

AUTOREPRODUCE: Automatic AI Experiment Reproduction with Paper Lineage

Xuanle Zhao¹, Zilin Sang², Yuxuan Li^{1,†}, Qi Shi^{1,†}, Shuo Wang¹, Duzhen Zhang³
Xu Han¹, Zhiyuan Liu¹, Maosong Sun¹

¹Tsinghua University ²Xidian University ³Chinese Academy of Sciences
{2429527z, yxuanl1995, qshi9510}@gmail.com

Abstract

Efficient experiment reproduction is critical to accelerating progress in artificial intelligence. However, the inherent complexity of method design and training procedures presents substantial challenges for automation. Notably, reproducing experiments often requires implicit domain-specific knowledge not explicitly documented in the original papers. To address this, we introduce the paper lineage algorithm, which identifies and extracts implicit knowledge from the relevant references cited by the target paper. Building on this idea, we propose AUTOREPRODUCE, a multi-agent framework capable of automatically reproducing experiments described in research papers in an end-to-end manner. AUTOREPRODUCE enhances code executability by generating unit tests alongside the reproduction process. To evaluate the reproduction capability, we construct REPRODUCEBENCH, a benchmark annotated with verified implementations, and introduce novel evaluation metrics to assess both the reproduction and execution fidelity. Experimental results demonstrate that AUTOREPRODUCE outperforms the existing strong agent baselines on all five evaluation metrics by a peak margin of over 70%. In particular, compared to the official implementations, AUTOREPRODUCE achieves an average performance gap of 22.1% on 89.74% of the executable experiment runs. The code will be available at <https://github.com/AI9Stars/AutoReproduce>

1 Introduction

The rapid advancement of artificial intelligence (AI) has heightened the demand for efficient workflow iterations, making the ability to reproduce experimental results increasingly critical [Wang et al., 2024b, Xi et al., 2025, Li et al., 2025]. While this capability is pivotal for advancing various fields, the complexity of method designs and training pipelines, often requiring specialized expertise for comprehension [Si et al., 2024, Li et al., 2024], substantially hinders automated experimental replication. For instance, developing a task-specific model often requires collaboration among experts in data processing, model design, and training pipeline [Qian et al., 2023].

Notably, the rapid evolution of AI has resulted in a substantial corpus of research papers, presenting a valuable testbed for exploring the automation of experiment replication [Erdil and Besiroglu, 2023]. Prior work has typically focused utilizing large language models (LLMs) on their automating in-depth analyses of existing papers (e.g., report or idea generation [Baek et al., 2024, Lu et al., 2024]) or assisting with discrete reproduction tasks (e.g., environment setup [Bogin et al., 2024], code repository refactor [Gandhi et al., 2025, Siegel et al., 2024]). However, comprehensive frameworks for automatic end-to-end reproduction remain unexplored. While the concurrent work [Seo et al., 2025] also attempts to address the task of automatic reproduction. They only focus on reproducing

[†]Corresponding authors.

the contents introduced in the paper, without considering the executability of the generated code, which is crucial for faithful experimental reproduction [Wang et al., 2024d].

Reproducing experiments efficiently remains a significant challenge due to the lack of sufficient experimental details in research papers. We observe that distinct research domains often possess implicit domain knowledge and common implementation practices, such as specific module architectures [Chen et al., 2024] and data processing pipelines [Nie et al., 2022], which have proven effective for in-domain researchers. These details may become implicit standards in subsequent studies. Consequently, an efficient reproduction strategy should entail identifying and integrating such domain-specific knowledge and practices relevant to the target paper.

Based on preceding considerations, we propose the paper lineage algorithm, which identifies potentially unstated implementations by analyzing the citation relationships and examining associated code repositories. Building upon this algorithm, we propose AUTOREPRODUCE, a multi-agent framework designed for the end-to-end reproduction of experiments in papers. AUTOREPRODUCE contains three key stages, *literature review*, *paper lineage* and *code development*, designed to be executed sequentially to generate valid reproductions. In the code development phase, AUTOREPRODUCE utilizes batch sampling for unit testing, which efficiently generates executable code.

To validate the effectiveness of AUTOREPRODUCE, we curate REPRODUCEBENCH, a benchmark comprising 13 research papers, each representing a distinct AI sub-domain, manually constructing reference code for these papers and executing to establish baseline performance. According to the instructions, the LLM agents are expected to reproduce the specified experiment implementation detailed in the paper. During the evaluation, five distinct metrics are employed to comprehensively assess the reproduction code, covering aspects from reproduction to execution fidelity. Experimental results illustrate that AUTOREPRODUCE achieves optimal performance on REPRODUCEBENCH over all five metrics, demonstrating the effectiveness of the proposed approach. We summarize our contributions as follows:

- We introduce *paper lineage*, an algorithm that enables the agent to learn implicit domain knowledge and implementation practices by analyzing the citation relationships of the target paper.
- We introduce AUTOREPRODUCE, a novel multi-agent framework designed for end-to-end experiment reproduction, achieving superior performance in both the alignment and execution fidelity.
- We introduce REPRODUCEBENCH, a benchmark designed to evaluate the experiment reproduction capabilities of agent systems, featuring manually curated reference code implementations and multi-level evaluation metrics.

