Paper2Code: Automating Code Generation from Scientific Papers in Machine Learning

Minju Seo[1], **Jinheon Baek**[1], **Seongyun Lee**[1], **Sung Ju Hwang**[1,2]
KAIST[1], DeepAuto.ai[2]
{minjuseo, jinheon.baek, seongyun, sungju.hwang}@kaist.ac.kr

Abstract

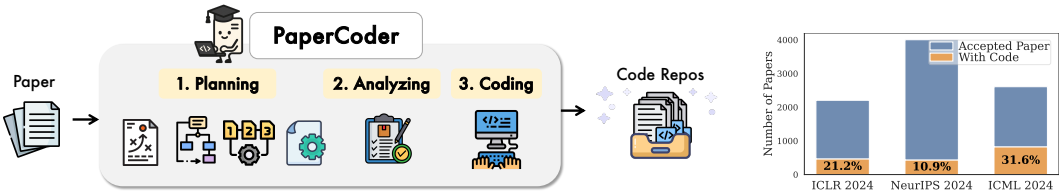
Despite the rapid growth of machine learning research, corresponding code implementations are often unavailable, making it slow and labor-intensive for researchers to reproduce results and build upon prior work. In the meantime, recent Large Language Models (LLMs) excel at understanding scientific documents and generating high-quality code. Inspired by this, we introduce PaperCoder, a multi-agent LLM framework that transforms machine learning papers into functional code repositories. PaperCoder operates in three stages: planning, where it constructs a high-level roadmap, designs the system architecture with diagrams, identifies file dependencies, and generates configuration files; analysis, which focuses on interpreting implementation-specific details; and generation, where modular, dependency-aware code is produced. Moreover, each phase is instantiated through a set of specialized agents designed to collaborate effectively across the pipeline. We then evaluate PaperCoder on generating code implementations from machine learning papers based on both model-based and human evaluations, specifically from the original paper authors, with author-released repositories as ground truth if available. Our results demonstrate the effectiveness of PaperCoder in creating high-quality, faithful implementations. Furthermore, it consistently shows strengths in the recently released PaperBench benchmark, surpassing strong baselines by substantial margins.

1 Introduction

Reproducibility lies at the heart of scientific progress, which enables researchers to validate findings, build upon prior work, and ultimately push the boundaries of knowledge [7, 3, 32]. However, across many disciplines, reproducing scientific results remains an enduring challenge. This is often due to incomplete documentation, missing experimental details, lack of access to data or proprietary tools, and, especially in machine learning research, the absence of corresponding code: for example, only 21.23 % of the papers accepted to top-tier machine learning conferences in 2024 provide their code implementations shown in Figure 1. As a result, researchers frequently invest substantial effort in reverse-engineering methods and experimental results from papers, a process that is both time-consuming and labor-intensive, subsequently slowing down the overall pace of scientific innovation.

Meanwhile, recent Large Language Models (LLMs) have shown outstanding capabilities in understanding and generating both natural language and programming code [12, 30, 37], with performances increasingly approaching or even surpassing that of domain experts in some scenarios. In addition, this progress has sparked growing interest in leveraging LLMs to accelerate scientific workflows, particularly in the early stages of ideation for new and valid research hypotheses [26, 46, 38, 2, 20, 47, 39]. Furthermore, some of these studies, as well as others focusing on later stages of automating experimental validations and improvements [16, 49, 4, 42], demonstrate the potential of LLMs to generate code and even carry out experiments end-to-end; however, they typically assume and heavily rely on access to pre-existing implementations, partial code snippets, or well-defined APIs. As such, it

Preprint. Under review.



(a) PaperCoder overview (b) Code availability

Figure 1: (a) PaperCoder overview. The proposed PaperCoder aims to transform given scientific papers (in machine learning domains) into code repositories, which consists of three sequential steps: planning, analyzing, and coding. (b) Code availability. The availability of the code repositories, where blue bars indicate the total number of accepted papers and orange regions represent the subset of papers with officially released code.

remains questionable whether generating complete and faithful implementations solely from research papers (without access to prior code, APIs, or additional supplementary materials) can be achievable.

To answer this question, we introduce PaperCoder, a multi-agent LLM-powered framework, designed to automatically generate executable code repositories in machine learning directly from and contextualized with research papers, which differs from prior work that requires partial implementations from human inputs. Specifically, PaperCoder aims to emulate the typical life cycle of human developers and researchers in writing the repository-level code, by decomposing the task into three structured stages: planning, analysis, and generation. First, during the planning stage, the proposed framework constructs a high-level roadmap to identify core components to implement, draws the overall system architecture with class and sequence diagrams to model structural relationships between modules, identifies file dependencies with their execution orders to guide correct build and execution flows, and generates configuration files to enable flexible customization of experimental workflows by human researchers. This is followed by the analysis stage, performing a fine-grained interpretation of each file and function with respect to their intended functionality, such as required inputs and outputs, interactions with other modules, and any algorithmic or architectural constraints derived from the source paper. Finally, in the generation stage, the framework synthesizes the entire code base based on the execution order determined earlier, along with the artifacts produced in the previous stages.

To validate the effectiveness of PaperCoder, we conduct extensive evaluations on a subset of recent machine learning papers accepted at top-tier venues in 2024, including NeurIPS, ICML, and ICLR referred to as our proposed Paper2Code benchmark. Furthermore, we also incorporate the recently released PaperBench benchmark in our evaluation suite, enabling the fine-grained evaluations of code implementations with some papers from ICML 2024 [40]. Then, on a battery of tests conducted with automated model-based evaluations (covering both reference-free and reference-based settings, conditional on the availability of author-released ground-truth repositories) as well as expert human evaluations (based on authors of original papers), PaperCoder demonstrates substantial improvements over baselines, generating more valid and faithful code bases that could meaningfully support human researchers in understanding and reproducing prior work. Specifically, 77% of the generated repositories by PaperCoder are rated as the best, and 85% of human judges report that the generated repositories are indeed helpful. Also, further analyses show that each component of PaperCoder (consisting of planning, analysis, and generation) contributes to the performance gains, but also that the generated code bases can be executed, sometimes with only minor modifications (averaging 0.48% of total code lines) in cases where execution errors occur.

2 Related Work

Large Language Models with Code Large language models (LLMs) demonstrate impressive capabilities in both text understanding and generation [30, 12, 37]. Recently, their applicability has extended beyond general knowledge tasks into specialized fields such as mathematics, science, and coding, showcasing impressive reasoning and knowledge representation capabilities [34, 43, 41]. Particularly, code-specialized LLMs [10, 9, 17] have garnered significant attention by achieving remarkable performance on various software engineering tasks [45], including software development, software design [36, 15], requirements elicitation [29], and specification formalization [27]. Our work aligns closely with this line of research, exploring and expanding upon the capabilities and applications of code-specialized LLMs.

Repository-Level Coding LLM-based code generation can be broadly categorized into single-file coding and multi-file (repository-level) coding. Single-file coding focuses on generating relatively

short code snippets to solve isolated tasks, such as programming competition problems or simple coding queries, and was the primary focus in the early stages of LLM-based coding research [21, 6, 1, 14]. As LLMs have advanced in code comprehension, long-context reasoning, and handling complex workflows, research has increasingly shifted toward repository-level coding, where multiple files are generated while jointly considering architectural and functional requirements [23, 18, 25]. Several recent studies have explored this direction [48, 31]. Multi-agent frameworks such as ChatDev leverage LLMs to collaborate through structured dialogues [36], while MetaGPT adopts a waterfall development model with role-specific generation stages [15]. These advancements highlight the growing capability of LLMs to support end-to-end software engineering tasks beyond single-file generation.

LLM-Powered Research Scientific progress typically involves a cycle of idea generation, hypothesis validation, and experimental execution [33]. LLMs have been applied to various stages of this cycle, including research ideation [2, 20], hypothesis generation [47, 35], and peer review [8, 22, 44], thereby helping researchers overcome existing limitations and accelerate scientific discovery [19, 26, 46]. In computer science and machine learning, where code-based experimentation is fundamental, LLMs have also been utilized to design experiments that enhance existing codebases. Many recent studies focus on refining implementations or optimizing performance, assuming the availability of the original implementation [16, 49, 4, 42]. However, this assumption limits their applicability in cases where source code is unavailable. Reproducibility plays a critical role in scientific discovery, enabling researchers to identify limitations, refine methodologies, and build upon prior work [32]. To evaluate LLMs' replication capabilities, PaperBench introduced a benchmark in which AI agents attempt to reproduce machine learning papers [40]. While this benchmark emphasizes evaluation, our work addresses a challenge of automatically generating code repositories from scientific papers in the absence of original implementations. To this end, we propose a framework that leverages LLMs for end-to-end research reproducibility. By focusing on repository-level reproduction, our approach expands the scope of LLM-powered automation beyond ideation and hypothesis generation, contributing to a critical yet underexplored aspect of scientific research.

3 Methods

3.1 Problem Definition

Research Repository Generation In machine learning research, experiments are typically conducted using a code. However, in many cases, researchers do not release their code, making it challenging for others to reproduce and verify the proposed method and experiments [28, 32]. When non-authors attempt to re-implement the code manually, the process is often labor-intensive, time-consuming. To address these challenges, we define the task of Research Repository Generation. The goal of this task is to read a paper and generate a corresponding repository that implements its methods and experiments. We define a repository as a collection of one or more code files necessary for reproducing the methods and experiments described in the paper. By automating this process, research repository generation facilitates the validation and reproducibility of scientific findings. Additionally, new research ideas often build upon prior work. However, when an researcher does not release their code, experimental validation becomes difficult, potentially hindering further advancements. Our approach aims to bridge this gap by enabling researchers to experimentally verify prior work, even when the original implementation is unavailable, thus contributing to a more reproducible and transparent research. In this study, we frame repository generation as a software development problem and assume that a model is responsible for writing the code to create a functional software repository. Given a research paper as input, the model generates a repository containing the code necessary to replicate the paper's methods and experiments. Formally, we define this as: $M(R) = C$; where R represents the paper, C represents the generated code, and M is a function or model. While M can take various forms, large language models (LLMs) are a representative candidate. A language model functions as a mapping from an input x (text) to an output y (text or code), making it well-suited for this task.

