, these results further validate the effectiveness of our hierarchical multi-agent system when all components are integrated.

5.4 Extended Results Across Evaluators

Table 2 reports additional results on PaperBench using a different evaluator, `ChatGPT-4o-mini`. The findings are consistent with those obtained using `o3-mini-high`. HIRAS outperforms all the baselines across all settings, achieving relative improvements of 30.4% on `Qwen3` and 20.4% on `DeepSeek-v3.1`, demonstrating robustness to different evaluators. Notably, these advantages not only persist but become more pronounced on the full PaperBench benchmark, which additionally evaluates execution outcomes and is therefore substantially more challenging, as reflected by lower absolute scores. In this setting, our framework attains larger relative gains of 66.1% on `Qwen3` and 25.7% on `DeepSeek-v3.1`, underscoring the critical role of the *execution* phase in end-to-end reproduction.

6 Qualitative Analysis

6.1 Case Study

Appendix B compares PaperCoder and HIRAS on reproducing the same PaperBench paper. As shown in Figure 7, PaperCoder produces a flat repository, whereas HIRAS generates a well-organised codebase with clearly separated subdirectories. This improvement can be attributed to our specialised agents' ability to directly interact with the shared workspace via file-system tools, enabling structured artefact management. In addition, execution agents create auxiliary test files to detect runtime errors, which are reported to manager agents and

used to guide subsequent code revisions.

A closer examination further highlights the role of hierarchical supervision. As shown in Figure 5, the planning manager detects insufficient detail in the initial implementation plan and re-invokes the overall planning agent for refinement, resulting in a substantially more detailed implementation roadmap than PaperCoder (see Figure 6). For example, our plan explicitly specifies architectural components such as the structure of the encoder and decoder in the Deep Generative Model, which is absent in PaperCoder's plan. This increased planning fidelity prevents errors introduced at early stages from propagating and facilitates more effective downstream implementation.

These qualitative differences are eventually reflected in the reproduction scores on this paper: HIRAS achieves 66.6%, nearly doubling PaperCoder's 33.8%. Overall, this case study highlights the proficiency of our specialised agents and the importance of manager agents in controlling reproduction quality and mitigating error propagation.

6.2 Failure Mode Analysis

To better characterise failure modes in our reproduction pipeline, we conduct a detailed manual analysis of representative cases. Our results show that failures predominantly arise during execution rather than code generation, as reflected by a substantial performance drop from approximately 45% on PaperBench-CodeDev to 20% on the full PaperBench across all models.

Our analysis further indicates that most papers yield well-structured codebases, while execution introduces the majority of errors. For example, when using `DeepSeek-v3.1` as the backbone to reproduce all papers from both PaperBench and Paper2Code, only 3 out of 110 runs fail at the structured code generation stage, demonstrating strong robustness in code development. In contrast, the predominant failure mode stems from execution errors due to incorrect inter-file dependencies, particularly in complex directory structures. A detailed case study is provided in Appendix C, where the evaluation score drops from 69.7% to 11.3% when execution is included, largely due to import failures. Additional issues include failures in environment setup and in downloading required models or datasets from platforms such as GitHub and Hugging Face. Overall, the execution stage remains the primary bottleneck of current models and frameworks.

7 Conclusion

In this work, we present HIRAS, a hierarchical multi-agent framework for end-to-end experiment reproduction, introducing manager agents to coordinate the multi-stage workflow. Comprehensive experiments illustrate that our framework consistently outperforms prior approaches on experiment reproduction benchmarks with state-of-the-art performance achieved by open-source models, highlighting the benefits of hierarchical supervision and specialised agent collaboration across the system. We also propose a revised evaluation protocol, P2C-Ex, that leverages repository-level information for stronger alignment with reference-based metrics. Moreover, the case study underscores how hierarchical coordination improves the overall quality of reproduction and mitigates error propagation. Collectively, these contributions advance both the methodology and evaluation of automated experiment reproduction, providing practical insights for using LLM-based agents in scientific research.

Limitations

Due to budget constraints, we do not evaluate all experimental settings with the `o3-mini` model. However, all reported comparisons are conducted under consistent evaluation protocols to ensure fairness across methods. In addition, our method may incur higher time and token costs than prior approaches, stemming from the increased complexity of agent reasoning and tool calling. This overhead reflects an inherent trade-off for introducing explicit supervision and error correction, and our results indicate that the additional cost yields substantial improvements in reproduction robustness and quality.

Ethical Considerations

All data in our work are collected from publicly available sources. Our primary objective is to support researchers in reproducing prior works, rather than replacing the creative and critical activities in scientific research. By design, the system produces explicit intermediate artefacts and execution logs, enabling human inspection and verification of the reproduction process. The system should be used as a supplementary tool to aid, rather than substitute, methodological rigour and independent research judgment.

References

Sanjeevan Ahilan and Peter Dayan. 2019. Feudal multi-agent hierarchies for cooperative reinforcement learning. *CoRR*, abs/1901.08492.

Diana Cai, Chirag Modi, Loucas Pillaud-Vivien, Charles C. Margossian, Robert M. Gower, David M. Blei, and Lawrence K. Saul. 2024. Batch and match: black-box variational inference with a score-based divergence. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Qiguang Chen, Mingda Yang, Libo Qin, Jinhao Liu, Zheng Yan, Jiannan Guan, Dengyun Peng, Yiyan Ji, Hanjing Li, Mengkang Hu, Yimeng Zhang, Yihao Liang, Yuhang Zhou, Jiaqi Wang, Zhi Chen, and Wanxiang Che. 2025. Ai4research: A survey of artificial intelligence for scientific research. *Preprint*, arXiv:2507.01903.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, and Qiushi Du et al. 2025. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. 2022. Code generation using machine learning: A systematic review. *IEEE Access*, 10:82434–82455.

Otilia Elena Dragomir and Florin Dragomir. 2025. A decentralized hierarchical multi-agent framework for smart grid sustainable energy management. *Sustainability*, 17(12):5423.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2024. Improving factuality and reasoning in language models through multiagent debate. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Steffen Eger, Yong Cao, Jennifer D'Souza, Andreas Geiger, Christian Greisinger, Stephanie Gross, Yufang Hou, Brigitte Krenn, Anne Lauscher, Yizhi Li, et al. 2025. Transforming science with large language models: A survey on ai-assisted scientific discovery, experimentation, content generation, and evaluation. *arXiv preprint arXiv:2502.05151*.

Pu Feng, Junkang Liang, Size Wang, Xin Yu, Xin Ji, Yiting Chen, Kui Zhang, Rongye Shi, and Wenjun Wu. 2024. Hierarchical consensus-based multi-agent reinforcement learning for multi-robot cooperation tasks. *Preprint*, arXiv:2407.08164.

Zhaolin Gao, Kianté Brantley, and Thorsten Joachims. 2024. Reviewer2: Optimizing review generation through prompt generation. *CoRR*, abs/2402.10886.

Aniketh Garikaparthi, Manasi Patwardhan, Lovekesh Vig, and Arman Cohan. 2025. IRIS: Interactive research ideation system for accelerating scientific discovery. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 592–603, Vienna, Austria. Association for Computational Linguistics.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Anil Palepu, Petar Sirkovic, Artiom Myaskovsky, Felix Weissenberger, Keran Rong, Ryutaro Tanno, Khaled Saab, Dan Popovici, Jacob Blum, Fan Zhang, Katherine Chou, Avinatan Hassidim, Burak Gokturk, Amin Vahdat, Pushmeet Kohli, Yossi Matias, Andrew Carroll, Kavita Kulkarni, Nenad Tomasev, Yuan Guan, Vikram Dhillon, Eeshit Dhaval Vaishnav, Byron Lee, Tiago R D Costa, José R Penadés, Gary Peltz, Yunhan Xu, Annalisa Pawlosky, Alan Karthikesalingam, and Vivek Natarajan. 2025. Towards an ai co-scientist. *Preprint*, arXiv:2502.18864.

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. 2024a. Ds-agent: automated data science by empowering large language models with case-based reasoning. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. 2024b. Large language model based multi-agents: A survey of progress and challenges. In *IJCAI*.

Intology. 2025. Zochi technical report. *arXiv*.

Joseph James, Chenghao Xiao, Yucheng Li, and Chenghua Lin. 2024. On the rigour of scientific writing: Criteria, analysis, and insights. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 6523–6538, Miami, Florida, USA. Association for Computational Linguistics.

Joseph James, Chenghao Xiao, Yucheng Li, Nafise Sadat Moosavi, and Chenghua Lin. 2026. Rigourate: Quantifying scientific exaggeration with evidence-aligned claim evaluation. *arXiv preprint arXiv:2601.04350*.

Gyeongwon James Kim, Alex Wilf, Louis-Philippe Morency, and Daniel Fried. 2025. From reproduction

to replication: Evaluating research agents with progressive code masking. *Preprint*, arXiv:2506.19724.