2 Related Work

2.1 LLMs for Experiment Automation

Large Language Models (LLMs) are increasingly utilized to automate diverse stages within machine learning workflows, including data engineering, model selection, hyperparameter optimization, and workflow evaluation [Gu et al., 2024]. For instance, in data engineering, LLMs are utilized to assist with many tasks such as dataset recommendation [Yang et al., 2025], adaptive data imputation [Zhang et al., 2023], context-aware data transformation [Liu et al., 2023b], and feature selection [Jeong et al., 2024]. Also, many works explore utilizing LLM-driven approaches for model selection, for example, AutoM³L [Luo et al., 2024] proposes a retrieval-based process to select the required model, while ModelGPT [Tang et al., 2024] employs generation-based methods for the same purpose. Furthermore, LLMs contribute to workflow evaluation by enabling performance prediction [Zhang et al., 2023] and supporting zero-cost proxy method development [Hollmann et al., 2022].

While current works automate discrete experimental stages, their lack of comprehensive end-to-end automation limits utility and scalability for large-scale workflows. In contrast, AUTOREPRODUCE aims to provide a full end-to-end automation of the experimental workflow.

2.2 LLMs for Research Code

Prior works have explored using LLMs for generating novel ideas [Li et al., 2024, Weng et al., 2024]. For example, Scimon [Wang et al., 2024c] focuses on scientific hypothesis discovery by elucidating

relationships between variables, while ResearchAgent [Baek et al., 2024] employs an agent-based system to generate open-ended ideas accompanied by conceptual methods and experimental designs. However, these approaches typically do not generate implementation code for these novel concepts, leading to unverifiable results. More recently, many works [Schmidgall et al., 2025, Schmidgall and Moor, 2025] utilize multi-agent frameworks to generate code implementations of proposed ideas, further promoting this field. Despite this advancement, the ideas generated by these approaches often remain high-level concepts, lacking the detailed substance described in formally proposed papers.

Reproducing experiments from papers is a highly rigorous process, requiring complete and accurate implementation of the proposed methods to achieve comparable performance. However, current approaches for automating experimental reproduction remain scarce. Recent concurrent studies [Starace et al., 2025, Seo et al., 2025] further underscore the significance of this research domain.

3 AUTOREPRODUCE

3.1 Problem Formulation

The rapid progress in the AI field yields numerous novel methods. However, manually reproducing their experiments is time-consuming and requires significant expertise, especially for innovative components and ensuring the code execution. To pipeline scientific experiment verification, we define automated experiment reproduction as the task of building LLM agents that generate executable code to reproduce methods and experiments according to the descriptions. Research papers, as verifiable method and result reports, are key resources for this automation. Formally, given a paper \mathcal{P} and instructions about experiment \mathcal{I} , the agent \mathcal{A} needs to reproduce the code implementation of the method and experiment proposed in the paper $\text{Code} = \mathcal{A}(\mathcal{P}, \mathcal{I})$, where Code is the output code.

3.2 Workflow

In this work, we propose AUTOREPRODUCE, a novel multi-agent framework for reproducing experiments proposed in research papers. The detailed pipeline of AUTOREPRODUCE is illustrated in Figure 1. AUTOREPRODUCE, designed to enhance the generation of highly reproducible and executable code, is structured into three key phases: (i) Literature Review, (ii) Paper Lineage, and (iii) Code Development.

The pipeline is executed collaboratively by a multi-agent framework comprising a research agent and a code agent. The research agent handles text-centric tasks, including paper summarization and related work analysis, while the code agent is responsible for code-oriented tasks, such as implementation and debugging.

3.2.1 Literature Review

Research papers often include redundant information for experiment reproduction. To comprehensively extract proposed methods and experimental details, the research agent employs a three-stage summarization process. First, an overall content summary provides a high-level overview. Subsequent summaries then focus on distilling specific method details (including formulas and implementation specifics) and the experimental settings necessary for reproduction. The quality of text extraction from PDF research papers critically impacts summarization, yet direct parsing methods [Schmidgall et al., 2025] often fail to capture crucial information like formulas and tables. To mitigate this, we employ Mineru [Wang et al., 2024a], a PDF-to-Markdown tool, which significantly improves formula extraction and preserves vital details. Recognizing that method structure diagrams [Zhang et al., 2024] also aid paper comprehension, AUTOREPRODUCE incorporates diagram information into summaries optionally.