3.2 Our Method

We introduce PaperCoder, a novel framework designed specifically for research repository generation that implements the methods and experiments described in a research paper, illustrated in Figure 2.

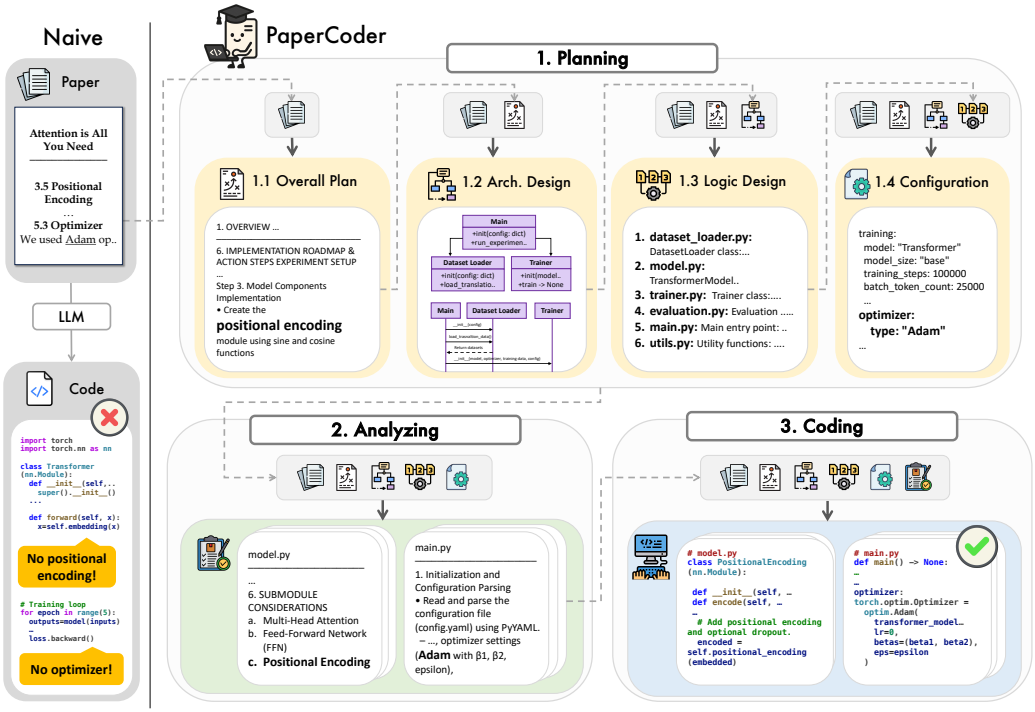


Figure 2: PaperCoder framework. (Left) The naive approach, where a model directly generates code from the paper. (Right) PaperCoder framework, which decomposes the task into three stages: (1) Planning, where a high-level implementation plan is constructed based on the paper's content, including overall plan, architectural design, logic design, and configuration files; (2) Analyzing, where the plan is translated into detailed file-level specifications; and (3) Coding, where the final codes are generated to implement the paper's methods and experiments.

The goal of our work is to model the repository generation process as: $M(R) = C$ where M is the model responsible for transforming the paper into an executable repository, R represents the research paper, and C represents the generated code. Inspired by software development methodologies, we adopt a structured approach that mirrors well-established, empirically validated software engineering principles. Specifically, we follow a plan-analysis-implementation workflow to systematically address the problem of research repository generation. To achieve this, we decompose the process into three sequential phases: 1) Planning. 2) Analysis. 3) Coding. Each phase utilizes task-specialized LLM agents that leveraging a multi-agent scenarios. Formally, we define the repository generation process as: $C = M(P) = M_{\text{code}}(R, P, A)$; where R represents the research paper, P is the plan and A indicates the analysis. Each components are generated as follows: $P = M_{\text{plan}}(R)$, $A = M_{\text{analysis}}(R, P)$ and $C = M_{\text{code}}(R, P, A)$. The following sections provide a detailed explanation of each phase.

3.2.1 Planning

Providing the research paper as a input to the model and expecting it to generate a complete repository is highly challenging. Research papers are primarily written to document discoveries and persuade readers, rather than to serve as structured input for software development. Consequently, papers often contain supplementary information that, while essential for conveying core concepts, is not directly relevant to implementation. It could be introduces noise, making repository generation difficult and less effective. To address this challenge, instead of solely using the paper as input, we propose decomposing the paper into a structured multi-aspect plan. This approach organizes the key implementation-relevant elements into four distinct components, ensuring that the generated repository is well-structured and aligned with the paper's methodology. Formally, $M_{\text{plan}}(R) = P = \{o, d, l, g\}$; where o indicates the overall plan, d is the architecture design, l represents the logic design and g is the configuration file. Each output is generated in each phase, and this phase follows a sequential structure, where the output of each stage serves as the input for the next. Note that each phase utilized by the same agent which is the plan agent M_{plan}.

Overall Plan The first step within planning phase, the overall plan, involves summarizing and organizing the core elements necessary to implement the research repository from a high-level perspective. This summary provides an essential conceptual framework that guides subsequent steps clearly. Formally, $M_{\text{plan}}(R) = o$, where R is the research paper and o indicates the overall plan.

Architecture Design The second stage involves architecting the software based on the overall plan generated in the previous stage and the research paper. Designing a well-structured architecture is essential, particularly for software systems where multiple features must interact seamlessly. This stage focuses on identifying the necessary components and defining their relationships to ensure a well-organized and functional repository. To achieve this, we asks to create key artifacts that define the software architecture. The file list is a structured collection of files required for the repository, outlining the software's modular structure. The class diagram provides a static representation of the system's data structures and interfaces. Using Unified Modeling Language (UML) notation, a standardized visual language for modeling software systems, we depict classes as rectangles, attributes and methods as lists, and relationships as connecting lines to illustrate how different components interact. The sequence diagram represents the program's call flow and object interactions dynamically, which uses UML notation to show actors as objects, messages as arrows, and lifelines as dashed lines, visually mapping out how components communicate over time. This structured approach ensures a clear and organized representation of the software's architecture. By constructing these artifacts, we visually represent the essential components described in the research paper, enabling a more structured and systematic approach to repository generation. This process facilitates better analysis of dependencies and relationships, ensuring that the generated repository aligns with the core idea of the paper. Formally, we define the architecture design process as: $M_{\text{plan}}(R, o) = d$ where R represents the paper, o refers to the previously generated overall plan, and d denotes the resulting architectural design artifacts.

Logic Design In software development, individual files rarely function in isolation. Instead, they generally exhibit interdependencies through imports and module interactions. For example, if a function A is defined in `utils.py` and later imported into `evaluation.py`, `utils.py` must be implemented before `evaluation.py` to maintain a correct dependency structure. To account for these dependencies, this stage takes as input the research paper along with the artifacts generated in the previous two stages. It then analyzes the logic of each file and its components, determining the necessary dependencies and the optimal execution order for implementation. As an output, it produces an ordered file list, detailing each file's role which files should be implemented considering dependencies and its dependencies within the repository. This approach ensures that repository generation considers not only individual file structures but also inter-file communication, facilitating a well-organized and logically coherent implementation. Formally, we define this dependency-aware file ordering process as $M_{\text{plan}}(R, o, d) = l$ where R represents the research paper, o refers to the overall plan, d denotes the architectural specifications, and l is the ordered file list which is the structured output defining the order and role of each file in the repository.

Configuration File Generation Finally, the configuration file generation step synthesizes all previously established outputs to produce a configuration file (`config.yaml`) containing hyperparameters and configuration required for model training. This process takes as input the overall plan, architectural design, and ordered file list produced in the previous stages. At this stage, users can review and modify the `config.yaml` file to identify and correct any missing or incorrectly specified details. For example, users may need to specify the path to a Hugging Face dataset or define the checkpoint storage directory. This step helps reduce hallucinations during the generation process, such as the model producing nonexistent datasets or referencing incorrect file paths. Formally, the configuration generation process is defined as $M_{\text{plan}}(R, o, d, l) = g$ where R represents the research paper, o refers to the overall plan, d denotes the software architecture specifications, l represents the logic design, and g is the resulting configuration file.

3.2.2 Analyzing

While the planning phase primarily focuses on designing the overall repository structure and outlining a high-level roadmap, the analyzing phase delves into the implementation specifics of each individual file. In this phase, the detailed purpose and necessary considerations for each file within the repository are thoroughly analyzed. The outputs generated at this stage clearly specify what each file should

achieve and highlight critical factors required for successful implementation. Specifically, the inputs to this analysis phase consist of the original research paper and previously generated artifacts (the overall plan, architecture design, logic designs, and configuration file). The outputs of this phase include file-level analyses documenting precise implementation details, which will later inform the code generation process. Formally, the file-level analysis process is defined as $\{M_{\text{analysis}}(R, P = \{o, d, l, g\}, f_i) = a_i\}_{i=1}^{n=|\text{file}|}$; where$f \in F, a \in A$ where R represents the research paper, P includes four plans, which o refers to the overall plan, d denotes the architectural specifications, l represents the logic design which consists of structured dependencies among files, g is the configuration file, and f_i specifies the target file for which the analysis is being conducted. This process is applied to each file individually to generate a a_i which is the comprehensive file-level implementation plan.