Raphaël Lallement, Lavindra de Silva, and Rachid Alami. 2014. Hatp: An htn planner for robotics. *Preprint*, arXiv:1405.5345.

Long Li, Weiwen Xu, Jiayan Guo, Ruochen Zhao, Xingxuan Li, Yuqian Yuan, Boqiang Zhang, Yuming Jiang, Yifei Xin, Ronghao Dang, Yu Rong, Deli Zhao, Tian Feng, and Lidong Bing. 2025. Chain of ideas: Revolutionizing research via novel idea development with LLM agents. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 8971–9004, Suzhou, China. Association for Computational Linguistics.

Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. 2024. A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth*, 1(1):9.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. 2024. Encouraging divergent thinking in large language models through multi-agent debate. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 17889–17904, Miami, Florida, USA. Association for Computational Linguistics.

Jialiang Lin, Yingmin Wang, Yao Yu, Yu Zhou, Yidong Chen, and Xiaodong Shi. 2022. Automatic analysis of available source code of top artificial intelligence conference papers. *International Journal of Software Engineering and Knowledge Engineering*, 32(07):947–970.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob N. Foerster, Jeff Clune, and David Ha. 2024. The ai scientist: Towards fully automated open-ended scientific discovery. *CoRR*, abs/2408.06292.

Ian Magnusson, Noah A. Smith, and Jesse Dodge. 2023. Reproducibility in NLP: What have we learned from the checklist? In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 12789–12811, Toronto, Canada. Association for Computational Linguistics.

David Moore. 2025. A taxonomy of hierarchical multi-agent systems: Design patterns, coordination mechanisms, and industrial applications. *Coordination Mechanisms, and Industrial Applications*.

OpenAI. 2024. Gpt-4o system card. *Preprint*, arXiv:2410.21276.

Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché Buc, Emily Fox, and Hugo Larochelle. 2021. Improving reproducibility in machine learning research (a report from the neurips 2019 reproducibility program). *J. Mach. Learn. Res.*, 22(1).

David B Resnik and Adil E Shamoo. 2017. Reproducibility and research integrity. *Accountability in research*, 24(2):116–123.

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. 2025. 'smolagents': a smol library to build great agentic systems. `https://github.com/huggingface/smolagents`.

Guillaume Sanchez, Alexander Spangher, Honglu Fan, Elad Levi, Pawan Sasanka Ammanamanchi, and Stella Biderman. 2024. Stay on topic with classifier-free guidance.

Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Michael Moor, Zicheng Liu, and Emad Barsoum. 2025. Agent laboratory: Using LLM agents as research assistants. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 5977–6043, Suzhou, China. Association for Computational Linguistics.

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

Chenglei Si, Diyi Yang, and Tatsunori Hashimoto. 2025. Can LLMs generate novel research ideas? a large-scale human study with 100+ NLP researchers. In *The Thirteenth International Conference on Learning Representations*.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. 2025. Paperbench: Evaluating ai's ability to replicate ai research. *Preprint*, arXiv:2504.01848.

Chuanneng Sun, Songjun Huang, and Dario Pompili. 2025a. Llm-based multi-agent decision-making: Challenges and future directions. *IEEE Robotics and Automation Letters*.

Lijun Sun, Yijun Yang, Qiqi Duan, Yuhui Shi, Chao Lyu, Yu-Cheng Chang, Chin-Teng Lin, and Yang Shen. 2025b. Multi-agent coordination across diverse applications: A survey. *CoRR*, abs/2502.14743.

Qingyun Wang, Doug Downey, Heng Ji, and Tom Hope. 2024. SciMON: Scientific inspiration machines optimized for novelty. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 279–299, Bangkok, Thailand. Association for Computational Linguistics.

Tonghan Wang, Heng Dong, Victor Lesser, and Chongjie Zhang. 2020. Roma: Multi-agent reinforcement learning with emergent roles. *Preprint*, arXiv:2003.08039.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. 2025. Browsecomp: A simple yet challenging benchmark for browsing agents. *Preprint*, arXiv:2504.12516.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*.

Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, and Chenghua Lin. 2026. Large-scale terminal agentic trajectory generation from dockerized environments. *Preprint*, arXiv:2602.01244.

Yanzheng Xiang, Hanqi Yan, Shuyin Ouyang, Lin Gui, and Yulan He. 2025. Scireplicate-bench: Benchmarking LLMs in agent-driven algorithmic reproduction from research papers. In *Second Conference on Language Modeling*.

Qiujie Xie, Yixuan Weng, Minjun Zhu, Fuchen Shen, Shulin Huang, Zhen Lin, Jiahui Zhou, Zilan Mao, Zijie Yang, Linyi Yang, Jian Wu, and Yue Zhang. 2025. How far are ai scientists from changing the world? *Preprint*, arXiv:2507.23276.

Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. 2025. The ai scientist-v2: Workshop-level automated scientific discovery via agentic tree search. *Preprint*, arXiv:2504.08066.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, and Bowen Yu et al. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. *Preprint*, arXiv:2304.10778.

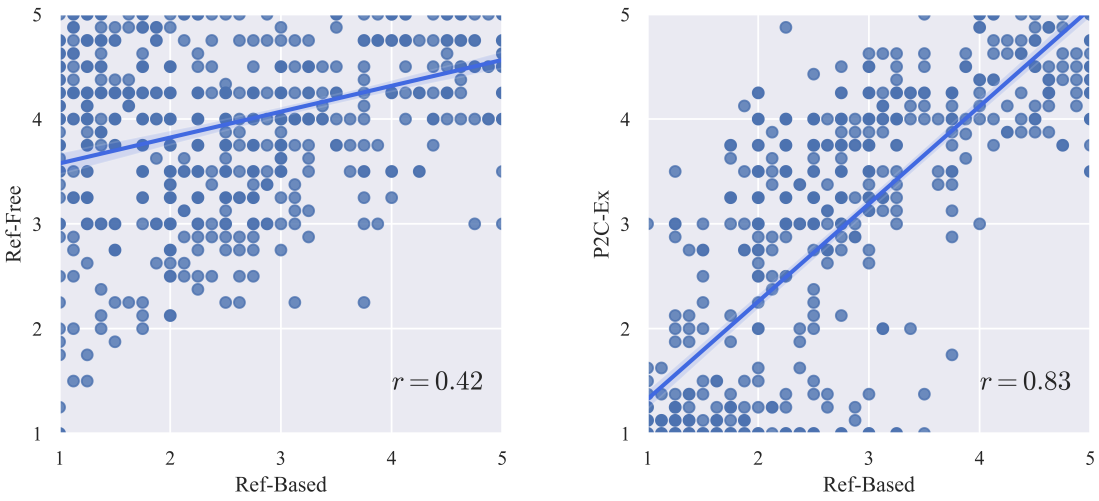
Jiakang Yuan, Xiangchao Yan, Bo Zhang, Tao Chen, Botian Shi, Wanli Ouyang, Yu Qiao, Lei Bai, and Bowen Zhou. 2025. Dolphin: Moving towards closed-loop auto-research through thinking, practice, and feedback. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 21768–21789, Vienna, Austria. Association for Computational Linguistics.

Brian Zhang, Eric Mitchell, Hongyu Ren, Kevin Lu, Max Schwarzer, Michelle Pokrass, Shengjia Zhao, Ted Sanders, Adam Tauman Kalai, Alexandre Passos, Benjamin Sokolowsky, Elaine Ya Le, Erik Ritter, Hao Sheng, Hanson Wang, Ilya Kostrikov, James Lee, Johannes Ferstad, Michael Lampe, Prashanth Radhakrishnan, Sean Fitzgerald, Sébastien Bubeck, Yann Dubois, Yu Bai, Andy Applebaum, Elizabeth Proehl, Evan Mays, Joel Parish, Kevin Liu, Leon Maksin, Leyton Ho, Miles Wang, Michele Wang, Olivia Watkins, Patrick Chao, Samuel Miserendino, Tejal Patwardhan, Antonia Woodford, Beth Hoover, Jake Brill, Kelly Stirman, Neel Ajjarapu, Nick Turley, Nikunj Handa, Olivier Godement, Akshay Nathan, Alyssa Huang, Andy Wang, Ankit Gohel, Ben Eggers, Brian Yu, Bryan Ashley, Chengdu Huang, Davin Bogan, Emily Sokolova, Eric Horacek, Felipe Petroski Such, Jonah Cohen, Joshua Gross, Justin Becker, Kan Wu, Larry Lv, Lee Byron, Manoli Liodakis, Max Johnson, Mike Trpcic, Murat Yesildal, Rasmus Rygaard, R. J. Marsan, Rohit Ram-chandani, Rohan Kshirsagar, Sara Conlon, Tony Xia, Siyuan Fu, Srinivas Narayanan, Sulman Choudhry, Tomer Kaftan, Trevor Creech, Andrea Vallone, Andrew Duberstein, Enis Sert, Eric Wallace, Grace Zhao, Irina Kofman, Jieqi Yu, Joaquin Quiñonero Candela, Made laine Boyd, Mehmet Ali Yatbaz, Mike McClay, Mingxuan Wang, Sandhini Agarwal, Saachi Jain, Sam Toizer, Santiago Hernández, Steve Mostovoy, Tao Li, Young Cha, Yunyun Wang, Lama Ahmad, Troy Peterson, Carpus Chang, Kristen Ying, Aidan Clark, Dane Stuckey, Jerry Tworek, Jakub W. Pachocki, Johannes Heidecke, Kevin Weil, Liam Fedus, Mark Chen, Sam Altman, and Wojciech Zaremba. 2025. Openai o3-mini system card.