3.2.2 Paper Lineage

Rome wasn’t built in a day. Much like this adage, current researchers typically advance their work upon existing studies, leveraging prior work as a foundation to propose innovative methods and conduct experiments [Bommasani et al., 2021]. As a result of this iterative process, many domain-specific consensus exist within the field [Dosovitskiy et al., 2020, Liu et al., 2023a], deriving from what we term *Paper Lineage*. To learn the fundamental knowledge of the target paper and uncover

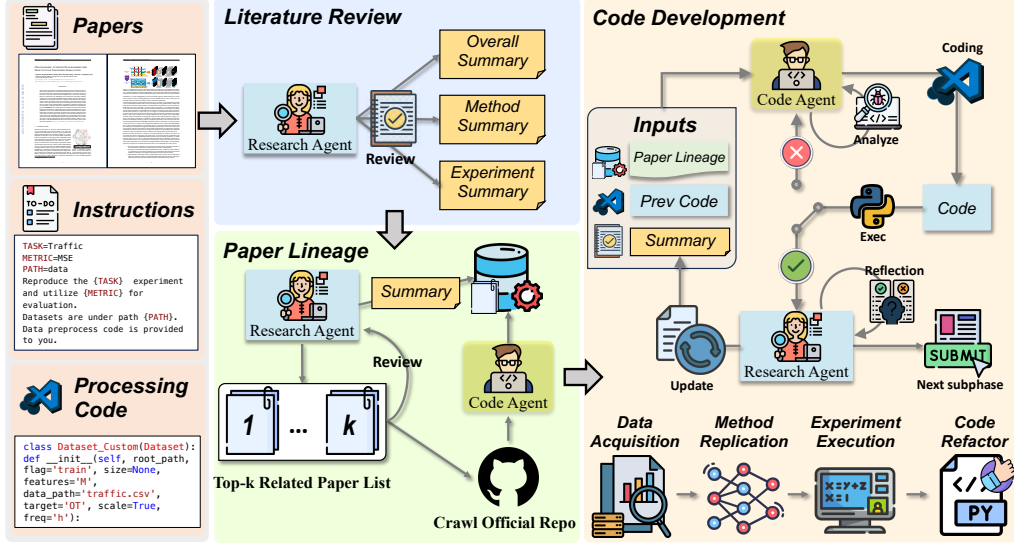


Figure 1: The paper content, instructions and data processing code (if necessary) are provided for each reproduction task. The workflow of AUTOREPRODUCE, which is decomposed into three subphases. (i) Literature Review, the research agent summarize the overall, method and experiment contents. (ii) Paper Lineage, the research agent lists and reviews related papers, and the code agent filters files in the corresponding repositories. (iii) Code Development, the code agent and research agent collaborate to construct executable reproduction code.

common practices within a research field, we propose the paper lineage algorithm. Specifically, the research agent identifies the k most relevant papers (default $k = 3$) from the reference list based on the citation relationship. The relevance depends on the degree of alignment between the selected paper and the target paper in terms of their research fields and methods. Arising from the enhanced capabilities of LLMs, we observe that state-of-the-art models possess substantial competence in identifying relevant reference works. Once the relevant paper list is obtained, the papers are downloaded via the ArXiv API. The research agent proceeds to summarize these lineage papers, extract any accessible corresponding code repository links, and subsequently employs the GitHub API to retrieve and organize their contents into structured formats. Given that many files within the repository are unrelated to the instructed experiment of the target paper, the code agent then identifies the relevant files by leveraging both the paper summary and objective instructions, subsequently extracting them by generating their corresponding file paths. These relevant code segments are subsequently integrated with their corresponding summaries to form $\langle \text{summary}, \text{code} \rangle$ pairs, which serve as domain-aligned, code implementation references for subsequent code generation. For papers lacking corresponding repositories, their summaries are directly utilized as high-quality knowledge sources. The Algorithm 1 summarizes the Paper Lineage framework.

3.2.3 Code Development

The code development phase concludes AUTOREPRODUCE workflow. In this phase, research and code agents collaborate to build reproducible and executable experiment code implementations. To enhance the reproducibility of the method and experiments described in the source paper, this phase is divided into three subphases, culminating in a final refactoring of the generated code.

Data Acquisition The initial subphase focuses on preparing the experiment data and analyze attributes. While some research leverages standardized datasets readily available through *Python* libraries such as *torchvision* or *huggingface*, a significant portion of research necessitates the construction of standardized datasets from raw data. Reproducing research code using LLM agents is often hindered by papers lacking detailed data preprocessing methods and standardized dataset formats. Our study addresses this challenge by leveraging a multi-agent framework to enhance the reproducibility of methods and experiments specifically. For cases involving custom raw data, we provide the corresponding preprocessing code, enabling LLM agents to directly load and utilize the data. The code agent enhances its comprehension of provided code and associated data attributes by analyzing critical properties (e.g., data shape and type) through the sampling of data batches and execution of