3.2.3 Coding

The final stage of our methodology is the coding phase, which produces code constituting the research repository. The generation of each file is guided by the comprehensive outputs from previous phases: the research paper itself, the overall plan, architecture design, logic designs, the configuration file, file-specific analyses, and previously generated code. As repository files often exhibit import dependencies among themselves, we adhere strictly to the ordered file list established during the planning phase, ensuring sequential consistency. Formally, the file-specific code generation process is defined as $\{M_{\text{coder}}(R, P = \{o, d, l, g\}, f_i, a_i) = c_i\}_{i=1}^{n=|\text{file}|}$; where $f \in F, a \in A, c \in C$. This process is applied sequentially for each file to ensure a structured and logically coherent repository implementation c_i. The initially produced code may require subsequent debugging or refinement to ensure correctness and full functionality. In this work, comprehensive debugging strategies and detailed error-correction workflows remain beyond the current scope of this paper.

4 Experimental Setup

4.1 Data

Paper2Code Benchmark To evaluate the effectiveness of our framework for reproducing both the proposed methods and experiments described in scientific papers, we constructed a experimental benchmark based on papers from ICML 2024, NeurIPS 2024, and ICLR 2024. Using the OpenReview API[1], we first filtered the accepted papers from each conference to include only those with publicly available GitHub repositories. Among these, we selected repositories with a total codebase of fewer than 70,000 tokens to ensure reproducibility within a manageable scope. To ensure quality, we performed model-based evaluations of the repositories and selected the top 30 highest-scoring papers from each venue. These 90 papers constitute our Paper2Code benchmark for experimental validation. A detailed list of the 90 papers is provided in Table 9, Table 10 and Table 11. For the human evaluation, we select 13 papers authored by the evaluation participants, and listed in Table 13.

PaperBench Code-Dev We further validate our framework using the PaperBench Code-Dev benchmark [40], which consists of a curated set of 20 papers from ICML 2024.

4.2 Baselines and Our Model

Paper2Code Benchmark To the best of our knowledge, there is no existing framework that directly addresses the task of end-to-end paper-to-code reproduction. However, we consider several related approaches as baselines, particularly those designed for software development from natural language inputs such as requirements. These serve as the closest comparison points in terms of producing functional code repositories. In addition, we include comparisons with a number of ablated variants that serve as simple or naive baselines, as follow: 1. **ChatDev** [36] - It is a multi-agent framework where LLM-driven agents collaboratively develop software through dialogues. In our experiments, the entire paper is provided as input(requirements) to generate a full code repository; 2. **MetaGPT** [15] - It adopts a role-based multi-agent paradigm, where software development is organized via Standardized Operating Procedures (SOPs), which specify structured workflows and responsibilities for each agent. We provide the entire paper as input and prompt the system to construct the full code

[1]https://docs.openreview.net/reference/api-v2

repository; 3. **Abstract** - A naive baseline where only the paper's abstract is provided to the language model, requiring it to implement the code repository based on minimal information; 4. **Paper** - Another naive baseline in which the full paper is provided as input, and the model is prompted to generate a corresponding code repository; 5. **PaperCoder (Ours)** - Our proposed method, which integrates planning, analysis and code generation to generate high-quality repositories from scientific papers; 6. **Oracle** - The official code repository released by the paper's authors. This serves as an upper bound for comparison, reflecting the intended implementation of the proposed in the paper.

PaperBench Code-Dev We compare our method against the two agent variants reported in the original benchmark. The **Basic Agent** follows a ReAct-style approach and is equipped with a predefined set of tools, including a bash shell command executor, a Python code runner, a web browser, and a paginated file reader for processing long documents. The **Iterative Agent** extends the Basic Agent by utilizing the full available computation time and incorporating custom prompting strategies that encourage step-by-step reasoning through subgoal decomposition (see Starace et al. [40] for details).

4.3 Evaluation Setups

4.3.1 Paper2Code Benchmark

Evaluating code generation typically relies on unit tests or verifying specific implementation components [5, 11]. However, such evaluations are often infeasible due to the lack of gold-standard repositories. Many papers do not release official code, and even when they do, test scripts are frequently unavailable. Additionally, manually annotating each paper with ground-truth implementation details is highly labor-intensive, rendering large-scale evaluation impractical.

Model-Based Evaluation To address this challenge, we adopt a model-based evaluation approach [50, 13, 24] that assesses repository quality with or without access to gold-standard repository. We consider two variants: (1) reference-based, which leverages both the paper and the author-provided repository, and (2) reference-free, which evaluates the generated repository using only the paper. For both variants, we prompt the language model to critique the required implementation components and evaluate their correctness. It assigns severity levels (high, medium and low) to any missing or flawed elements and generates an overall correctness score on a 1 to 5 scale. To ensure stability, we report the average score across multiple generations using n-way sampling. In this experiments, we set the $n = 8$ and use the `o3-mini-high` as a evaluation model.

Reference-Based Evaluation When an official repository is available, we treat it as one plausible implementation of the method described in the paper. While multiple valid implementations may exist, the author-released version is generally considered the most accurate, as it best captures the core components necessary for reproducing experimental results. In this setting, we provide both the paper and the gold-standard repository as input, and prompt the evaluation model to identify and critique the required components. The model then compares the predicted repository against these components and assigns a correctness score from 1 to 5, reflecting both component coverage and the severity of any errors.

Reference-Free Evaluation In many cases, author-released code is unavailable. To handle such scenarios, we propose a reference-free evaluation strategy that relies solely on the paper and the generated repository. As in the reference-based setting, the model is prompted to infer the required implementation components from the paper, critique them, and assess whether they are adequately implemented in the generated repository. A correctness score from 1 to 5 is then assigned based on the presence and quality of these components.

Human Evaluation While model-based evaluation provides a scalable and automated approach of assessing performance, we additionally conduct human evaluation to offer a complementary and comparative assessment of our method against various baselines. Unlike model-based evaluation, which scores each repository independently, human annotators are instructed to comparatively rank repositories generated by different methods. Given the complexity of the task, which requires both understanding the paper and judging the faithfulness of its implementation, we recruit MS and PhD students majoring in computer science with experience authoring at least one peer-reviewed paper. To

Table 1: Results on our experimental Paper2Code benchmark. We report average model-based evaluation scores for each conference. Oracle denotes the official repository released by the paper's authors. Reference-based evaluation assesses correctness (on a 1–5 scale) by comparing the generated repository against both the paper and the official implementation, while reference-free evaluation relies solely on the paper. We report statistics on the average number of tokens, files, and functions per repository. The best scores are highlighted in bold.

	Reference-based			Reference-free			Statistics		
	ICML	NeurIPS	ICLR	ICML	NeurIPS	ICLR	# of Tokens	# of Files	# of Funcs
ChatDEV	2.97 (0.58)	2.96 (0.69)	2.70(0.63)	4.12 (0.53)	4.01 (0.74)	4.00 (0.65)	6150.54	6.99	23.82
MetaGPT	2.75 (0.70)	2.95 (0.87)	2.48 (0.48)	3.63 (0.75)	3.59 (0.92)	3.52 (0.60)	5405.21	3.24	18.08
Abstract	2.43 (0.49)	2.35 (0.62)	2.28 (0.42)	3.01 (0.60)	2.99 (0.78)	3.03 (0.64)	3376.99	1.28	12.62
Paper	3.28 (0.67)	3.22 (0.80)	3.08 (0.66)	4.30 (0.53)	4.08 (0.84)	4.15 (0.63)	3846.33	1.79	14.84
PaperCoder	**3.72** (0.54)	**3.83** (0.50)	**3.68** (0.52)	**4.73** (0.44)	**4.77** (0.38)	**4.73** (0.32)	14343.38	6.97	35.22
Oracle	-	-	-	4.80 (0.32)	4.83 (0.38)	4.84 (0.26)	32149.04	28.00	122.03

ensure accurate judgment, each participant is assigned a paper they have authored as the first author. The evaluation proceeds as follows. First, annotators define key implementation criteria based on the paper, covering the `Data Processing`, `Method`, and `Evaluation` sections. They then review and rank the generated repositories within the following three comparison groups: Group 1: Model Variants of Our Method. Repositories generated by our system using different backbone models (e.g., `o3-mini` vs. three open-source alternatives); Group 2: Naive Baselines. Repositories generated using only the Paper or the Abstract as input; Group 3: Related Works. Repositories generated by existing software development frameworks, such as MetaGPT and ChatDev. Within each group, annotators assign a relative ranking. Rankings are converted into scores on a 1–5 scale. In groups with three candidates, the top-ranked repository receives 5 points, the second receives 3, and the third receives 1. After completing all group-level rankings, annotators are asked to select the overall best repository and provide a brief justification. They are also asked whether the top-ranked repository would make reproducing the paper's methods and experiments easier than starting from scratch. If not, they are required to explain why. Finally, annotators revisit the repository generated by our method and evaluate whether each of the previously defined implementation criteria is fully ($\bigcirc$), partially ($\triangle$) or not ($\times$) satisfied. Detailed evaluation templates are provided in the Figure 11.

4.3.2 PaperBench Code-Dev Evaluation

For evaluating our system on the PaperBench Code-Dev benchmark, we adopt the official evaluation protocol, which measures replication accuracy across a curated set of ICML 2024 papers. Specifically, we follow the rubric authored by the original paper's authors, which defines a hierarchical set of implementation requirements. The evaluation model assesses each generated repository based on whether the submitted code correctly implements any of the specified requirements. Importantly, the evaluation focuses only on the code development node which whether the candidate repository contains a correct implementation of some requirement.