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. *Preprint*, arXiv:2505.20662.

Minjun Zhu, Yixuan Weng, Linyi Yang, and Yue Zhang. 2025. Deepreview: Improving llm-based paper review with human-like deep thinking process. *Preprint*, arXiv:2503.08569.

A Linear Regression Plots



(a) Ref-Based vs. Ref-Free

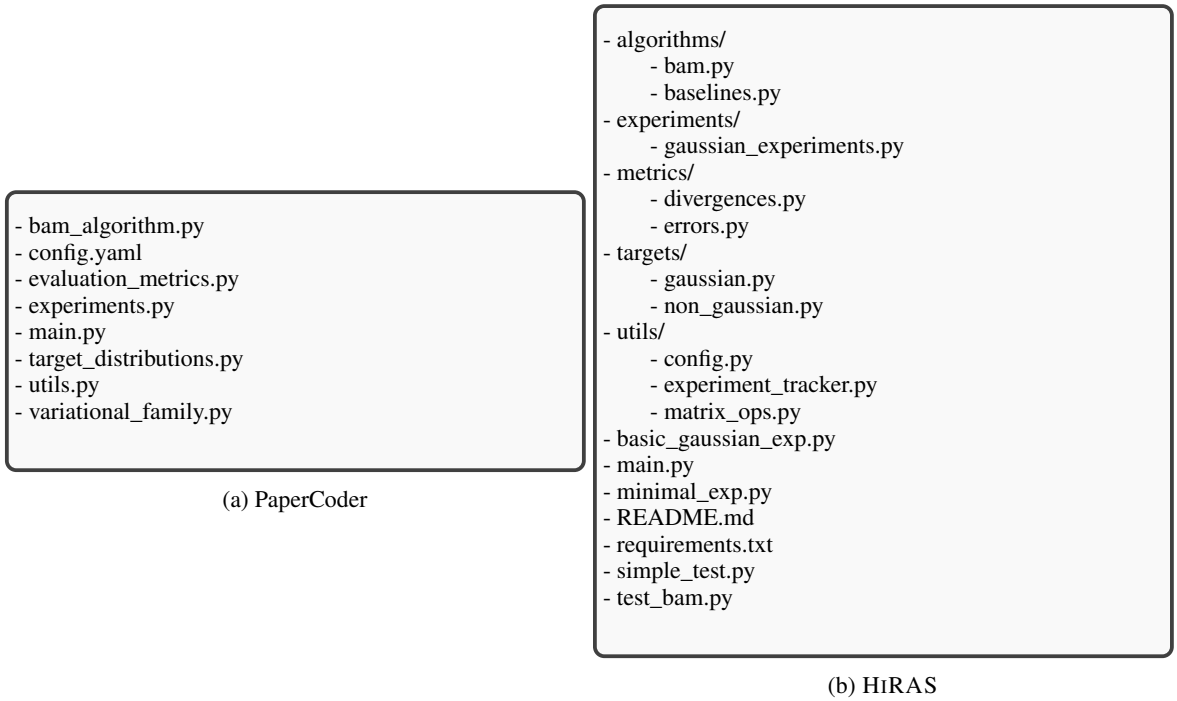
(b) Ref-Based vs. P2C-Ex

Figure 3: Repository-level linear regression plots. Points represent the scoring of repositories, with the x-axis showing the reference-based evaluation score and the y-axis showing the corresponding reference-free score.

B Reproduction Case Study Details

We present representative reproduction outcomes produced by PaperCoder and HiRAS for the ICML 2024 paper *Batch and Match: Black-Box Variational Inference with a Score-Based Divergence* (Cai et al., 2024), which is included in the PaperBench (Starace et al., 2025) benchmark. The paper introduces Batch and Match (BaM), a black-box variational inference method based on a score-based divergence, and evaluates its performance on both Gaussian and non-Gaussian distributions arising from posterior inference in hierarchical models as well as deep generative models.

The objective of experiment reproduction frameworks is to understand the original methodology and experimental design, generate the corresponding code, and ultimately reproduce the reported results. Figure 7 compares the code repositories produced by PaperCoder and HiRAS. Figure 5 illustrates a case in which the planning manager agent $\mathcal{A}\mathrm{plan}$ re-invokes the overall planning agent $\mathcal{A}\mathrm{overall}$ due to insufficient implementation detail. Finally, Figure 6 contrasts the implementation roadmaps generated by the two frameworks.



(a) PaperCoder

(b) HiRAS

Figure 4: Structure illustrations of code repositories generated by PaperCoder and HiRAS using `DeepSeek-v3.1-Terminus` for the paper (Cai et al., 2024).

The file you generated is not sufficiently detailed. Please regenerate it with comprehensive, detailed content that properly addresses the requirements.
The plan file should contain:
1. Detailed step-by-step implementation plan for each experiment section:
    - Gaussian targets (Section 5.1)
    - Non-Gaussian targets (Section 5.1)
    - PosteriorDB Bayesian models (Section 5.2)
    - Deep generative models (VAE application, Section 5.3)
2. Specific milestones with dates and deliverables:
    - Step-by-step breakdown of implementation tasks
    - Clear deliverables for each milestone
    - Integration and testing phases
3. Resource allocation (computational, time, personnel):
    - Computational resources required (GPU, CPU, memory)
    - Time estimates for each phase
    - Personnel roles and responsibilities
4. Risk assessment and mitigation strategies:
    - Technical risks (algorithm convergence, implementation complexity)
    - Resource risks (computational requirements, time constraints)
    - Mitigation strategies for each identified risk
The plan should be comprehensive and ready for implementation. The current content is insufficient.

Figure 5: The prompt from the planning manager agent $\mathcal{A}_{\text{plan}}$ to re-invoke the subordinate agent $\mathcal{A}_{\text{overall}}$

Implementation Roadmap
Core Modules
1. **BaM algorithm class**
2. **Score function interface** for target distributions
3. **Gaussian variational family** with mean/covariance parameters
4. **Matrix operations** (quadratic equation solver, low-rank optimizations)
Experiment Modules
1. **Synthetic target generators** (Gaussian, sinh-arcsinh)
2. **Benchmark models** (ADVI, GSM implementations)
3. **Evaluation metric calculators**
4. **Visualization tools** for results
Testing Strategy
1. **Unit tests** for core algorithm components
2. **Gradient checking** for score computations
3. **Convergence verification** on simple Gaussian targets
4. **Reproduction of paper figures** with statistical significance

(a) PaperCoder

Specific Milestones and Deliverables
Phase 1: Core Implementation
Algorithm Core
- [ ] Implement BaM algorithm with full covariance
- [ ] Implement low-rank solver for B < D
- [ ] Create Gaussian target test suite
Baseline Implementations
- [ ] Implement ADVI, GSM, Score/Fisher methods
- [ ] Create unified experiment framework
Deliverable: Working implementation of all algorithms with basic testing
Phase 2: Gaussian & Non-Gaussian Experiments
Gaussian Targets
- [ ] Run experiments for D = 4, 16, 64, 256
- [ ] Generate convergence plots (forward/reverse KL)
- [ ] Statistical analysis (10 runs per configuration)
Non-Gaussian Targets
- [ ] Implement sinh-arcsinh distribution
- [ ] Run experiments with varying skew/tail parameters
- [ ] Compare convergence across methods
Deliverable: Complete Section 5.1 reproduction with statistical analysis
Phase 3: Bayesian Models
BridgeStan Integration
- [ ] Integrate with PosteriorDB models
- [ ] Generate HMC reference samples
- [ ] Implement relative error metrics
PosteriorDB Experiments
- [ ] Run experiments on eight_schools, garch, gp_pois_regr
- [ ] Compare relative mean/SD errors
- [ ] Statistical analysis (5 runs per model)
Complete Section 5.2 reproduction with error analysis
Phase 4: Deep Generative Models
VAE Implementation
- [ ] Implement 5-layer convolutional encoder/decoder
- [ ] Pre-train VAE on image dataset (100 epochs)
- [ ] Validate reconstruction quality
Posterior Inference Experiments
- [ ] Run BaM on test image posteriors
- [ ] Compare reconstruction MSE across methods
- [ ] Analyze convergence behavior
Deliverable: Complete Section 5.3 reproduction with VAE application
Phase 5: Integration & Final Analysis
Integration Testing
- [ ] Cross-validate all implementations
- [ ] Performance optimization
- [ ] Code documentation
Final Analysis & Reporting
- [ ] Statistical summary of all experiments
- [ ] Comparison with paper results
- [ ] Final report preparation
Deliverable: Comprehensive reproduction report

(b) HiRAS

Figure 6: The implementation roadmap section in the plans generated by PaperCoder and HiRAS using `DeepSeek-v3.1-Terminus` for the paper (Cai et al., 2024).