Algorithm 1 Paper Lineage Algorithm

```
1: Input: Paper  $\mathcal{P}$ ; Research Agent  $\mathcal{RA}$ ; Code Agent  $\mathcal{CA}$ ; Integer  $k$  (default 3); Instructions  $\mathcal{I}$ .
2: Output: Set of paper lineage elements  $K_{\text{lineage}}$ .
3: Initialize  $K_{\text{lineage}}$ .
4:  $\{P_1, \dots, P_k\} \leftarrow \mathcal{RA}(\mathcal{P}, k)$ . ▷ Research Agent identifies top- $k$  relevant papers
5: for each paper  $P_i$  in  $\{P_1, \dots, P_k\}$  do
6:    $\text{Text}_i \leftarrow \text{DownloadPaper}(P_i)$ . ▷ e.g., via ArXiv/Semantic Scholar API
7:    $(S_i, U_i) \leftarrow \mathcal{RA}(\text{Text}_i)$ . ▷ Research Agent extracts summary & repo URL
8:    $\mathcal{K}_i \leftarrow S_i$ . ▷ Initialize  $\mathcal{K}_i$  with summary
9:   if  $U_i$  is valid and accessible then
10:     $\text{Repo}_i \leftarrow \text{DownloadRepo}(U_i)$ . ▷ e.g., via GitHub API
11:     $\text{Code}_i \leftarrow \mathcal{CA}(S_i, \text{Repo}_i, \mathcal{I})$ . ▷ Code Agent filters for relevant code
12:     $\mathcal{K}_i \leftarrow \langle S_i, \text{Code}_i \rangle$ . ▷ Update  $\mathcal{K}_i$  to summary-code pair
13:   end if
14:   Add  $\mathcal{K}_i$  to  $K_{\text{lineage}}$ .
15: end for
16: Return  $K_{\text{lineage}}$ .
```

relevant code snippets. Our experiments validate that these properties are essential for successfully generating executable code for method components.

When code execution errors occur, the code agent needs to modify the code through the EDIT command. Specifically, the code agent identifies the start and end line numbers for replacement and generates the new code for that segment. This method substantially reduces the generated text length. The EDIT command is used in the subsequent phases.

Method Replication At this subphase, the method implementations are collaboratively reproduced by the code and the research agents. The code agent generates code snippets informed by the preceding summary, data attributes, and relevant domain knowledge derived from the Paper Lineage. The research agent evaluates the alignment of generated code with the summary, providing feedback and updating the summary as needed. Specifically, the code agent first produces an initial implementation, which undergoes subsequent refinement to ensure executability. Drawing inspiration from the software engineering field, which utilizes unit testing for debugging, the code agent samples data batches to ensure the code implementation at this stage. Furthermore, we observe that most errors occurred in the internal computations of the model functions. The code agent supports debugging data flow by printing data properties at crucial locations. The research agent reflects the code implementation details against the summary, highlighting any mismatches and updating the summary accordingly. This iterative collaboration continues until the code achieves full alignment with the methods described in the summary, at which point the research agent submits the code. After this subphase, the method proposed in the paper is reproduced and ready for subsequent experiments.

Experiments Execution The third subphase focuses on generating the implementation of the overall experimental process. Adhering to a pipeline similar to the previous subphase, the code agent and research agent collaborate to ensure the experimental code accurately reflects the implementation settings in the instructions and papers. To accelerate refinement and debugging, sampling data batches for unit testing is also integrated into code execution during this subphase, which significantly shortens the execution time. Furthermore, to avoid missing some epoch-related settings during unit testing, the code agent generates the full experiment implementations and accelerates execution using control flow statements like `break` and `continue`.

4 REPRODUCEBENCH

4.1 Contributions

Paper replication has garnered significant recent research attention, proposing several concurrent benchmarks and methods. Because AUTOREPRODUCE prioritizes method and experiment reproducibility by focusing on code execution and final performance, which differs from Paper2Code [Seo et al., 2025] that generates code in one attempt and lacks subsequent debugging or refinement. Thus,

Table 1: List of papers in REPRODUCEBENCH. To illustrate the specific experiments, we list each experimental detail, including the method names, task domains, datasets, and evaluation metrics.

Method	Domain	Dataset	Metrics
<i>IEBins</i> [Shao et al., 2023]	Monocular Depth Estimation	NYU-Depth-v2	$\delta < 1.25$
<i>iTransformer</i> [Liu et al., 2023c]	Time Series Forecasting	Traffic	MSE
<i>DKD</i> [Zhao et al., 2022]	Knowledge Distillation	CIFAR-100	Accuracy
<i>SimVP</i> [Gao et al., 2022]	Video Prediction	Moving MNIST	MSE
<i>HumanMAC</i> [Chen et al., 2023a]	Human Motion Prediction	HumanEva-I	ADE
<i>SFNet</i> [Cui et al., 2023]	Image Dehazing	SOTS-Indoor	PSNR
<i>LSM</i> [Wu et al., 2023]	Solving PDEs	Darcy	MSE
<i>Swin-Unet</i> [Cao et al., 2022]	Medical Image Segmentation	Synapse	DSC
<i>TDGNN-w</i> [Wang and Derr, 2021]	Node Classification	Citeseer	Accuracy
<i>TimeVAE</i> [Desai et al., 2021]	Time Series Generation	Sine 20%	Predictor
<i>WCDM</i> [Jiang et al., 2023]	Low-light Image Enhance	LOLv1	SSIM
<i>BSPM</i> [Choi et al., 2023]	Collaborative Filtering	Gowalla	Recall
<i>DAT-S</i> [Chen et al., 2023b]	Image Super-Resolution	DF2K/Set5	PSNR

we propose REPRODUCEBENCH, which generally focuses on papers on model architecture design methods, a scope different from PaperBench [Starace et al., 2025]. For evaluation, we utilize five distinct metrics to comprehensively evaluate the reproduction capacities of LLM agent systems.