5 Experimental Results and Analysis

Main Results Table 1 presents the main experimental results, our method consistently outperform across all conferences and both evaluation modes. Under the reference-based setting, our approach achieves the highest average correctness scores of 3.72, 3.83, and 3.68 on ICML, NeurIPS, and ICLR papers, respectively. Similarly, in the reference-free evaluation, our method attains scores of 4.73, 4.77, and 4.73, outperforming all baselines. Compared to software development work baselines such as ChatDev and MetaGPT, our method exhibits substantial performance gains. Notably, although ChatDev generates a comparable number of files (6.99 vs. 6.97), our method produces significantly more functions (35.22 vs. 23.82), indicating a higher level of granularity and completeness in the generated repositories. In contrast, MetaGPT lags behind both in evaluation scores and in code quantity metrics. Ablation variants using only the abstract or full paper underperform relative to ours. This highlights the effectiveness of our multi-stage framework in systematically extracting and organizing implementation-relevant information. For reference, we include an Oracle score that represents the upper bound, corresponding to the official implementation by the authors. While there is still a gap to the Oracle, our method achieves comparable results that are closer to this upper bound compared to existing LLM-based frameworks. Overall, these results demonstrate that our proposed system not only surpasses prior approaches in generating accurate and comprehensive code

Table 3: Human evaluation results. For model-based evaluations, including reference-based and reference-free settings, correctness scores are converted into rankings for comparability. In the human evaluation, 13 authors were asked to rank repositories within two groups based on method type. Rankings are then converted into scores by assigning 5, 3, and 1 points to the first-, second-, and third-ranked repositories, respectively.

	Score (↑)			Ranking (↓)		
	Ref-based	Ref-free	Human	Ref-based	Ref-free	Human
Abstract	2.36 (0.34)	2.99 (0.53)	1.62 (1.26)	3.00 (0.00)	3.00 (0.00)	2.69 (0.63)
Paper	3.19 (0.44)	4.22(0.52)	3.15 (1.28)	1.86 (0.36)	1.79 (0.43)	1.92 (0.64)
PaperCoder (Ours)	**3.74** (0.30)	**4.71** (0.27)	**4.23** (1.30)	**1.14** (0.36)	**1.07** (0.27)	**1.38** (0.65)
ChatDev	2.67(0.63)	3.87 (0.36)	2.69 (1.11)	2.50 (0.52)	2.36 (0.50)	2.15 (0.55)
MetaGPT	2.65 (0.46)	3.38 (0.66)	1.77 (1.30)	2.07 (0.52)	2.14 (0.52)	2.61 (0.65)
PaperCoder (Ours)	**3.74** (0.30)	**4.71** (0.27)	**4.54** (1.20)	**1.00** (0.00)	**1.00** (0.00)	**1.23** (0.60)

repositories but also excels in producing fine-grained, detail-oriented implementations necessary for faithful reproduction of research methods.

Correlation between Reference-based and Reference-free We investigate the alignment between reference-based and reference-free evaluation metrics by computing their correlation across a large set of generated repositories. As shown in Figure 3, the two evaluation scores exhibit a strong positive correlation, with a Pearson correlation coefficient of $r = 0.79$ and a statistically significant p-value of 0.00. This high correlation suggests that reference-free evaluation which does not rely on access to ground-truth code can serve as a reliable proxy for reference-based assessment. The result supports the feasibility of using reference-free methods in scenarios where official implementations are unavailable, while still maintaining consistency with traditional evaluation standards.

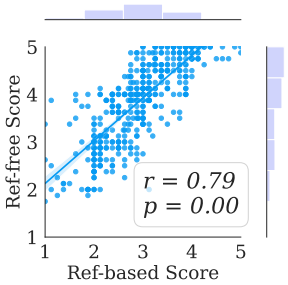


Figure 3: Correlation between reference-based and reference-free model-based evaluations.

PaperBench Code-Dev Results To further assess the effectiveness of our approach in realistic research scenarios, we evaluate it on the PaperBench Code-Dev benchmark. As shown in Table 2, our method achieves a best replication score. This demonstrates the superiority of our structured multi-agent pipeline, which performs coordinated planning, analysis, and implementation.

Table 2: Results on the PaperBench Code-Dev benchmark using `o3-mini-high`. For BasicAgent and IterativeAgent, results are averaged over three runs, and standard errors are reported.

Model	**Replication scores (%)**
BasicAgent	5.1 ± 0.8
IterativeAgent	16.4 ± 1.4
PaperCoder	**44.26**

Human Evaluation Results To assess the practical utility and subjective quality of the generated code, we conduct a comprehensive human evaluation. We include both model-based evaluations (reference-based and reference-free) and direct human assessments, where authors or expert annotators rank the quality of each generated repository. Table 3 summarizes the results across these three evaluation types. Across all evaluation criteria, our method consistently achieves the highest scores and the best rankings. In terms of average human evaluation scores, our system achieves 4.23 and 4.38 comparing both ablation variations and software development works, respectively, substantially outperforming all baselines. Similarly, in model-based reference-based and reference-free evaluations, our method attains 3.74 (ref-based) and 4.71 (ref-free), outperforming alternatives such as ChatDEV and MetaGPT. In the ranking-based evaluation, our method also secures the top position across all metrics. Notably, it achieves the best average rank in human evaluations (1.38 and 1.31), while other baselines such as MetaGPT and only Abstract consistently rank lower. The low variance in our ranking (e.g., 1.00 in both reference-based and reference-free) further demonstrates the robustness of our system across diverse evaluation settings. These results clearly indicate that our proposed approach produces more useful and accurate implementations than existing methods, regardless of whether the evaluation is automated or conducted by human experts.

Detailed Analysis on Generated Repository We conduct a fine-grained human evaluation to better understand the practical utility and implementation quality of the generated repositories. Specifically, we asked authors to review a set of repositories about covering both baselines and ablated variants

Table 4: Human evaluation results on model variants of PaperCoder. DS-Coder refers to DeepSeek-Coder-V2-Lite-Instruct, Qwen-Coder to Qwen2.5-Coder-7B-Instruct, and DS-Distil-Qwen to DeepSeek-R1-Distill-Qwen-14B. o3-mini-high denotes the high reasoning-effort of o3-mini. Other settings follow Table 3.

		DS-Coder	Qwen-Coder	DS-Distill-Qwen	o3-mini-high
Score (↑)	Ref-based	1.63 (0.43)	1.8 (0.28)	2.07 (0.30)	**3.74** (0.30)
	Ref-free	1.82 (0.39)	2.1 (0.28)	2.29 (0.29)	**4.71** (0.27)
	Human	1.41 (0.64)	2.74 (1.14)	3.05 (1.04)	**4.79** (0.74)
Ranking (↓)	Ref-based	3.36 (0.93)	2.93 (0.62)	2.36 (0.63)	**1.00** (0.00)
	Ref-free	3.36 (0.84)	2.86 (0.66)	2.14 (0.36)	**1.00** (0.00)
	Human	3.69 (0.48)	2.69 (0.85)	2.46 (0.78)	**1.15** (0.55)

and to select the one they considered most suitable for reproducing their work. 77% (10 out of 13 authors) selected the repository generated by our method as the top choice, followed by 3 authors who preferred the Paper variant. The most common reasons cited for favoring our method include completeness, clean structure, and faithfulness to the original paper. Full results are shown in Table 8.

To further assess real-world usefulness, we asked participants whether the top-ranked repository would make it easier to reproduce the paper's methods and experiments compared to starting from scratch. As a result, 85% responded positively, reinforcing the practical value of our approach. In addition, we performed a section-level implementation analysis of our generated repositories. For each paper, we asked to define criteria per section—Data Processing, Method, and Evaluation—and asked authors to evaluate whether each criterion was fully implemented (◯), partially implemented (△), or missing (×). These responses were numerically mapped to scores of 1, 0.5, and 0, respectively. The average number of criteria per section was 1.62 for Data Processing, 3.54 for Method, and 2.00 for Evaluation. As shown in Figure 4, the coverage rates were 48%, 85%, and 70% for the three sections, respectively. Follow-up analysis of the △ and × responses revealed that missing or incomplete implementations were the most common reasons for low scores in Data Processing and Evaluation, while Method sections were more often penalized for both missing logic and misunderstandings of the paper's core methodology. Overall, this section-level evaluation confirms that our method not only produces structurally sound code, but also provides substantial practical value as judged by original paper authors supporting both accurate implementation and reproducibility.

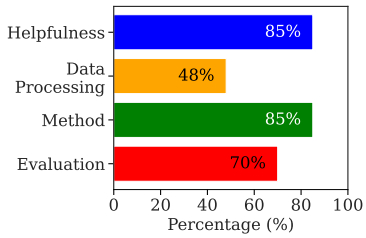


Figure 4: Human evaluation results on the helpfulness and section-level implementation accuracy of repositories generated by our framework.

Analysis on Human Alignment for Evaluation We measure their correlation with human judgments for assessing the reliability of model-based evaluation metrics. Table 5 reports the rank correlation between human scores and model-based scores across both reference-based and reference-free settings, using two evaluation models: `GPT-4o` and `o3-mini-high`. The results show moderate to strong correlations in all settings. In the reference-based evaluation, `o3-mini-high` achieves the highest correlation with human judgments (0.67), followed closely by `GPT-4o` (0.63). In the reference-free setting, both models exhibit strong alignment with human scores, each achieving a correlation of 0.63. These results suggest that model-based evaluations can serve as reliable proxies for human judgment. This supports the use of automatic scoring in large-scale experiments and reinforces the credibility of the evaluation results presented in this study. Based on these findings, we adopt the `o3-mini-high` model as the evaluation model in all experiments.