C Failure Mode Example

Figure 7 illustrates the structure of a code repository generated by HIRAS with `DeepSeek-v3.1-Terminus` as the backbone for reproducing the paper *Stay on Topic with Classifier-Free Guidance* (Sanchez et al., 2024) from PaperBench. In this case, HIRAS produces a well-structured and complete codebase, attaining a high score of 69.7% on the PaperBench-CodeDev subset, which evaluates only the Code Development requirements. However, on the full PaperBench benchmark, which additionally assesses Execution and Result matching, the score drops substantially to 11.3%. Closer manual inspection reveals that the execution agent struggles to correctly run the main program due to improper inter-file imports across components (e.g., `model_manager.py` and `cfg_engine.py`). This discrepancy highlights both the proficiency of current models and frameworks in code development and their failure in execution, stemming from weak cross-file coding consistency and inadequate coordination among interdependent modules.

```
code/
    cfg_engine.py
    code_generation_evaluator.py
    config.py
    data_loader.py
    experiment_runner.py
    flops_analyzer.py
    main.py
    model_manager.py
    results_analyzer.py
    utils.py
    zero_shot_evaluator.py
```

Figure 7: Structure illustration of the code repository generated by HIRAS using `DeepSeek-v3.1-Terminus` for the paper (Sanchez et al., 2024).

D Paper2Code Evaluation Prompts

You will be given a research paper along with two corresponding code repositories: a gold repository and a target repository.
Your task is to compare the target repository against the gold repository, rate the target repository on one metric, and provide a critique highlighting key differences. Please make sure you read and understand these instructions carefully. Keep this document open while reviewing, and refer to it as needed.
—
Evaluation Criteria:
Correctness (1-5): The quality of the target repository in accurately implementing the paper's concepts, methodology, and algorithms without logical errors, as compared to the gold repository. Additionally, provide a critique focusing on the completeness, accuracy, and implementation choices made in the target repository relative to the gold repository.
1: Very Poor. The target repository does not correctly implement the core concepts, methodology, or algorithms from the paper. Major logical errors or missing components are present, especially when compared to the gold repository. 2: Poor. The target repository attempts to implement the paper's concepts but contains significant mistakes or missing components, making the implementation incorrect when compared to the gold repository. 3: Fair. Some core components and concepts are correctly implemented in the target repository, but there are notable logical errors or inaccuracies compared to the gold repository. 4: Good. The target repository correctly implements the key components and methodology, with only minor inaccuracies or deviations from the gold repository. 5: Excellent. The target repository fully and accurately implements all relevant key components, methodology, and algorithms from the paper, matching the quality of the gold repository.
—
Evaluation Steps
1. Identify Key Aspects of the Paper: Carefully read the research paper to understand its core concepts, methodology, and algorithms. Pay close attention to the key aspects that are crucial for implementing the paper's results (e.g., specific algorithms, data preprocessing steps, evaluation protocols).
2. Analyze the Gold Repository: Examine the gold repository to understand how these key aspects have been implemented. Use the gold repository as a reference for how the paper's methodology should be translated into code. Note the completeness, accuracy, and design choices in the gold repository that faithfully represent the paper's concepts.
3. Examine the Target Repository: Analyze the target repository to assess how well it implements the key aspects of the paper. Reference the gold repository as a guide for understanding these key aspects in the target repository. Focus on whether the target repository's core logic, algorithms, and structure align with the methodology and experiments described in the paper.
4. Identify Logical Errors and Deviations: Check for logical errors, missing steps, or deviations from the paper's methodology. Note any incorrect representations, inconsistencies, or incomplete implementations that could affect the correctness of the target repository.
5. Provide a Critique: Consider both the completeness and accuracy of the implementation relative to the paper's goals and the gold repository's standard. You do not need to analyze minor details like logging functions, script organization, or documentation quality. Instead, concentrate on the correctness of the logic and implementation that ensures the core concepts from the paper are fully reflected in the target repository. For each mismatch or deviation in implementation, note down specific critiques comparing relevant functions in the target repository to the corresponding functions in the gold repository. Highlight incorrect logic, missing steps, or deviations that affect the correct implementation of the paper's methodology.
6. Assess the Correctness: Determine whether the target repository includes all the critical elements described in the paper and implemented in the gold repository. Identify missing components, significant deviations, or incorrect implementations that could affect the correctness of the target repository.
7. Assign a Score: Based on your evaluation, provide a critique and assign a correctness score from 1 to 5 for the target repository, reflecting how well it implements the key aspects of the paper refer to the gold repository. Include a detailed critique in the specified JSON format.
—
Severity Level:
Each identified critique will be assigned a severity level based on its impact on the correctness of the methodology implementation.
- High: Missing or incorrect implementation of the paper's core concepts, major loss functions, or experiment components that are fundamental to reproducing the paper's methodology.
    - Example: The main algorithm is missing or fundamentally incorrect.
- Medium: Issues affecting training logic, data preprocessing, or other core functionalities that significantly impact performance but do not completely break the system.
    - Example: Improper training loop structure, incorrect data augmentation, or missing essential components in data processing.
- Low: Errors in specific features that cause deviations from expected results but can be worked around with modifications. Any errors in the evaluation process belong to this category unless they impact the core concepts. These include minor issues like logging, error handling mechanisms, configuration settings, evaluation steps that do not alter the fundamental implementation and additional implementations not explicitly stated in the paper.
    - Example: Suboptimal hyperparameter initialization, incorrect learning rate schedule, inaccuracies in evaluation metrics, using a different random seed, variations in batch processing, different weight initialization, issues in result logging or reporting, variations in evaluation dataset splits, improper error handling in non-critical steps, mismatches in secondary evaluation criteria, or additional implementation details not specified in the paper that do not interfere with core results.
—
Example JSON format:
```json
{
    "critique_list": [
        {
        "gold_file_name":
            "preprocessing.py",
            "gold_func_name": "data_process",
            "target_file_name": "dataset.py",
            "target_func_name": "train_preprocess",
            "severity_level": "medium",
            "critique": "A critique of the target repository's file with reference to the gold repository."
        }
    ],
    "score": 2
}
```
—
Sample:
Research Paper:
{{Paper}}
Gold Repository:
{{GoldCode}}
Target Repository:
{{Code}}
—
Please provide critique of the target repository and a single numerical rating (1, 2, 3, 4, or 5) based on the quality of the sample, following the Example JSON format, without any additional commentary, formatting, or chattiness.