4.2 Paper Selection

To evaluate the experimental reproduction capacity of AUTOREPRODUCE, we curate 13 papers across diverse domains. This selection covers tasks of varying complexity, for example, common applications like knowledge distillation [Zhao et al., 2022] and uncommon ones such as solving Partial Differential Equations (PDEs) [Wu et al., 2023], as well as diverse experimental conditions, including training from scratch or utilizing pre-trained models. Following this, we construct REPRODUCEBENCH by first creating reproduction instructions for the specific experiment in each paper, and subsequently manually curate reference code implementations from their official repositories, which involves pruning non-essential code snippets and refactoring the code. This process is time-consuming due to redundant functions in the repositories. Besides, all the experiments are rerun to verify reproducibility and establish baseline performance. We adopt our reproduced experimental results as the baseline for fair comparison and prevent potential misinterpretation.

4.3 Evaluation

Given the availability of reference code and performance baselines in our curated REPRODUCEBENCH, we conduct evaluations from two key perspectives to explore: (i) Does the generated code accurately reflect the core contributions and experimental setup as proposed in the source paper? (ii) Can the generated code fully reproduce the experimental performance metrics rerun by ourselves? To quantitatively answer these questions, we introduce two primary evaluation metrics, evaluating from alignment and execution aspects, respectively.

Align-Score For the alignment evaluation, we conduct comparative analyses between the generated code, the key content described in the paper and the human-curated reference code implementation. To conduct comprehensive evaluations of the generated code implementations, we evaluate the results from three aspects. (1) **Paper-Level**: We utilize the LLM (default o1) to list 5 key points vital for reproducing the paper experiment and evaluate their alignment with the generated code. (2) **Code-Level**: We utilize the LLM to compare generated code against the reference code across four criteria: Overall Structure, Model Details, Training Details and Experimental Integrity. (3) **Mixed-Level**: During our evaluation, we observe that paper-level evaluation may miss implementation details, and Code-Level evaluation focuses on specific implementations and unrelated code, which results in inflated and deflated scores, respectively. To tackle this limitation, we propose the mixed-level evaluation. This method involves providing the LLM with key points utilized in the paper-level evaluation alongside reference code, allowing it to understand the implementation of each point and subsequently score the generated code accordingly. The assessment results effectively capture the crucial aspects and the detailed implementation, showing improved consistency with human evaluation. In contrast to the evaluation method in Paper2Code [Seo et al., 2025], which

assigns a single overall score (1-5) to the generated code. Our approach evaluates key points of the implementations, enabling a fine-grained assessment of reproduced code.

Also, human evaluations are performed to evaluate the reproduction results. For each sample, the task instruction, full paper content, reference code, and generated code are provided to the human evaluator. Evaluators assessed the generated code based on three aspects: the reproducibility of the method proposed in the paper, the hyperparameter setting and the experiment pipeline, with a maximum score of 10, 5, and 5, respectively.

Exec-Score Since experimental reproduction is a code generation task, the execution results of the generated code hold equal importance. We assess the generated code by measuring its execution rate (**Exec Rate**) and the final experimental performance (**Perf Gap**). Recognizing the lack of unified performance metrics across different methods, we propose the relative performance gap for evaluation. This involves comparing the final performance of generated code with our curated reference code to quantify their relative difference.

$$\text{Performance Gap} = \frac{1}{n} \sum_{i=1}^n \frac{|P_i^{\text{ref}} - P_i^{\text{agent}}|}{\max(P_i^{\text{ref}}, P_i^{\text{agent}})} \quad (1)$$

where P_{ref} and P_{agent} are the performance obtained under the reference code and agent-generated code, respectively. Since not all generated code can be executed, the $P_{\text{agent},i}$ is set to 0 for the non-executable instances, resulting in a maximum performance gap of 1.0. Furthermore, to prevent the performance gap from exceeding 1.0, especially in cases with small reference performance values (e.g. $P_i^{\text{ref}} = 0.1$ and $P_i^{\text{agent}} = 0.3$ when utilizing MSE as the evaluation metric), we normalize the gap by the larger performance value of P_i^{ref} and P_i^{agent} . To ensure fair comparison, we utilize the evaluation code from the corresponding code repository to assess their performance results.

5 Experiments

5.1 Baselines

Prior research predominantly centres on software development [Qian et al., 2023] rather than the requirements of experiment reproduction. While both approaches leverage code to achieve specified goals, their core implementation principles differ substantially. Reproducing experiments necessitates a rigorous correspondence between the implemented code and the method detailed in the target paper, whereas software development often prioritizes overall system functionality and completeness. Meanwhile, prior works have explored agent-assisted paper generation [Schmidgall et al., 2025], which typically demonstrate their proposed innovations by generating associated code and authoring academic papers. However, a key limitation is that these innovations, often abstract, lack implementation details, thereby hindering generate concrete implementations [Baek et al., 2024].

To better compare our proposed AUTOREPRODUCE, we compare previous work on software development and paper generation, including ChatDev [Qian et al., 2023], Agent Laboratory [Schmidgall et al., 2025] and PaperCoder [Seo et al., 2025]. To our knowledge, PaperCoder is a concurrent work compared to AUTOREPRODUCE. For all the metrics in the align-score, we employ o1 as the LLM judge for evaluations. The reference performances that we utilize in calculating performance gap metrics are obtained by rerunning the code on Tesla A100 GPUs.