Table 5: Rank correlation between model-based and human evaluations is measured using GPT-4o (gpt-4o-2024-11-20) and o3-mini with high reasoning-effort.

Reference-based		Reference-free	
GPT-4o	o3-mini-high	GPT-4o	o3-mini-high
0.67	**0.71**	**0.67**	**0.67**

Analysis on Different LLMs To investigate the impact of the underlying language model on overall performance, we experiment with four different LLM backbones: DS-Coder (DeepSeek-Coder-V2-Lite-Instruct), Qwen-Coder (Qwen2.5-Coder-7B-Instruct), DS-Distill-Qwen (DeepSeek-R1-Distill-Qwen-14B), and `o3-mini-high` (the high reasoning-effort variant of `o3-mini`). Each model is used within the our framework, and we evaluate the outputs using reference-based, reference-free, and human evaluation metrics. Results are summarized in Table 4. We observe that `o3-mini-high`

consistently outperforms all other models across all evaluation dimensions. In terms of reference-based and reference-free scores, it achieves 3.74 and 4.71, respectively, with a higher score of 4.79 in human evaluation. Notably, it positions the top ranking in all settings, with an average rank of 1.00 in both reference-based and reference-free metrics, and 1.15 in human evaluations. Among the other models, DS-Distill-Qwen performs best, followed by Qwen-Coder and DS-Coder. This performance trend holds consistently across both model-based scores and human judgments. These results highlight the importance of selecting a capable and well-aligned backbone model for generating high-quality, executable repositories from research papers. Based on this results, we mainly use the `o3-mini-high` as the basis backbone model for all experiments.

Ablation Studies We conduct ablation experiments to examine the contribution of each module in our multi-stage pipeline. Starting with a base that uses only the full paper text (Paper), we incrementally introduce components in the order they are executed: overall plan, architecture design, core logic, configuration file, and finally the analysis module. As shown in Table 6, performance steadily improves as additional steps are incorporated. Performance temporarily dips when the architecture design module is added. This can be attributed to the lack of dependency-aware ordering at this stage. For example, generating `main.py` before defining `utils.py`, which is only resolved later during the core logic phase. Once core logic is introduced, which defines file dependencies and generation order, scores improve substantially. Subsequent additions, such as the configuration file and final analysis stage, further refine the output. Our full system (Ours) achieves the highest scores under both reference-based (3.72) and reference-free (4.73) evaluation settings. These findings highlight the importance of sequential structuring in our pipeline.

Table 6: Ablation study results on PaperCoder. We report the average scores and standard deviations on the ICML 2024 subset of Paper2Code benchmark using model-based reference-based and reference-free evaluation.

	Ref-based	Ref-free
Abstract	2.43 (0.49)	3.01 (0.60)
Paper	3.28 (0.67)	4.30 (0.53)
+ Overall Plan	3.40 (0.57)	4.34 (0.58)
+ Arch. Design	3.13 (0.68)	4.07 (0.74)
+ Logic Design	3.60 (0.52)	4.50 (0.57)
+ Config File	3.66 (0.45)	4.45 (0.53)
+ Analysis (Ours)	3.72 (0.54)	4.73 (0.44)

Analysis on Executability To verify whether the generated code is not only structurally sound but also executable with minimal intervention, we conduct a manual debugging analysis on five representative papers. For each case, we attempts to execute the generated repository and records the number of lines modified to achieve a successful run. Table 7 describes the total number of lines, the modified lines required, and the percentage of changes per repository. Remarkably, we find that on average, only 0.48% of the code needs to be altered for successful execution. Most modifications involve routine fixes such as updating deprecated OpenAI API calls to their latest versions or correcting simple type conversions. These findings indicate that our system does not merely produce syntactically correct or superficially plausible code, but outputs repositories that are functionally robust and executable with minimal human correction.

6 Conclusion and Limitations

6.1 Conclusion

In this work, we introduce PaperCoder, the framework designed to automatically generate code repositories from machine learning research papers. The framework follows a three-stage pipeline planning, analysis, and coding. The planning phase includes the construction of an overall plan, architectural design via diagrams, core logic planning with file dependencies, and configuration file generation. This plan is then refined through a detailed analysis of per-file implementation requirements, followed by sequential code generation guided by the planning outputs.

We evaluate PaperCoder on a comprehensive experimental Paper2Code benchmark consisting of 90 recent papers from top-tier machine learning conferences, as well as the PaperBench Code-Dev benchmark. Our framework consistently outperforms existing baselines across model-based evaluation metrics. In addition, we conduct human evaluations with graduate-level computer science researchers, who assess the quality of repositories generated for their own papers. Results show that 77% of participants preferred PaperCoder's implementation over alternatives, and 83% found the outputs practically useful for real-world usage. These findings collectively demonstrate the effectiveness of PaperCoder in bridging the gap between research ideas and working code.

6.2 Limitations

While our PaperCoder shows strong performance on recent machine learning papers, its current scope is limited to this specific domain. Extending the framework to support a broader range of scientific fields remains an important direction for future work. Additionally, our primary evaluation relies on model-based metrics, which may not fully capture executability or runtime correctness. Although we perform case studies involving manual debugging to validate executability, a scalable and automated approach to execution-based evaluation—including fault localization and debugging—is a promising avenue for future research. Incorporating such capabilities would further enhance PaperCoder's utility in real-world research workflows.

References

[1] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

[2] Jinheon Baek, Sujay Kumar Jauhar, Silviu Cucerzan, and Sung Ju Hwang. Researchagent: Iterative research idea generation over scientific literature with large language models. *CoRR*, abs/2404.07738, 2024. doi: 10.48550/ARXIV.2404.07738. URL https://doi.org/10.48550/arXiv.2404.07738.

[3] Monya Baker. 1,500 scientists lift the lid on reproducibility, 2016. URL https://www.nature.com/articles/533452a.

[4] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Madry. Mle-bench: Evaluating machine learning agents on machine learning engineering. *CoRR*, abs/2410.07095, 2024. doi: 10.48550/ARXIV.2410.07095. URL https://doi.org/10.48550/arXiv.2410.07095.

[5] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=ktrw68Cmu9c.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

[7] Open Science Collaboration. Estimating the reproducibility of psychological science. *Science*, 349(6251):aac4716, 2015. doi: 10.1126/science.aac4716. URL https://www.science.org/doi/abs/10.1126/science.aac4716.

[8] Mike D'Arcy, Tom Hope, Larry Birnbaum, and Doug Downey. MARG: multi-agent review generation for scientific papers. *CoRR*, abs/2401.04259, 2024. doi: 10.48550/ARXIV.2401.04259. URL https://doi.org/10.48550/arXiv.2401.04259.

[9] DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *CoRR*, abs/2406.11931, 2024. doi: 10.48550/ARXIV.2406.11931. URL https://doi.org/10.48550/arXiv.2406.11931.

[10] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, 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, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, and S. S. Li. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, abs/2501.12948, 2025. doi: 10.48550/ARXIV.2501.12948. URL https://doi.org/10.48550/arXiv.2501.12948.

[11] Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *CoRR*, abs/2301.09043, 2023. doi: 10.48550/ARXIV.2301.09043. URL https://doi.org/10.48550/arXiv.2301.09043.

[12] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. The llama 3 herd of models. *CoRR*, abs/2407.21783, 2024. doi: 10.48550/ARXIV.2407.21783. URL https://doi.org/10.48550/arXiv.2407.21783.

[13] Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. Gptscore: Evaluate as you desire. In Kevin Duh, Helena Gómez-Adorno, and Steven Bethard, editors, *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, pages 6556–6576. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.NAACL-LONG.365. URL https://doi.org/10.18653/v1/2024.naacl-long.365.

[14] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In Joaquin Vanschoren and Sai-Kit Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html.

[15] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for A multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

[16] Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation. In *Forty-first International Conference on Machine*

Learning, ICML 2024, Vienna, Austria, July 21-27, 2024. OpenReview.net, 2024. URL https://openreview.net/forum?id=1Fs1LvjYQW.

[17] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186, 2024. doi: 10.48550/ARXIV.2409.12186. URL https://doi.org/10.48550/arXiv.2409.12186.

[18] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *CoRR*, abs/2403.07974, 2024. doi: 10.48550/ARXIV.2403.07974. URL https://doi.org/10.48550/arXiv.2403.07974.

[19] Steven A. Lehr, Aylin Caliskan, Suneragiri Liyanage, and Mahzarin R. Banaji. Chatgpt as research scientist: Probing gpt's capabilities as a research librarian, research ethicist, data generator and data predictor. *CoRR*, abs/2406.14765, 2024. doi: 10.48550/ARXIV.2406.14765. URL https://doi.org/10.48550/arXiv.2406.14765.

[20] Long Li, Weiwen Xu, Jiayan Guo, Ruochen Zhao, Xingxuan Li, Yuqian Yuan, Boqiang Zhang, Yuming Jiang, Yifei Xin, Ronghao Dang, Deli Zhao, Yu Rong, Tian Feng, and Lidong Bing. Chain of ideas: Revolutionizing research via novel idea development with LLM agents. *CoRR*, abs/2410.13185, 2024. doi: 10.48550/ARXIV.2410.13185. URL https://doi.org/10.48550/arXiv.2410.13185.

[21] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814, 2022. doi: 10.48550/ARXIV.2203.07814. URL https://doi.org/10.48550/arXiv.2203.07814.