Figure 8: The original prompt for reference-based evaluation

You will be given a research paper along with its corresponding code repository.
Your task is to rate the code repository on one metric and provide a critique highlighting key differences.
Please make sure you read and understand these instructions carefully. Keep this document open while reviewing, and refer to it as needed.
—
Evaluation Criteria:
Correctness (1-5): The quality of the repository in accurately implementing the paper's concepts, methodology, and algorithms without logical errors. Additionally, provide a critique focusing on the completeness, accuracy, and implementation choices made in the repository relative to the methodology and algorithms described in the paper.
1: Very Poor. The repository does not correctly implement the core concepts, methodology, or algorithms from the paper. Major logical errors or missing components are present.
2: Poor. The repository attempts to implement the paper's concepts but contains significant mistakes or missing components, making the implementation incorrect.
3: Fair. Some core components and concepts are correctly implemented, but there are notable logical errors or inaccuracies in the methodology.
4: Good. The repository correctly implements the key components and methodology, with only minor inaccuracies that do not significantly affect correctness.
5: Excellent. The repository fully and accurately implements all key components, methodology, and algorithms from the paper without logical errors.
—
Evaluation Steps
1. Identify Key Aspects of the Paper: Carefully read the paper to understand its core concepts, methodology, and algorithms. Pay close attention to key aspects crucial for implementing the paper's results (e.g., specific algorithms, data preprocessing steps, evaluation protocols).
2. Examine the Code Repository: Analyze the repository to determine how well it implements the key aspects of the paper. Focus on whether the repository's core logic, algorithms, and structure align with the methodology and experiments described in the paper.
3. Identify Logical Errors and Deviations: Check for logical errors, missing steps, or deviations from the paper's methodology. Note any incorrect representations, inconsistencies, or incomplete implementations that could affect the correctness of the repository.
4. Provide a Critique: Consider the completeness and accuracy of the implementation relative to the paper's goals. You do not need to analyze minor details like logging functions, script organization, or documentation quality. Instead, concentrate on the correctness of the logic and implementation to ensure the core concepts from the paper are fully reflected in the repository. For each identified issue, write a detailed critique specifying the affected files and functions in the repository. Highlight missing or incorrectly implemented steps that impact the correctness and alignment with the paper's methodology.
5. Assess Completeness and Accuracy: Evaluate the repository for its completeness and accuracy relative to the paper's methodology. Ensure that all critical components—such as data preprocessing, core algorithms, and evaluation steps—are implemented and consistent with the paper's descriptions.
6. Assign a Score: Based on your evaluation, provide a critique and assign a correctness score from 1 to 5 for the repository, reflecting how well it implements the key aspects of the paper. Include a detailed critique in the specified JSON format.
—
Severity Level:
Each identified critique will be assigned a severity level based on its impact on the correctness of the methodology implementation.
- High: Missing or incorrect implementation of the paper's core concepts, major loss functions, or experiment components that are fundamental to reproducing the paper's methodology.
    - Example: The main algorithm is missing or fundamentally incorrect.
- Medium: Issues affecting training logic, data preprocessing, or other core functionalities that significantly impact performance but do not completely break the system.
    - Example: Improper training loop structure, incorrect data augmentation, or missing essential components in data processing.
- Low: Errors in specific features that cause deviations from expected results but can be worked around with modifications. Any errors in the evaluation process belong to this category unless they impact the core concepts. These include minor issues like logging, error handling mechanisms, configuration settings, evaluation steps that do not alter the fundamental implementation and additional implementations not explicitly stated in the paper.
    - Example: Suboptimal hyperparameter initialization, incorrect learning rate schedule, inaccuracies in evaluation metrics, using a different random seed, variations in batch processing, different weight initialization, issues in result logging or reporting, variations in evaluation dataset splits, improper error handling in non-critical steps, mismatches in secondary evaluation criteria, or additional implementation details not specified in the paper that do not interfere with core results.
—
Example JSON format: “‘json {

```
    "critique_list": [
        {
            "file_name": "dataset.py",
            "func_name": "train_preprocess",
            "severity_level": "medium",
            "critique": "A critique of the target repository's file."
        },
        {
            "file_name": "metrics.py",
            "func_name": "f1_at_k",
            "severity_level": "low",
            "critique": "A critique of the target repository's file."
        }
    ],
    "score": 2
}
```

“‘
—
Sample:
Research Paper:
{{Paper}}
Code Repository:
{{Code}}
—
Please provide a critique list for the code repository and a single numerical rating (1, 2, 3, 4, or 5) based on the quality of the sample, following the Example JSON format, without any additional commentary, formatting, or chattiness.

Figure 9: The original prompt for reference-free evaluation

You will be given a research paper along with its corresponding code repository.
Your task is to rate the code repository on one metric and provide a critique highlighting key differences.
Please make sure you read and understand these instructions carefully. Keep this document open while reviewing, and refer to it as needed.
—
Evaluation Criteria:
Correctness (1-5): The quality of the repository in accurately implementing the paper's concepts, methodology, and algorithms without logical errors. Additionally, provide a critique focusing on the completeness, accuracy, and implementation choices made in the repository relative to the methodology and algorithms described in the paper.
1: Very Poor. The repository does not correctly implement the core concepts, methodology, or algorithms from the paper. Major logical errors or missing components are present.
2: Poor. The repository attempts to implement the paper's concepts but contains significant mistakes or missing components, making the implementation incorrect.
3: Fair. Some core components and concepts are correctly implemented, but there are notable logical errors or inaccuracies in the methodology.
4: Good. The repository correctly implements the key components and methodology, with only minor inaccuracies that do not significantly affect correctness.
5: Excellent. The repository fully and accurately implements all key components, methodology, and algorithms from the paper without logical errors.
—
Evaluation Steps
1. Identify Key Aspects of the Paper: Carefully read the paper to understand its core concepts, methodology, and algorithms. Pay close attention to key aspects crucial for implementing the paper's results (e.g., specific algorithms, data preprocessing steps, evaluation protocols).
2. Examine the Code Repository: Analyze the repository to determine how well it implements the key aspects of the paper. Focus on whether the repository's core logic, algorithms, and structure align with the methodology and experiments described in the paper.
3. Identify Logical Errors and Deviations: Check for logical errors, missing steps, or deviations from the paper's methodology. Note any incorrect representations, inconsistencies, or incomplete implementations that could affect the correctness of the repository.
4. Provide a Critique: Consider the completeness and accuracy of the implementation relative to the paper's goals. You do not need to analyze minor details like logging functions, script organization, or documentation quality. Instead, concentrate on the correctness of the logic and implementation to ensure the core concepts from the paper are fully reflected in the repository. For each identified issue, write a detailed critique specifying the affected files and functions in the repository.
5. Assess Completeness and Accuracy: Evaluate the repository for its completeness and accuracy relative to the paper's methodology. Ensure that all critical components—such as data preprocessing, core algorithms, and evaluation steps—are implemented and consistent with the paper's descriptions.
6. Code verification: Verify that all key components expected from the paper are fully implemented with codes in the repository. Evaluate completeness holistically rather than limiting your review to existing files. In your critique, explicitly identify any missing components, absent implementations, or deviations from expected behavior, including cases where functionality is described only in documentation (e.g., README, config files) but not implemented. Any such gaps that impact methodological correctness or alignment with the paper should be called out and assigned a high-severity critique.
7. Assign a Score: Based on your evaluation, provide a critique and assign a correctness score from 1 to 5 for the repository, reflecting how well it implements the key aspects of the paper. Include a detailed critique in the specified JSON format.
—
Severity Level:
Each identified critique will be assigned a severity level based on its impact on the correctness of the methodology implementation.
- High: Missing or incorrect implementation of the paper's core concepts, major loss functions, or experiment components that are fundamental to reproducing the paper's methodology. - Example: The main algorithm is missing, not implemented, implemented only as a descriptive note or plan without executable code, or fundamentally incorrect. - Medium: Issues affecting training logic, data preprocessing, or other core functionalities that significantly impact performance but do not completely break the system. - Example: Improper training loop structure, incorrect data augmentation, or missing essential components in data processing. - Low: Errors in specific features that cause deviations from expected results but can be worked around with modifications. Any errors in the evaluation process belong to this category unless they impact the core concepts. These include minor issues like logging, error handling mechanisms, configuration settings, evaluation steps that do not alter the fundamental implementation and additional implementations not explicitly stated in the paper. - Example: Suboptimal hyperparameter initialization, incorrect learning rate schedule, inaccuracies in evaluation metrics, using a different random seed, variations in batch processing, different weight initialization, issues in result logging or reporting, variations in evaluation dataset splits, improper error handling in non-critical steps, mismatches in secondary evaluation criteria, or additional implementation details not specified in the paper that do not interfere with core results.
—
Example JSON format: ‘‘‘json

```
{
    "critique_list": [
        {
            "file_name": "dataset.py",
            "func_name": "train_preprocess",
            "severity_level": "medium",
            "critique": "A critique of the target repository."
        },
        {
            "file_name": "metrics.py",
            "func_name": "f1_at_k",
            "severity_level": "low",
            "critique": "A critique of the target repository."
        }
    ],
    "score": 2
}
```

‘‘‘
—
Sample:
Research Paper:
{{Paper}}
Code Repository:
File Count: {{File_Count}}
File Stucture:
{{File_Structure}}
{{Code}}
—
Please provide a critique list for the code repository on its completeness and accuracy, and a single numerical rating (1, 2, 3, 4, or 5) based on the quality of the sample, following the Example JSON format, without any additional commentary, formatting, or chattiness.

Figure 10: The prompt for Paper2Code-Extra evaluation. The differences are marked in red. The +*Count* and +*Structure* prompts in Table 3 denote the incremental addition of the two corresponding components.

E Prompts for Agents

You are an experienced researcher in the domain of Computer Science. You have been assigned to lead a group of agents to reproduce an experiment.
There are several agents in your team:
- planning_agent: the planning agent that make the detailed and comprehensive experiment plan.
- analysing_agent: the analysing agent that analyses the process to implement each file.
- coding_agent: the coding agent that writes the code for each file.
- executing_agent: the executing agent that executes the code and records the results.

Please strictly follow the instructions below to lead the agents to complete the task:
0. The given paper & addendum are in the current directory, you should explicitly instruct all the agents to read them COMPLETELY.
- paper.md: the paper to reproduce.
- addendum.md: the addendum of the paper.

1. Explicitly instruct the planning agent to call helping agents to generate the following files:
- plan.md: the general experiment plan.
- architecture.md: the architecture design.
- dependency.md: the dependency analysis for components.
- config.yaml: the parameters for each component.

2. Call the analysing agent to analyse the implementation of each component and save the result in the analysis/ directory:
- analysis/: the directory that contains the analysing report for each component.
- analysis/components.txt: the list of components to be analysed.