5.2 Main Results

We evaluate the implementation code generated by various baselines using different LLM backbones, with three runs conducted for each paper. Detailed results in Table 2 demonstrate that our proposed AUTOREPRODUCE achieves superior performance across all the metrics, notably in execution rate and performance gap. By employing a sampling batch approach for efficient debugging, AUTOREPRODUCE significantly enhances the execution rate of the generated code and reduces its performance gap with the reference code. Specifically, AUTOREPRODUCE achieves a performance gap of **22.1%** compared to the official implementation on **89.74%** of the executable experiment runs. Our analysis indicates that LLM judges often overrate consistency when solely comparing key paper points against generated code. This overestimation stems from the generality of textual descriptions, which can reward broad functional similarity rather than precise replication. Conversely, directly comparing

Table 2: The evaluation of various agents on REPRODUCEBENCH, utilizing both o1-as-judge and execution. The presented results for each metric represent the mean value derived from three independent runs conducted across all papers in the benchmark. The best and second-best performances are indicated in **Bold** and Underline, respectively.

Baselines	LLM	Align-Score			Exec-Score (%)	
		Paper-Level	Code-Level	Mixed-Level	Exec Rate	Perf Gap (\downarrow)
ChatDev	GPT-4o	57.33	32.80	43.33	2.56	99.62
Agent Laboratory	GPT-4o	63.47	35.32	48.64	23.08	82.31
PaperCoder	o3-mini	<u>90.41</u>	47.54	60.26	17.94	89.23
AUTOREPRODUCE	Claude-3.5-Sonnet	90.35	<u>52.23</u>	<u>69.37</u>	<u>82.05</u>	<u>34.46</u>
AUTOREPRODUCE	o3-mini	91.05	57.34	73.46	89.74	30.13

Table 3: The comparative analysis of human evaluation scores, including the calculated mean and standard deviation. We employ Claude-3.5-Sonnet as the LLM backbone in this study.

Baselines	LLM	Method	Parameter	Experiment	Overall
ChatDev	GPT-4o	4.08 ± 1.00	2.85 ± 0.37	1.92 ± 0.15	8.86 ± 1.12
AUTOREPRODUCE	Claude-3.5-Sonnet	7.23 ± 0.90	3.69 ± 0.37	3.83 ± 0.14	14.19 ± 0.99

generated code with reference implementations is also problematic, as reference code frequently includes settings unmentioned in the paper, thereby skewing reproducibility assessments. In contrast, our proposed mixed-level scoring aligns more closely with human evaluators and offers a more reliable metric for evaluating reproduction capabilities.

Our human evaluation aims to confirm the reasonableness of the mixed-level metric by assessing and comparing code implementations generated by ChatDev [Qian et al., 2023] and AUTOREPRODUCE. Comparing the results in Table 8 and 3, the mixed-level scores correlate more closely with human evaluation scores than paper-level or code-level scores. Scoring feedback highlights a discrepancy: while LLM agents generally succeed in reproducing the overall model architecture specified in papers, their implementations frequently diverge from reference code in finer details, especially concerning hyperparameter configurations like convolution stride and padding values. Furthermore, research papers often omit crucial training configuration details (e.g., learning rate decay strategies and schedulers), which complicates precise experimental reproduction.

5.3 Ablation Study

To explore and evaluate our proposed methods, we conduct ablation experiments to investigate: (i) The utility of structure diagrams information and the impact of content extraction tools during literature review (ii) The effectiveness of our proposed Paper Lineage method and (iii) The performance difference between including debugging and refinement versus one-time code generation in the Code Development phase. Due to time constraints, we conduct only one experiment for each paper and evaluate on the Mixed-Level and Performance Gap metrics.

The results in Table 4 show that current settings reach the best overall performance. While including the Structure Diagram understanding (w/ Structure Diagram) shows slightly better performance on the Mixed-Level metric. However, upon reviewing the scores of this particular implementation against the default setting, we observed no substantial overall difference. Furthermore, our results demonstrate that the proposed *Paper Lineage* algorithm, coupled with debugging and refinement processes, produces significant improvements across both evaluation metrics.

Table 4: The ablation study for evaluating AUTOREPRODUCE with various subphases, the results are measured using mixed-level and performance gap metrics. We employ Claude-3.5-Sonnet as the LLM backbone.

Ablations	Mixed-Level	Perf Gap % (\downarrow)
w/ Structure Diagram	70.14	35.83
w/o Extraction Tool	58.42	47.81
w/o Paper Lineage	63.15	39.59
w/o Refine	65.78	36.37
w/o Debug+Refine	68.32	88.78
AUTOREPRODUCE	69.37	34.46

5.4 Analysis

5.4.1 Paper Lineage Correlation

AUTOREPRODUCE constructs paper lineages by selecting related papers via content and citation analysis. To assess the relevance of lineage papers selected by AUTOREPRODUCE for each target paper, we establish an expert-curated set of references as the gold standard for comparison. Specifically, for each target paper, we manually select five gold standard references, selected based on their high research relevance and temporal proximity to the target paper.