[22] Weixin Liang, Yuhui Zhang, Hancheng Cao, Binglu Wang, Daisy Ding, Xinyu Yang, Kailas Vodrahalli, Siyu He, Daniel Scott Smith, Yian Yin, Daniel A. McFarland, and James Zou. Can large language models provide useful feedback on research papers? A large-scale empirical analysis. *CoRR*, abs/2310.01783, 2023. doi: 10.48550/ARXIV.2310.01783. URL https://doi.org/10.48550/arXiv.2310.01783.

[23] Tianyang Liu, Canwen Xu, and Julian J. McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=pPjZIOuQuF.

[24] Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: NLG evaluation using gpt-4 with better human alignment. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 2511–2522. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.EMNLP-MAIN.153. URL https://doi.org/10.18653/v1/2023.emnlp-main.153.

[25] Yuliang Liu, Xiangru Tang, Zefan Cai, Junjie Lu, Yichi Zhang, Yanjun Shao, Zexuan Deng, Helan Hu, Zengxian Yang, Kaikai An, Ruijun Huang, Shuzheng Si, Sheng Chen, Haozhe Zhao, Zheng Li, Liang Chen, Yiming Zong, Yan Wang, Tianyu Liu, Zhiwei Jiang, Baobao Chang, Yujia Qin, Wangchunshu Zhou, Yilun Zhao, Arman Cohan, and Mark B. Gerstein. Ml-bench: Evaluating large language models and agents for machine learning tasks on repository-level code. 2023. URL https://api.semanticscholar.org/CorpusID:265221105.

[26] Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob N. Foerster, Jeff Clune, and David Ha. The AI scientist: Towards fully automated open-ended scientific discovery. *CoRR*, abs/2408.06292, 2024. doi: 10.48550/ARXIV.2408.06292. URL https://doi.org/10.48550/arXiv.2408.06292.

[27] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *CoRR*, abs/2402.16667, 2024. doi: 10.48550/ARXIV.2402.16667. URL https://doi.org/10.48550/arXiv.2402.16667.

[28] Ian Magnusson, Noah A. Smith, and Jesse Dodge. Reproducibility in NLP: what have we learned from the checklist? In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 12789–12811. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.FINDINGS-ACL.809. URL https://doi.org/10.18653/v1/2023.findings-acl.809.

[29] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. Clarifygpt: Empowering llm-based code generation with intention clarification. *CoRR*, abs/2310.10996, 2023. doi: 10.48550/ARXIV.2310.10996. URL https://doi.org/10.48550/arXiv.2310.10996.

[30] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023. doi: 10.48550/ARXIV.2303.08774. URL https://doi.org/10.48550/arXiv.2303.08774.

[31] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing AI software engineering with repository-level code graph. *CoRR*, abs/2410.14684, 2024. doi: 10.48550/ARXIV.2410.14684. URL https://doi.org/10.48550/arXiv.2410.14684.

[32] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Hugo Larochelle. Improving reproducibility in machine learning research(a report from the neurips 2019 reproducibility program). *J. Mach. Learn. Res.*, 22:164:1–164:20, 2021. URL https://jmlr.org/papers/v22/20-303.html.

[33] Karl Raimund Sir Popper. The logic of scientific discovery. *Systematic Biology*, 26:361, 1959. URL https://philotextes.info/spip/IMG/pdf/popper-logic-scientific-discovery.pdf.

[34] Vignesh Prabhakar, Md Amirul Islam, Adam Atanas, Yao-Ting Wang, Joah Han, Aastha Jhunjhunwala, Rucha Apte, Robert Clark, Kang Xu, Zihan Wang, and Kai Liu. Omniscience: A domain-specialized LLM for scientific reasoning and discovery. *CoRR*, abs/2503.17604, 2025. doi: 10.48550/ARXIV.2503.17604. URL https://doi.org/10.48550/arXiv.2503.17604.

[35] Biqing Qi, Kaiyan Zhang, Haoxiang Li, Kai Tian, Sihang Zeng, Zhang-Ren Chen, and Bowen Zhou. Large language models are zero shot hypothesis proposers. *CoRR*, abs/2311.05965, 2023. doi: 10.48550/ARXIV.2311.05965. URL https://doi.org/10.48550/arXiv.2311.05965.

[36] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 15174–15186. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.810. URL https://doi.org/10.18653/v1/2024.acl-long.810.

[37] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy P. Lillicrap, Jean-Baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, Ioannis Antonoglou, Rohan Anil, Sebastian Borgeaud, Andrew M. Dai, Katie Millican, Ethan Dyer, Mia Glaese, Thibault Sottiaux, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, James Molloy, Jilin Chen, Michael Isard, Paul Barham, Tom Hennigan, Ross McIlroy, Melvin Johnson, Johan Schalkwyk, Eli Collins, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Clemens Meyer, Gregory Thornton, Zhen Yang, Henryk Michalewski, Zaheer Abbas, Nathan Schucher, Ankesh Anand, Richard Ives, James Keeling, Karel Lenc, Salem

Haykal, Siamak Shakeri, Pranav Shyam, Aakanksha Chowdhery, Roman Ring, Stephen Spencer, Eren Sezener, and et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *CoRR*, abs/2403.05530, 2024. doi: 10.48550/ARXIV.2403.05530. URL https://doi.org/10.48550/arXiv.2403.05530.

[38] Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Zicheng Liu, and Emad Barsoum. Agent laboratory: Using LLM agents as research assistants. *CoRR*, abs/2501.04227, 2025. doi: 10.48550/ARXIV.2501.04227. URL https://doi.org/10.48550/arXiv.2501.04227.

[39] Chenglei Si, Diyi Yang, and Tatsunori Hashimoto. Can llms generate novel research ideas? A large-scale human study with 100+ NLP researchers. *CoRR*, abs/2409.04109, 2024. doi: 10.48550/ARXIV.2409.04109. URL https://doi.org/10.48550/arXiv.2409.04109.

[40] Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Jo hannes Heidecke, Amelia Glaese, and Tejal A. Patwardhan. Paperbench: Evaluating ai's ability to replicate ai research. 2025. URL https://arxiv.org/pdf/2504.01848.

[41] Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625:476 – 482, 2024. URL https://www.nature.com/articles/s41586-023-06747-5.

[42] Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. Automl-agent: A multi-agent LLM framework for full-pipeline automl. *CoRR*, abs/2410.02958, 2024. doi: 10.48550/ARXIV.2410.02958. URL https://doi.org/10.48550/arXiv.2410.02958.

[43] Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=z8TW0ttBPp.

[44] Yixuan Weng, Minjun Zhu, Guangsheng Bao, Hongbo Zhang, Jindong Wang, Yue Zhang, and Linyi Yang. Cycleresearcher: Improving automated research via automated review. *CoRR*, abs/2411.00816, 2024. doi: 10.48550/ARXIV.2411.00816. URL https://doi.org/10.48550/arXiv.2411.00816.

[45] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *CoRR*, abs/2407.01489, 2024. doi: 10.48550/ARXIV.2407.01489. URL https://doi.org/10.48550/arXiv.2407.01489.

[46] Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Nicolaus Foerster, Jeff Clune, and David Ha. The ai scientist-v2: Workshop-level automated scientific discovery via agentic tree search. 2025. URL https://arxiv.org/pdf/2504.08066.

[47] Zonglin Yang, Xinya Du, Junxian Li, Jie Zheng, Soujanya Poria, and Erik Cambria. Large language models for automated open-domain scientific hypotheses discovery. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 13545–13565. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.FINDINGS-ACL.804. URL https://doi.org/10.18653/v1/2024.findings-acl.804.

[48] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 2471–2484. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.EMNLP-MAIN.151. URL https://doi.org/10.18653/v1/2023.emnlp-main.151.

[49] Lei Zhang, Yuge Zhang, Kan Ren, Dongsheng Li, and Yuqing Yang. Mlcopilot: Unleashing the power of large language models in solving machine learning tasks. In Yvette Graham and Matthew Purver, editors, *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2024 - Volume 1: Long Papers, St. Julian's, Malta, March 17-22, 2024*, pages 2931–2959. Association for Computational Linguistics, 2024. URL https://aclanthology.org/2024.eacl-long.179.

[50] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html.

Table 7: **Executability results across generated repositories**. We manually select five papers and generate corresponding repositories using PaperCoder. For each repository, we report the number of lines modified during debugging, the total number of lines, and the percentage of modified lines.

Repo Name	CoLoR	cognitive-behaviors	RADA	Self-Instruct	G-EVAL	Average
Modified lines	2	1	1	25	10	6.5
Total lines	1132	2060	1609	1334	1374	1251.5
Percentage	0.18	0.05	0.06	1.87	0.73	0.48

Table 8: **Qualitative analysis of top-ranked repositories.** We categorize the reasons why human annotators selected the repositories generated by our framework as their top choice.

Completeness	Clean Structure	Faithfulness to Paper	Ease of Use	Code Quality	Unique Strengths
8	6	5	4	2	3

A Additional Experimental Designs

A.1 Implementation Details

All experiments are conducted using the `o3-mini` with high reasoning effort version (`o3-mini-high`) as the default backbone, released on January 31, 2025. To collect paper metadata and content, we utilize the `openreview_scraper`[2] and the Semantic Scholar API[3]. For document processing, we convert papers into structured JSON format using the `s2orc-doc2json` library[4].

A.2 Human Evaluation Process

To conduct evaluations with human judges, we recruited 13 researchers from the South Korea, majoring in computer science, each with a minimum of 1 published papers. For annotation, they were provided with a 4-page document, which includes the task instructions, annotation examples, and 10 generated repositories grouped into three sets. Each repository was anonymized using a `repoX` naming format to prevent bias regarding the generation method. Following the question guidelines in the document, annotators reviewed and evaluated the repositories generated by different methods and models. We note that they were compensated at a rate of $X per hour. Also, on average, evaluating 10 repositories for a single paper took approximately 45 minutes. Table 11 is a detailed annotation example.