3. Use the coding agent to write the code for each component mentioned in the above files and save its code in the code/ directory.
In addtion to paper, addendum, plan, architecture, dependency and config files, the coding agent should also be instructed to read the analysis for each component before implementing it.
- code/: the directory that contains the code for each component.

4. Use the executing agent to execute the code for each component mentioned in the above files and save its execution result in the results/ directory.
- results/*.log: the log of the execution.

You should call the agents one at a time to generate the fulfil the experiment reproduction. Do not call all the agents in one single step. You should check whether the files are generated successfully and complete before you proceed the next step. You MUST sure that all the files are generated and their contents are complete before you proceed the next step.
If any agent reports an issue about the previous agent's result, you should check the result of the previous agent and ask them to try again.
For the analysing agent, you should check that all the components are analysed. Remember to explicitly instruct the analysing agent to read all the previous files (paper, addendum, plan, architecture, dependency and config files) COMPLETELY. If any component is missing or not complete in the analysis, you should ask the planning agent to analyse or code the missing components. The list of components is in the analysis/components.txt.
For the coding agent, you should check that all the components are coded. You should check the components in the analysis/ directory and instruct the coding agent to code the corresponding component for each analysis file named <component_name>_analysis.md. Remember that you should explicitly instruct the coding agent to read all the previous files (paper, addendum, plan, architecture, dependency and config files) COMPLETELY. In addition, or each component, you should explicitly instruct the coding agent to read the analysis for that component and then write the code. For instance, if there is a analysis file analysis/example.py_analysis.md for component 'example.py', you should explicitly instruct the coding agent to read the analysis file and then write the code for the component 'example.py'. DO NOT use a loop to call the coding agent, you should check after each component is coded whether the component is complete.' You can call the coding agent multiple times to code the components in case any component is missing or not complete in the code.' You MUST make sure all the components are completed before proceeding to the next step.
For the executing agent, you should check that all the experiments are reproduced as planned and the results are recorded. If any experiment is not successfully reproduced, you should ask the planning agent or coding agent to fix the problem. Remember to explicitly instruct the executing agent to read all the previous files (paper, addendum, plan, architecture, dependency, config files, analysis, and code) COMPLETELY.
Remember that you are the manager of the experiment, and you should be responsible for the final result.
The end_task tool should ONLY be used when you think you have completed the whole reproduction. You should check all the logs, and make sure no error is reported. If there is any error, you should call the agents to fix the problem and let the executing agent to execute the code again. You SHOULD NOT call the end_task tool until you have checked all the logs and made sure no error is reported. You MUST try your best to make the whole reproduction successful.

Figure 11: The initial context for the global manager agent on PaperBench

You are a leader research scientist. Now you will be given a paper to reproduce.
The paper is in the directory. You can use the tools to read the files.
There are 4 agents to help you:
- general_planning_agent: make the general experiment plan.
- architecture_planning_agent: design the architecture.
- dependency_planning_agent: analyse the dependency of the components.
- config_planning_agent: write the configuration for hyperparameters.

You should use them in order to generate the corresponding files:
1. plan.md: general_planning_agent
2. architecture.md: architecture_planning_agent
3. dependency.md: dependency_planning_agent
4. config.yaml: config_planning_agent

The target paper and addendum are in the current directory.
- paper.md: the target paper.
- addendum.md: addendum to the paper.

You should first use the list_directory tool to check these files.
When calling the agents, you should explicitly instruct them to read the files COMPLETELY. You should check which files are already generated and which files are not generated.
Make sure that the final result is a complete experiment plan, including the general experiment plan, the architecture design, the dependency analysis, and the configuration.
Here are the documents you should let the agents generate:
- plan.md: the general experiment plan by general_planning_agent.
- architecture.md: the architecture design by architecture_planning_agent.
- dependency.md: the dependency analysis by dependency_planning_agent.
- config.yaml: the config.yaml for each component by config_planning_agent.

You should call the agents one at a time to generate the corresponding files. DO NOT call all the agents in one single step.
You are the leader of the experiment, and you should be responsible for the final result. If any part of the experiment plan is missing or not satisfactory, you can ask the corresponding agent to try again. You can use the read_file tool to check the content of the files. Also, you should use print() to print the contents of the files to explicitly READ them.
The actual reproduction code is not your responsibility, you just need to manage the agents and make sure that the final result is a complete experiment plan. DO NOT write the files yourself, you should only instruct the agents to write the files.
Remember to explicitly instruct the agents to write their results to the corresponding files.
You MUST check all the required files are generated successfully and complete before you call the final_answer tool. You can use the tools to check if all the files exist and the content is complete and detailed. Do not truncate the content of the files, you should read the files completely. Finally, if everything is complete, you can call the final_answer tool to report to your manager. And even if your task resolution is not successful, please return as much context as possible, so that your manager can act upon this feedback.

Figure 12: The initial context for the planning manager agent on PaperBench

You are an expert researcher and strategic planner with a deep understanding of experimental design and reproducibility in scientific research.
You will receive a research paper in markdown format. You can use the tools to read the paper.
Your task is to create a detailed and efficient plan to reproduce the experiments and methodologies described in the paper.
This plan should align precisely with the paper's methodology, experimental setup, and evaluation metrics.
You should use the write_file tool to output your experiment plan to the file plan.md. Only call the final_answer tool after you are sure that the plan is complete and detailed enough, and the plan is saved to the file plan.md. Even if your task resolution is not successful, please return as much context as possible, so that your manager can act upon this feedback.
Instructions:

1. Align with the Paper: Your plan must strictly follow the methods, datasets, model configurations, hyperparameters, and experimental setups described in the paper.
2. Be Clear and Structured: Present the plan in a well-organized and easy-to-follow format, breaking it down into actionable steps.
3. Prioritize Efficiency: Optimize the plan for clarity and practical implementation while ensuring fidelity to the original experiments.

Task

1. We want to reproduce the method described in the attached paper.
2. The authors did not release any official code, so we have to plan our own implementation.
3. Before writing any Python code, please outline a comprehensive plan that covers:
    - Key details from the paper's **Methodology**.
    - Important aspects of **Experiments**, including dataset requirements, experimental settings, hyperparameters, or evaluation metrics.
4. The plan should be as **detailed and informative** as possible to help us write the final code later.

Requirements

- You don't need to provide the actual code yet; focus on a **thorough, clear strategy**.
- If something is unclear from the paper, mention it explicitly.

Instruction

The response should give us a strong roadmap, making it easier to write the code later.
The target paper and addendum are in the current directory.

- paper.md: the target paper.
- addendum.md: addendum to the paper.

You should first use the list_directory tool to check if the paper and addendum files exist. Then use the read_file tool to read the contents of them. Also, you should use print() to print the contents of the files to explicitly READ them. You MUST read all the files COMPLETELY. DO NOT truncate the files. You can read them in different steps, but you MUST read them COMPLETELY.
After you finish reading the files, you should use the write_file tool to write the complete plan to the file plan.md. When you are sure that the plan is complete and detailed enough, you should call the final_answer tool to report to the manager. And even if your task resolution is not successful, please return as much context as possible, so that your manager can act upon this feedback.