Furthermore, given that AUTOREPRODUCE default selects three papers for each source paper, we evaluate under two conditions. (i) Mean Top-1 Recall@3. Whether the most relevant expert-curated reference is included among the three lineage papers selected by AUTOREPRODUCE. (ii) Mean Hits@3. The number of lineage papers selected by AUTOREPRODUCE are present within the expert-curated set. Our analysis of the results, presented in Table 5, reveals that highly correlated papers are extracted. The metrics indicate a high degree of alignment between the papers selected by AUTOREPRODUCE and the expert-curated reference sets. Thus, a comprehensive grasp of the related work for the target paper is achieved to enhance reproducibility.

Table 5: The correlation analysis of papers selected. We utilize Claude-3.5-Sonnet and calculate the mean values of the given metrics.

Ablations	Top-1 Recall@3	Hits@3
AUTOREPRODUCE	0.77	2.23

5.4.2 Debugging and Refinement

Given that AUTOREPRODUCE utilizes EDIT command for debugging and refinement, we conduct a further analysis on the details of the code modifications. To characterize the modification statistics, we define two key metrics: average editing turns and average lines per turn, which are calculated by $N_{\text{turns}}/N_{\text{runs}}$ and $N_{\text{lines}}/N_{\text{turns}}$ respectively. Here, N_{lines} is the total number of edited lines, N_{turns} is the total number of edit turns, and N_{runs} is the total number of successful runs. This analysis considers only cases where the generated code is executable and within the method replication, and experiment execution subphases.

Table 6: The average number of turns and lines when utilizing the EDIT command.

Phase	Debugging		Refinement	
	Turns	Lines	Turns	Lines
<i>o3-mini</i>				
Method	2.14	21.74	1.23	15.65
Experiment	0.94	30.42	0.72	24.73
<i>Claude-3.5-Sonnet</i>				
Method	5.78	33.52	2.16	40.53
Experiment	2.84	34.18	1.59	19.29

The results indicate that utilizing o3-mini as the LLM backbone is significantly more efficient than Claude-3.5-Sonnet, which is evidenced by the fewer debugging and refinement iterations required to converge on high replicability and executable experiment implementations.

5.4.3 Execution Error

Our error analysis of code generated by both PaperCoder [Seo et al., 2025] and AUTOREPRODUCE reveals that the majority of issues stemmed from incorrect data shapes during model internal calculations. Intuitively, while LLMs can generate specified network structures, achieving correct data flow without test data is challenging, particularly for complex architectures. For tasks involving pre-trained models, it is often not possible to generate a completely correct model architecture implementation in a single attempts. This is mainly because LLMs still face minor challenges in accurately reproducing architectural implementations of common pre-trained models, for example, hidden layer dimensions and normalization layer configurations. Although debugging resolves numerous errors, some issues persist, particularly when encountering complex data flows.

6 Conclusion

In this study, we innovatively explore the automatic experiment reproduction problem and propose AUTOREPRODUCE, a novel multi-agent workflow with paper lineage algorithm and unit testing to generate highly reproducible and executable experiment code implementations. Furthermore, to comprehensively evaluate the performance of relevant LLM agent systems, we introduce REPRO-

DUCEBENCH, an experiment reproduction benchmark that contains the 13 papers and human-curated implementation code, which employs five distinct criteria to assess the performance of generated code, spanning evaluation from alignment to execution fidelity. The results show that this approach enables a more robust evaluation of the reproducibility capabilities of LLM agent systems. Finally, our results show AUTOREPRODUCE performs the best compared to other approaches, especially in terms of mixed-level alignment score and final performance gap.

Acknowledgement

The work is initiated and supported by AI9Stars Team. We are grateful for the support of the OpenBMB and InfiniteTensor teams.