A.3 Reference-Based Evaluation

In the reference-based evaluation setting, the repository may exceed the model's context length. Following Starace et al. [40], when this occurs, we prompt the model to select the most relevant files for evaluation. The selected subset is then used as the reference for scoring. We use the gpt-4o-2024-11-20 as the evaluation model.

[2]https://github.com/pranftw/openreview_scraper

[3]https://www.semanticscholar.org/product/api

[4]https://github.com/allenai/s2orc-doc2json

Prompt for generating the overall plan in the planning stage

[System]

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 JSON format.
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.

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.

[User]
Paper
{paper_json}

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.

Figure 5: Prompt for generating the overall plan in the planning stage.

Prompt for generating the architecture design in the planning stage

[User]

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.

—

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."
}
[/CONTENT]

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 wrapped inside [CONTENT][/CONTENT] like the format example, nothing else.

Action
Follow the instructions for the nodes, generate the output, and ensure it follows the format example.

Figure 6: Prompt for generating the architecture design in the planning stage. This prompt follows the previous prompt and response shown in Figure 5.

Prompt for generating the logic design in the planning stage

[User]
Your goal is break down tasks according to PRD/technical design, generate a task list, and analyze task dependencies.
You will break down tasks, analyze dependencies.

You outline a clear PRD/technical design for reproducing the paper's method and experiments.

Now, let's break down tasks according to PRD/technical design, generate a task list, and analyze 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.