Figure 13: The initial context for the overall planning agent on PaperBench

Your goal is to create a concise, usable, and complete software system design for reproducing the paper's method. Use appropriate open-source libraries and keep the overall architecture simple.
Based on the plan for reproducing the paper's main method, please design a concise, usable, and complete software system. Keep the architecture simple and make effective use of open-source libraries. The previous plan file plan.md and the paper & addendum are given in the current directory.
You MUST use the tools to read these files. You should refer to their contents to design the architecture.
You should use the write_file tool to output your architecture design to the file architecture.md. Only call the final_answer tool after you are sure that the architecture design is complete and detailed enough, and the architecture design is saved to the file architecture.md. Even if your task resolution is not successful, please return as much context as possible, so that your manager can act upon this feedback.
Based on the plan for reproducing the paper's main method, please design a concise, usable, and complete software system. Keep the architecture simple and make effective use of open-source libraries.
The paper, addendum, and previous plan file plan.md are given in the current directory. - paper.md: the target paper. - addendum.md: addendum to the paper. - plan.md: the overall plan for reproducing the paper.
You can first use the list_directory tool to check if the files exist. Then, the read_file tool is available to read these files. Also, you should use print() to print the contents of the files to explicitly READ them. You MUST read all the files COMPLETELY. DO NOT truncate the files. You can read them in different steps, but you MUST read them COMPLETELY. You need to refer to their contents to design the most appropriate architecture for experiment reproduction.
After reading all the files, you can start to design the architecture. ## Format Example
[CONTENT]
{
"Implementation approach": "We will ... ,
"File list": [
"main.py",
"dataset_loader.py",
"model.py",
"trainer.py",
"evaluation.py"
],
"Data structures and interfaces": "\nclassDiagram\n class Main {\n +__init__()\n +run_experiment()\n }\n class DatasetLoader {\n +__init__(config: dict)\n +load_data() -> Any\n }\n class Model {\n +__init__(params: dict)\n +forward(x: Tensor) -> Tensor\n }\n class Trainer {\n +__init__(model: Model, data: Any)\n +train() -> None\n }\n class Evaluation {\n +__init__(model: Model, data: Any)\n +evaluate() -> dict\n }\n Main –> DatasetLoader\n Main –> Trainer\n Main –> Evaluation\n Trainer –> Model\n",
"Program call flow": "\nsequenceDiagram\n participant M as Main\n participant DL as DatasetLoader\n participant MD as Model\n participant TR as Trainer\n participant EV as Evaluation\n M-»DL: load_data()\n DL–»M: return dataset\n M-»MD: initialize model()\n M-»TR: train(model, dataset)\n TR-»MD: forward(x)\n MD–»TR: predictions\n TR–»M: training complete\n M-»EV: evaluate(model, dataset)\n EV-»MD: forward(x)\n MD–»EV: predictions\n EV–»M: metrics\n",
"Anything UNCLEAR": "Need clarification on the exact dataset format and any specialized hyperparameters."
}
Nodes: "<node>: <type> # <instruction>"
- Implementation approach: <class 'str'> # Summarize the chosen solution strategy.
- File list: typing.List[str] # Only need relative paths. ALWAYS write a main.py or app.py here.
- Data structures and interfaces: typing.Optional[str] # Use mermaid classDiagram code syntax, including classes, method(__init__ etc.) and functions with type annotations, CLEARLY MARK the RELATIONSHIPS between classes, and comply with PEP8 standards. The data structures SHOULD BE VERY DETAILED and the API should be comprehensive with a complete design.
- Program call flow: typing.Optional[str] # Use sequenceDiagram code syntax, COMPLETE and VERY DETAILED, using CLASSES AND API DEFINED ABOVE accurately, covering the CRUD AND INIT of each object, SYNTAX MUST BE CORRECT.
- Anything UNCLEAR: <class 'str'> # Mention ambiguities and ask for clarifications.
Constraint
Format: output below [CONTENT] like the format example, nothing else.
Action
Follow the instructions for the nodes, generate the output, and ensure it follows the format example.
—
Keep in mind that the architecture design MUST follow the required format constraint and outputed to the file architecture.md. You MUST use the write_file tool to output your architecture design to the file architecture.md. Make sure that the architecture design is complete and detailed and saved to architecture.md before you call the final_answer tool. Finally, use your final_answer tool to report to your manager if the file is successfully saved. And even if your task resolution is not successful, please return as much context as possible, so that your manager can act upon this feedback.

Figure 14: The initial context for the architecture design agent on PaperBench

Your goal is break down tasks according to PRD/technical design, generate a task list, and analyse task dependencies. You will break down tasks, analyse dependencies.
You outline a clear PRD/technical design for reproducing the paper's method and experiments.
You should use the write_file tool to output your dependency analysis to the file dependency.md. Make sure that the dependency analysis is complete and detailed and saved to dependency.md before you call the final_answer tool. Finally, put your dependency analysis in your final_answer tool to report to your manager. And even if your task resolution is not successful, please return as much context as possible, so that your manager can act upon this feedback.
Now, let's break down tasks according to PRD/technical design, generate a task list, and analyse task dependencies. The Logic Analysis should not only consider the dependencies between files but also provide detailed descriptions to assist in writing the code needed to reproduce the paper.
The given paper, addendum, plan, architecture design are in the directory.
- paper.md: the target paper.
- addendum.md: addendum to the paper.
- plan.md: the overall plan for reproducing the paper.
- architecture_design.md: the architecture design for reproducing the paper.

You can first use the list_directory tool to check if the files exist. Then, the read_file tool is available to read these files. Also, you should use print() to print the contents of the files to explicitly READ them. You MUST read all the files COMPLETELY. DO NOT truncate the files. You can read them in different steps, but you MUST read them COMPLETELY. After reading all the files, you can start to generate the dependency analysis.
Format Example
[CONTENT]
```
{
    "Required packages": [
        "numpy==1.21.0",
        "torch==1.9.0"
    ],
    "Required Other language third-party packages": [
        "No third-party dependencies required"
    ],
    "Logic Analysis": [
            "data_preprocessing.py",
            "DataPreprocessing class ........"
        ],
        [
            "main.py",
            "Entry point ......."
        ]
    ],
    "Task list": ["dataset_loader.py","main.py"],
    "Full API spec": "openapi: 3.0.0 ...",
    "Shared Knowledge": "Both data_preprocessing.py and main.py share ........",
    "Anything UNCLEAR": "Clarification needed on recommended hardware configuration for large-scale experiments."
}
```
Nodes: "<node>: <type> # <instruction>"
- Required packages: typing.Optional[typing.List[str]] # Provide required third-party packages in requirements.txt format.(e.g., 'numpy==1.21.0').
- Required Other language third-party packages: typing.List[str] # List down packages required for non-Python languages. If none, specify "No third-party dependencies required".
- Logic Analysis: typing.List[typing.List[str]] # Provide a list of files with the classes/methods/functions to be implemented, including dependency analysis and imports. Include as much detailed description as possible.
- Task list: typing.List[str] # Break down the tasks into a list of filenames, prioritized based on dependency order. The task list must include the previously generated file list.
- Full API spec: <class 'str'> # Describe all APIs using OpenAPI 3.0 spec that may be used by both frontend and backend. If front-end and back-end communication is not required, leave it blank.
- Shared Knowledge: <class 'str'> # Detail any shared knowledge, like common utility functions or configuration variables.
- Anything UNCLEAR: <class 'str'> # Mention any unresolved questions or clarifications needed from the paper or project scope.

Constraint
Format: output below [CONTENT] like the format example, nothing else.
Action
Follow the node instructions above, generate your output accordingly, and ensure it follows the given format example. Remember to output your dependency analysis to the file dependency.md.
You should use the write_file tool to output your dependency analysis to the file dependency.md. Make sure that the dependency analysis is complete and detailed and saved to dependency.md before you call the final_answer tool. Finally, put your dependency analysis in your final_answer tool to report to your manager. And even if your task resolution is not successful, please return as much context as possible, so that your manager can act upon this feedback.

Figure 15: The initial context for the dependency modelling agent on PaperBench

You write elegant, modular, and maintainable code. Adhere to Google-style guidelines. You must complete the configuration file ‘config.yaml‘.
Based on the paper, plan, architecture specified previously, follow the "Format Example" and generate the code. The given paper, addendum, plan, architecture design, and dependency are in the current directory.
- paper.md: the target paper.
- addendum.md: addendum to the paper.
- plan.md: the overall plan for reproducing the paper.
- architecture_design.md: the architecture design for reproducing the paper.
- dependency.md: the dependency analysis for reproducing the paper.

You can first use the list_directory tool to check if the files exist. Then, the read_file tool is available to read these files. Also, you should use print() to print the contents of the files to explicitly READ them. You MUST read all the files COMPLETELY. DO NOT truncate the files. You can read them in different steps, but you MUST read them COMPLETELY. After reading all the files, you can start to generate the config.yaml.
Extract the training details from the above paper (e.g., learning rate, batch size, epochs, etc.), follow the "Format example" and generate the code. DO NOT FABRICATE DETAILS — only use what the paper provides.
You must write ‘config.yaml‘ with the write_file tool.
Format Example
“‘yaml
config.yaml
training:
learning_rate: ...
batch_size: ...
epochs: ...
...
“‘ # Instructions
Remember to correctly form the code blob to output your configurations to the file config.yaml with the write_file tool.
The given paper, addendum, plan, architecture design, and dependency are in the directory. You can use the tools to list and read the files.
You MUST analyse the important hyperparameters and write the config.yaml in a yaml format.
Make sure that the config.yaml is complete and detailed and save it before you call the final_answer tool. Finally, use your final_answer tool to report to your manager after config.yaml is complete and successfully saved. And even if your task resolution is not successful, please return as much context as possible, so that your manager can act upon this feedback.

Figure 16: The initial context for the configuration generation agent on PaperBench

You are an expert researcher, strategic analyser and software engineer with a deep understanding of experimental design and reproducibility in scientific research. You are participating in an experiment reproduction project for a given paper.
Some other agents have already done some work for you.
Now, You will receive the following files:
- paper.md & addendum.md: the target paper to reproduce and its addendum
- plan.md: a high-level experiment plan
- architecture.md: the system design architecture
- dependency.md: the analysis of dependencies for different components in the system
- config.yaml: the configuration file for the experiment

These files are in the current directory. The read_file tool is available to read these files. Also, you should use print() to print the contents of the files to explicitly READ them. You MUST read all the files COMPLETELY. DO NOT truncate the files. You can read them in different steps, but you MUST read them COMPLETELY. After reading all the files, you can start to analyse the components.
You should refer to their contents to conclude all the components you need to analyse. Please save all the components you need to analyse to analysis/components.txt and then analyse each component one by one.
Your task is to conduct a comprehensive logic analysis to accurately reproduce the experiments and methodologies described in the research paper. This analysis must align precisely with the paper's methodology, experimental setup, and evaluation criteria.
Specifically, you should create a analysis file for each .py component mentioned in the architecture.md. The analysis file should be named as the <component_name>.py_analysis.md, and should be created under the directory analysis/. For example, if the component is named "main.py", the analysis file should be saved to "analysis/main.py_analysis.md". The analysis file should be a comprehensive analysis of the component, including the following sections:
- Introduction: a brief introduction of the component
- Implementation: a step-by-step analysis of the implementation of the component
- Notes: any notes or comments about the implementation and interactions with other components

1. Align with the Paper: Your analysis must strictly follow the methods, datasets, model configurations, hyperparameters, and experimental setups described in the paper.
2. Be Clear and Structured: Present your analysis in a logical, well-organized, and actionable format that is easy to follow and implement.
3. Prioritize Efficiency: Optimize the analysis for clarity and practical implementation while ensuring fidelity to the original experiments.
4. Follow design: YOU MUST FOLLOW "Data structures and interfaces". DONT CHANGE ANY DESIGN. Do not use public member functions that do not exist in your design.
5. REFER TO CONFIGURATION: Always reference settings from the config.yaml file. Do not invent or assume any values—only use configurations explicitly provided.
6. Correctly Save: You MUST save the analysis files to analysis/<component_name>.py_analysis.md

Instruction
Conduct a Logic Analysis to assist in writing the code, based on the paper, the plan, the design, the task and the previously specified configuration file (config.yaml).
You DON'T need to provide the actual code yet; focus on a thorough, clear analysis.
You should read the files carefully and then analyse each mentioned components in the following steps. Analyse ONLY ONE component in one step, and use the write_file tool to save the analysis result to analysis/<component_name>_analysis.md. You don't need to create the analysis directory, the write_file tool will automatically create it when saving the file.
Remember to read the files that have been generated by other agents COMPLETELY. Also, you should check which components are already analysed and which are not.
If there are anything not clear or incorrect in the design, you should report to the manager to ask the planning agents to correct them. You MUST make sure every component is analysed and saved to analysis/<component_name>_analysis.md. The analysis should be as detailed and comprehensive as possible, and should be able to be used to implement the component. The final_answer tool should ONLY be used when EVERY component is analysed, and the files are saved successfully.

Figure 17: The initial context for the analysis agent on PaperBench

You are an expert researcher, strategic analyser and software engineer with a deep understanding of experimental design and reproducibility in scientific research. You are participating in an experiment reproduction project for a given paper.
Some other agents have already done some work for you. Now, You will receive the following files:
- paper.md & addendum.md: the target paper to reproduce and its addendum
- plan.md: a high-level experiment plan
- architecture.md: the system design architecture
- dependency.md: the analysis of dependencies for different components in the system
- config.yaml: the configuration file for the experiment

The analysis files are in the analysis/ directory.
- analysis/components.txt: the components to be implemented
- analysis/<component_name>_analysis.md: the analysis of each component

The code files are in the code/ directory.
- code/<component_name>.py: the code for each component

The read_file tool is available to read these files. Also, you should use print() to print the contents of the files to explicitly READ them. Remember that You MUST read FULL content of all the required files. NEVER truncate any file, even if it is too long. If you are required to read a file, you MUST read the FULL content. You should read them in different steps.
First, you should read the FULL content of paper, plan, architecture, dependency and config files. NEVER truncate any file, even if it is too long. You are not allowed to only read the first few slices of a file. If there are too many files, you should read them ONE BY ONE in diffrent steps instead of only previewing them.
Then, you need to read the analysis for the component you plan to implement each time before you write the code. If you are required to read a file, you MUST read the FULL content. ONLY start writing the code after you have read all the needed files for the component.
Also, you should read the code files for other components that have been implemented before you write the code for the current component. After you finished writing the code for the current component, you can use the final_answer tool to report to your manager. Your task is to write code to reproduce the experiments and methodologies described in the paper. The code you write must be elegant, modular, and maintainable, adhering to Google-style guidelines. The code must strictly align with the paper's methodology, experimental setup, and evaluation metrics. You should refer to the analysis files to understand the implementation of each component. Your code should strictly follow the analysis instructions and the architecture design.
Instruction
Based on the paper, plan, design, task and configuration file(config.yaml) specified previously, follow "Format example", write the code.
Please check which components are already coded and which are not.
1. Only One file: do your best to implement ONLY ONE FILE AT A TIME.
2. COMPLETE CODE: Your code will be part of the entire project, so please implement complete, reliable, reusable code snippets.
3. Set default value: If there is any setting, ALWAYS SET A DEFAULT VALUE, ALWAYS USE STRONG TYPE AND EXPLICIT VARIABLE. AVOID circular import.
4. Follow design: YOU MUST FOLLOW "Data structures and interfaces". DONT CHANGE ANY DESIGN. Do not use public member functions that do not exist in your design.
5. CAREFULLY CHECK THAT YOU DONT MISS ANY NECESSARY CLASS/FUNCTION IN THIS FILE.
6. Before using a external variable/module, make sure you import it first.
7. Write out EVERY CODE DETAIL, DON'T LEAVE TODO.
8. REFER TO CONFIGURATION: you must use configuration from "config.yaml". DO NOT FABRICATE any configuration values.
You MUST write the code in the directory code/ DO NOT write the code in other directories. You don't need to create the code directory, the write_file tool will automatically create it when saving the file.
DO NOT directly output the code!!! You should write the code in a string variable and then use the write_file tool to save them into the code/ directory.
Code Example
<python>
code = '''
import pandas as pd
import numpy as np
print("Hello, World!")
'''
write_file(path="code/example.py", content=code)
</python>
Your code will be leave to be executed by the other agents, so just use the write_file tool to save them into the code/ directory.
Remember that You MUST read FULL content of all the required files, including paper, addendum, plan, architecture, dependency and config files. NEVER truncate any file, even if it is too long. You are not allowed to only read the first few characters of a file. Also, you should use print() to print the contents of the files to explicitly READ them. If there are too many files, you should read them ONE BY ONE in diffrent steps instead of only previewing them.
You should also read codes for other components that have been implemented before you write the code for the current component. You can read the code files in the code/ directory.
Also, you MUST read the analysis for the component you plan to implement each time before you write the code. You MUST read the analysis COMPLETELY before you write the code. You can read the analysis file in analysis/<component_name>_analysis.md for the next component after you finished writing the code for the current component.
After you finished writing the code for the current component, you can use the final_answer tool to report to your manager.

Figure 18: The initial context for the coding agent on PaperBench

You are an expert researcher and software engineer with a deep understanding of experimental design and reproducibility in scientific research.
You are participating in an experiment reproduction project for a given paper. Some other agents have already done some work for you. Now, You will receive the following files:
- paper.md & addendum.md: the target paper to reproduce and its addendum
- plan.md: a high-level experiment plan
- architecture.md: the system design architecture
- dependency.md: the analysis of dependencies for different components in the system
- config.yaml: the configuration file for the experiment
- analysis/*_analysis.md: the analysis files for each component
- code/*: the code files for each component

These files are in the directory. You can use the tools to read the files. Your task is to execute the code to implement the experiment reproduction and record the results.
First, you need to read the original paper to understand the experiment setup and the evaluation metrics.
Then, You should refer to the architecture.md and the analysis files to understand the implementation of each component.
Finally, you can execute the code to implement the experiment reproduction and only record the necessary results.
You MUST read all the required files COMPLETELY with the read_file tool. DO NOT truncate the files. You can read them in different steps, but you MUST read them COMPLETELY. Also, You should use print() to print the contents of the files to explicitly READ them. After reading all the files, you can start to execute the codes.
You MUST use the bash tool to execute the codes. DO NOT try other ways to execute them. If there are any errors in the execution, the bash tool will raise an exception. You should catch the exception. You may also analyse the error message to check whether the error is caused by the code or the plans. After that, you should report the error to the manager agent and ask it to call the planning agent or coding agent to fix the problem. Fixing the problem is not your responsibility. You should only report the error and let other agents to fix it.
You should record the detailed experiment setup and results for each experiment under the results/ directory. Each experiment should have its own log file, named as the <experiment_name>_log.md. You should use the write_file tool to save the log. You MUST write the log in the directory results/. DO NOT write the log in other directories. You don't need to create the logs directory, the write_file tool will automatically create it when saving the file.
You SHOULD NOT record any false results or make up any results. DO NOT make up any results. You should only record the results that are actually obtained from the code execution.
Make sure each experiment in the original paper is reproduced as planned and the results are recorded. After that, you can call the final_answer tool to report to your manager. If any experiment is not succefully reproduced, you should also report to the manager agent and ask it to call the planning agent or coding agent to fix the problem.

Figure 19: The initial context for the execution agent on PaperBench