—–

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 ........"
        ],
        [
            "trainer.py",
            "Trainer ....... "
        ],
        [
            "dataset_loader.py",
            "Handles loading and ........"
        ],
        [
            "model.py",
            "Defines the model ......."
        ],
        [
            "evaluation.py",
            "Evaluation class ........ "
        ],
        [
            "main.py",
            "Entry point ......."
        ]
    ],
    "Task list": [
        "dataset_loader.py",
        "model.py",
        "trainer.py",
        "evaluation.py",
        "main.py"
    ],
    "Full API spec": "openapi: 3.0.0 ...",
    "Shared Knowledge": "Both data_preprocessing.py and trainer.py share ........",
    "Anything UNCLEAR": "Clarification needed on recommended hardware configuration for large-scale experiments."
}
```

[/CONTENT]

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 wrapped inside [CONTENT][/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.

Figure 7: Prompt for generating the logic design in the planning stage. This prompt follows the previous prompt and response shown in Figure 6.

Prompt for generating the configuration file in the planning stage

[User]
You write elegant, modular, and maintainable code. Adhere to Google-style guidelines.

Based on the paper, plan, design specified previously, follow the "Format Example" and generate the code.
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‘.

ATTENTION: Use ’##’ to SPLIT SECTIONS, not ’#’. Your output format must follow the example below exactly.

—

Format Example
Code: config.yaml
“‘yaml
config.yaml
training:
    learning_rate: ...
    batch_size: ...
    epochs: ...
...
“‘

—

Code: config.yaml

Figure 8: Prompt for generating the configuration file in the planning stage. This prompt follows the previous prompt and response shown in Figure 7.

Prompt for analyzing

[System]
You are an expert researcher, strategic analyzer and software engineer with a deep understanding of experimental design and reproducibility in scientific research.
You will receive a research paper in JSON format, an overview of the plan, a design in JSON format consisting of "Implementation approach", "File list", "Data structures and interfaces", and "Program call flow", followed by a task in JSON format that includes "Required packages", "Required other language third-party packages", "Logic Analysis", and "Task list", along with a configuration file named "config.yaml".

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.

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.

[User]
Context
Paper
{The content of the paper in json format}

—-

Overview of the plan
{The content of the overall plan}

—-

Design
{The content of the architecture design}

—-

Task
{The content of the logic design}

—-

Configuration file
“‘yaml
{The content of the configuration file}
“‘
—-

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.

Write the logic analysis in '{The name of the file to be generated"}', which is intended for '{Description of the file generated through the "Logic Analysis" step of the logic design.}'.

—-

Logic Analysis: {todo_file_name}

Figure 9: Prompts for analyzing. {} indicate placeholders to be filled with the content described in the accompanying explanation. The prompt is presented to the LLM for each file, following the sequence defined in the logic design.

Prompt for coding

[System]
You are an expert researcher and software engineer with a deep understanding of experimental design and reproducibility in scientific research.
You will receive a research paper in JSON format, an overview of the plan, a Design in JSON format consisting of "Implementation approach", "File list", "Data structures and interfaces", and "Program call flow", followed by a Task in JSON format that includes "Required packages", "Required other language third-party packages", "Logic Analysis", and "Task list", along with a configuration file named "config.yaml".
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.
Write code with triple quoto.

[User]
Context
Paper
{The content of the paper in json format}

—-

Overview of the plan
{The content of the overall plan}

—-

Design
{The content of the architecture design}

—-

Task
{The content of the logic design}

—-

Configuration file
“‘yaml
{The content of the configuration file}
“‘
—-

Code Files
{The content of the code files generated in the previous step.}

—-

Format example
Code: {todo_file_name}
“‘python
todo_file_name
...
“‘

—-

Instruction
Based on the paper, plan, design, task and configuration file(config.yaml) specified previously, follow "Format example", write the code.

We have {done_file_lst}.
Next, you must write only the "{todo_file_name}".
1. Only One file: do your best to implement THIS ONLY ONE FILE.
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.

{detailed_logic_analysis}

Code: {todo_file_name}

Figure 10: Prompts for coding. {} indicate placeholders to be filled with the content described in the accompanying explanation. The prompt is presented to the LLM for each file, following the sequence defined in the logic design. Previously generated code files are accumulated and provided as part of the ## Code Files input.

Table 9: List of ICLR 2024 papers used in our experimental benchmark. We evaluate each paper using the model-based, reference-free setting, with `gpt-4o-2024-11-20` as the evaluation model.

Paper	Source	Score
Generative Judge for Evaluating Alignment	Poster	4
Distributional Preference Learning: Understanding and Accounting for Hidden Context in RLHF	Poster	4
Inherently Interpretable Time Series Classification via Multiple Instance Learning	Oral	3.9
iTransformer: Inverted Transformers Are Effective for Time Series Forecasting	Oral	3.9
Tell Your Model Where to Attend: Post-hoc Attention Steering for LLMs	Poster	3.9
Knowledge Distillation Based on Transformed Teacher Matching	Poster	3.9
Meaning Representations from Trajectories in Autoregressive Models	Poster	3.8
A Simple Interpretable Transformer for Fine-Grained Image Classification and Analysis	Poster	3.8
VDC: Versatile Data Cleanser based on Visual-Linguistic Inconsistency by Multimodal Large Language Models	Poster	3.8
Vocos: Closing the gap between time-domain and Fourier-based neural vocoders for high-quality audio synthesis	Poster	3.8
SliceGPT: Compress Large Language Models by Deleting Rows and Columns	Poster	3.8
Beyond Accuracy: Evaluating Self-Consistency of Code Large Language Models with IdentityChain	Poster	3.8
Guiding Masked Representation Learning to Capture Spatio-Temporal Relationship of Electrocardiogram	Poster	3.8
Social Reward: Evaluating and Enhancing Generative AI through Million-User Feedback from an Online Creative Community	Oral	3.7
Language Model Detectors Are Easily Optimized Against	Poster	3.7
Improving protein optimization with smoothed fitness landscapes	Poster	3.7
SparseFormer: Sparse Visual Recognition via Limited Latent Tokens	Poster	3.7
AutoVP: An Automated Visual Prompting Framework and Benchmark	Poster	3.7
Hierarchical Context Merging: Better Long Context Understanding for Pre-trained LLMs	Poster	3.7
SEABO: A Simple Search-Based Method for Offline Imitation Learning	Poster	3.7
OpenChat: Advancing Open-source Language Models with Mixed-Quality Data	Poster	3.7
Rethinking The Uniformity Metric in Self-Supervised Learning	Poster	3.7
VONet: Unsupervised Video Object Learning With Parallel U-Net Attention and Object-wise Sequential VAE	Poster	3.6
Efficient Backpropagation with Variance-Controlled Adaptive Sampling	Poster	3.6
Structuring Representation Geometry with Rotationally Equivariant Contrastive Learning	Poster	3.6
ControlVideo: Training-free Controllable Text-to-Video Generation	Poster	3.6
Context-Aware Meta-Learning	Poster	3.6
RECOMBINER: Robust and Enhanced Compression with Bayesian Implicit Neural Representations	Poster	3.6
Peering Through Preferences: Unraveling Feedback Acquisition for Aligning Large Language Models	Poster	3.6
Modulate Your Spectrum in Self-Supervised Learning	Poster	3.6

Table 10: List of NeurIPS 2024 papers used in our experimental benchmark. We evaluate each paper using the model-based, reference-free setting, with `gpt-4o-2024-11-20` as the evaluation model.

Paper	Source	Score
PACE: marrying generalization in PArameter-efficient fine-tuning with Consistency rEgularization	Oral	4
The Road Less Scheduled	Oral	4
G-Retriever: Retrieval-Augmented Generation for Textual Graph Understanding and Question Answering	Poster	4
Binarized Diffusion Model for Image Super-Resolution	Poster	4
Learning to Predict Structural Vibrations	Poster	4
Attack-Aware Noise Calibration for Differential Privacy	Poster	4
Make Your LLM Fully Utilize the Context	Poster	3.9
Smoothed Energy Guidance: Guiding Diffusion Models with Reduced Energy Curvature of Attention	Poster	3.9
Sm: enhanced localization in Multiple Instance Learning for medical imaging classification	Poster	3.9
AutoTimes: Autoregressive Time Series Forecasters via Large Language Models	Poster	3.9
End-to-End Ontology Learning with Large Language Models	Poster	3.8
Scaling transformer neural networks for skillful and reliable medium-range weather forecasting	Poster	3.8
Autoregressive Image Generation without Vector Quantization	Oral	3.7
Adaptive Randomized Smoothing: Certified Adversarial Robustness for Multi-Step Defences	Oral	3.7
Generalizable Person Re-identification via Balancing Alignment and Uniformity	Poster	3.7
Universal Neural Functionals	Poster	3.7
Are Self-Attentions Effective for Time Series Forecasting?	Poster	3.7
xMIL: Insightful Explanations for Multiple Instance Learning in Histopathology	Poster	3.7
Leveraging Environment Interaction for Automated PDDL Translation and Planning with Large Language Models	Poster	3.7
Task-Agnostic Machine Learning-Assisted Inference	Poster	3.7
Make Continual Learning Stronger via C-Flat	Poster	3.7
DARG: Dynamic Evaluation of Large Language Models via Adaptive Reasoning Graph	Poster	3.7
AsyncDiff: Parallelizing Diffusion Models by Asynchronous Denoising	Poster	3.7
You Only Look Around: Learning Illumination Invariant Feature for Low-light Object Detection	Poster	3.6
MutaPLM: Protein Language Modeling for Mutation Explanation and Engineering	Poster	3.6
Advancing Training Efficiency of Deep Spiking Neural Networks through Rate-based Backpropagation	Poster	3.6
Improved off-policy training of diffusion samplers	Poster	3.6
Navigating the Effect of Parametrization for Dimensionality Reduction	Poster	3.6
Long-Range Feedback Spiking Network Captures Dynamic and Static Representations of the Visual Cortex under Movie Stimuli	Poster	3.6
InfLLM: Training-Free Long-Context Extrapolation for LLMs with an Efficient Context Memory	Poster	3.6

Table 11: List of ICML 2024 papers used in our experimental benchmark. We evaluate each paper using the model-based, reference-free setting, with `gpt-4o-2024-11-20` as the evaluation model.

Paper	Source	Score
SAMformer: Unlocking the Potential of Transformers in Time Series Forecasting with Sharpness-Aware Minimization and Channel-Wise Attention	Oral	4
Autoformalizing Euclidean Geometry	Poster	4
Recurrent Distance Filtering for Graph Representation Learning	Poster	4
CosPGD: an efficient white-box adversarial attack for pixel-wise prediction tasks	Poster	3.9
Token-level Direct Preference Optimization	Poster	3.9
BayOTIDE: Bayesian Online Multivariate Time Series Imputation with Functional Decomposition	Oral	3.8
CurBench: Curriculum Learning Benchmark	Poster	3.8
Exploring the Low-Pass Filtering Behavior in Image Super-Resolution	Poster	3.8
Towards Efficient Exact Optimization of Language Model Alignment	Poster	3.7
On the Effectiveness of Supervision in Asymmetric Non-Contrastive Learning	Poster	3.7
Drug Discovery with Dynamic Goal-aware Fragments	Poster	3.7
Fool Your (Vision and) Language Model With Embarrassingly Simple Permutations	Poster	3.7
Image Restoration Through Generalized Ornstein-Uhlenbeck Bridge	Poster	3.7
Timer: Generative Pre-trained Transformers Are Large Time Series Models	Poster	3.7
Mitigating Oversmoothing Through Reverse Process of GNNs for Heterophilic Graphs	Poster	3.7
Scribble-Supervised Semantic Segmentation with Prototype-based Feature Augmentation	Poster	3.7
ConvNet vs Transformer, Supervised vs CLIP: Beyond ImageNet Accuracy	Poster	3.7
CLIF: Complementary Leaky Integrate-and-Fire Neuron for Spiking Neural Networks	Oral	3.6
FiT: Flexible Vision Transformer for Diffusion Model	Oral	3.6
Decomposing Uncertainty for Large Language Models through Input Clarification Ensembling	Oral	3.6
SparseTSF: Modeling Long-term Time Series Forecasting with *1k* Parameters	Oral	3.6
Sample-specific Masks for Visual Reprogramming-based Prompting	Oral	3.6
Boundary Exploration for Bayesian Optimization With Unknown Physical Constraints	Poster	3.6
Listwise Reward Estimation for Offline Preference-based Reinforcement Learning	Poster	3.6
Graph Distillation with Eigenbasis Matching	Poster	3.6
Temporal Spiking Neural Networks with Synaptic Delay for Graph Reasoning	Poster	3.6
Position: Quo Vadis, Unsupervised Time Series Anomaly Detection?	Poster	3.6
Neural SPH: Improved Neural Modeling of Lagrangian Fluid Dynamics	Poster	3.6
Self-Play Fine-Tuning Converts Weak Language Models to Strong Language Models	Poster	3.6
Unveiling and Harnessing Hidden Attention Sinks: Enhancing Large Language Models without Training through Attention Calibration	Poster	3.6

Table 12: List of papers used in human evaluation. We evaluate each paper using the model-based, reference-free setting, with `gpt-4o-2024-11-20` as the evaluation model.

Paper	Score
VideoICL: Confidence-based Iterative In-context Learning for Out-of-Distribution Video Understanding	2.6
Identity Decoupling for Multi-Subject Personalization of Text-to-Image Models	3.3
Knowledge-Augmented Language Model Verification	3.3
SEA: Sparse Linear Attention with Estimated Attention Mask	2.8
HarmAug: Effective Data Augmentation for Knowledge Distillation of Safety Guard Models	3.0
Graph Generation with Diffusion Mixture	3.7
Adaptive-RAG: Learning to Adapt Retrieval-Augmented Large Language Models through Question Complexity	2.9
Sketch-of-Thought: Efficient LLM Reasoning with Adaptive Cognitive-Inspired Sketching	4.0
Mol-LLaMA: Towards General Understanding of Molecules in Large Molecular Language Model	3.5
Rethinking Code Refinement: Learning to Judge Code Efficiency	3.1
Knowledge-Augmented Reasoning Distillation for Small Language Models in Knowledge-Intensive Tasks	3.2
Concept-skill Transferability-based Data Selection for Large Vision-Language Models	3.0
Aligning to thousands of preferences via system message generalization	3.5

Table 13: List of papers used in executability analysis.

Repo Name	Paper
CoLoR	Efficient Long Context Language Model Retrieval with Compression
cognitive-behaviors	Cognitive Behaviors that Enable Self-Improving Reasoners, or, Four Habits of Highly Effective STaRs
RADA	Retrieval-Augmented Data Augmentation for Low-Resource Domain Tasks
Self-Instruct	Self-Instruct: Aligning Language Models with Self-Generated Instructions
G-EVAL	G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment

Name:
Paper:
Github:

[General]

1. If someone wants to reproduce the methods and experiments in your paper, which components would they need to implement? Please break down into the following sections: **(1) data processing, (2) method (e.g., model training or main pipeline), and (3) evaluation.**

For example, in Self-Instruct (TLDR; Self-Instruct, a framework for improving the instruction-following capabilities of language models by bootstrapping off their own generations.):

Data Processing

- N/A

Method (e.g., Model training or Main pipeline)

1. Instruction Generation
2. Classification Task Identification
3. Instance Generation
4. Filtering

Evaluation

- Training the model using the generated synthetic data via our methods
- Evaluating the trained model

Your Answer
Data Processing Method (e.g., Model training or Main pipeline) Evaluation

Figure 11: Human Evaluation Format

[Comparison]

2. Given a set of repositories, which one is the most helpful for reproducibility—that is, which one best re-implements the methods and experiments as intended by the paper?

Please review the provided repositories (Group 1: repo1–repo4, Group 2: repo5–repo7, Group 3: repo8–repo10) and rank them based on how well they are implemented.

It is worth noting that the same repository may appear more than once between repo1 and repo10; this is not an error.

(Optional things: Feel free to leave a comment explaining why you ranked them that way)

[Group1: repo1-repo4]

1st	
2nd	
3rd	
4th	

[Group2: repo5-repo7]

1st	
2nd	
3rd	

[Group3: repo8-repo10]

1st	
2nd	
3rd	

Among the top-ranked repositories in each group, which one do you think is the best? If the repositories are the same, you can select any of them. Please briefly explain your reason.

[All: repo1-repo10]

1st	
Reason	

Figure 12: Human Evaluation Format

[Detailed Analysis about the 1st Repository]

3. Do you think the first-ranked repository you chose would make it easier to reproduce the paper's methods and experiments than starting from scratch?

Yes	
No	

If you selected 'No', please briefly explain why. Otherwise, you may leave this blank.

Reason for No	

4. Based on the key components you mentioned in question 1, how well does the **"repo10"** repository support them?
Please check one of the following for each component:
(o = fully implemented, △ = partially implemented, x = not implemented)
If you select △ or ×, please briefly explain your reason.

Example: Self-Instruct (TLDR; Self-Instruct, a framework for improving the instruction-following capabilities of language models by bootstrapping off their own generations.)

Data Processing

- N/A

Method (e.g., Model training or Main pipeline)

1. Instruction Generation (o)
2. Classification Task Identification (o)
3. Instance Generation (△) : *They don't implement output-first and input-first separately.*
4. Filtering (△) : *They only implemented it using the ROUGE-L-based filter, not with the exact same input-output pairs.*

Evaluation

- Training the model using the generated synthetic data via our methods (o)
- Evaluating the trained model (x): *They only provided the training code.*

Your Answer
Data Processing Method (e.g., Model training or Main pipeline) Evaluation

Figure 13: Human Evaluation Format