References

- J. Baek, S. K. Jauhar, S. Cucerzan, and S. J. Hwang. Researchagent: Iterative research idea generation over scientific literature with large language models. *arXiv preprint arXiv:2404.07738*, 2024.
- B. Bogin, K. Yang, S. Gupta, K. Richardson, E. Bransom, P. Clark, A. Sabharwal, and T. Khot. Super: Evaluating agents on setting up and executing tasks from research repositories. *arXiv preprint arXiv:2409.07440*, 2024.
- R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.
- H. Cao, Y. Wang, J. Chen, D. Jiang, X. Zhang, Q. Tian, and M. Wang. Swin-unet: Unet-like pure transformer for medical image segmentation. In *European conference on computer vision*, pages 205–218. Springer, 2022.
- J. Chen, J. Mei, X. Li, Y. Lu, Q. Yu, Q. Wei, X. Luo, Y. Xie, E. Adeli, Y. Wang, et al. Transunet: Rethinking the u-net architecture design for medical image segmentation through the lens of transformers. *Medical Image Analysis*, 97:103280, 2024.
- L.-H. Chen, J. Zhang, Y. Li, Y. Pang, X. Xia, and T. Liu. Humanmac: Masked motion completion for human motion prediction. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 9544–9555, 2023a.
- Z. Chen, Y. Zhang, J. Gu, L. Kong, X. Yang, and F. Yu. Dual aggregation transformer for image super-resolution. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 12312–12321, 2023b.
- J. Choi, S. Hong, N. Park, and S.-B. Cho. Blurring-sharpening process models for collaborative filtering. In *Proceedings of the 46th international ACM SIGIR conference on research and development in information retrieval*, pages 1096–1106, 2023.
- Y. Cui, Y. Tao, Z. Bing, W. Ren, X. Gao, X. Cao, K. Huang, and A. Knoll. Selective frequency network for image restoration. In *The eleventh international conference on learning representations*, 2023.
- A. Desai, C. Freeman, Z. Wang, and I. Beaver. Timevae: A variational auto-encoder for multivariate time series generation. *arXiv preprint arXiv:2111.08095*, 2021.
- A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.
- E. Erdil and T. Besiroglu. Explosive growth from ai automation: A review of the arguments. *arXiv preprint arXiv:2309.11690*, 2023.
- S. Gandhi, D. Shah, M. Patwardhan, L. Vig, and G. Shroff. Researchcodeagent: An llm multi-agent system for automated codification of research methodologies. *arXiv preprint arXiv:2504.20117*, 2025.

- Z. Gao, C. Tan, L. Wu, and S. Z. Li. Simvp: Simpler yet better video prediction. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 3170–3180, 2022.
- Y. Gu, H. You, J. Cao, M. Yu, H. Fan, and S. Qian. Large language models for constructing and optimizing machine learning workflows: A survey. *arXiv preprint arXiv:2411.10478*, 2024.
- N. Hollmann, S. Müller, K. Eggenberger, and F. Hutter. Tabpfn: A transformer that solves small tabular classification problems in a second. *arXiv preprint arXiv:2207.01848*, 2022.
- D. P. Jeong, Z. C. Lipton, and P. Ravikumar. Llm-select: Feature selection with large language models. *arXiv preprint arXiv:2407.02694*, 2024.
- H. Jiang, A. Luo, H. Fan, S. Han, and S. Liu. Low-light image enhancement with wavelet-based diffusion models. *ACM Transactions on Graphics (TOG)*, 42(6):1–14, 2023.
- L. Li, W. Xu, J. Guo, R. Zhao, X. Li, Y. Yuan, B. Zhang, Y. Jiang, Y. Xin, R. Dang, et al. Chain of ideas: Revolutionizing research via novel idea development with llm agents. *arXiv preprint arXiv:2410.13185*, 2024.
- Z.-Z. Li, D. Zhang, M.-L. Zhang, J. Zhang, Z. Liu, Y. Yao, H. Xu, J. Zheng, P.-J. Wang, X. Chen, et al. From system 1 to system 2: A survey of reasoning large language models. *arXiv preprint arXiv:2502.17419*, 2025.
- H. Liu, C. Li, Q. Wu, and Y. J. Lee. Visual instruction tuning. *Advances in neural information processing systems*, 36:34892–34916, 2023a.
- S.-C. Liu, S. Wang, W. Lin, C.-W. Hsiung, Y.-C. Hsieh, Y.-P. Cheng, S.-H. Luo, T. Chang, and J. Zhang. Jarvix: A llm no code platform for tabular data analysis and optimization. *arXiv preprint arXiv:2312.02213*, 2023b.
- Y. Liu, T. Hu, H. Zhang, H. Wu, S. Wang, L. Ma, and M. Long. itransformer: Inverted transformers are effective for time series forecasting. *arXiv preprint arXiv:2310.06625*, 2023c.
- C. Lu, C. Lu, R. T. Lange, J. Foerster, J. Clune, and D. Ha. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.
- D. Luo, C. Feng, Y. Nong, and Y. Shen. Autom3l: An automated multimodal machine learning framework with large language models. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pages 8586–8594, 2024.
- Y. Nie, N. H. Nguyen, P. Sinthong, and J. Kalagnanam. A time series is worth 64 words: Long-term forecasting with transformers. *arXiv preprint arXiv:2211.14730*, 2022.
- C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, et al. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.
- S. Schmidgall and M. Moor. Agentrxiv: Towards collaborative autonomous research. *arXiv preprint arXiv:2503.18102*, 2025.
- S. Schmidgall, Y. Su, Z. Wang, X. Sun, J. Wu, X. Yu, J. Liu, Z. Liu, and E. Barsoum. Agent laboratory: Using llm agents as research assistants. *arXiv preprint arXiv:2501.04227*, 2025.
- M. Seo, J. Baek, S. Lee, and S. J. Hwang. Paper2code: Automating code generation from scientific papers in machine learning. *arXiv preprint arXiv:2504.17192*, 2025.
- S. Shao, Z. Pei, X. Wu, Z. Liu, W. Chen, and Z. Li. Iebins: Iterative elastic bins for monocular depth estimation. *Advances in Neural Information Processing Systems*, 36:53025–53037, 2023.
- C. Si, D. Yang, and T. Hashimoto. Can llms generate novel research ideas? a large-scale human study with 100+ nlp researchers. *arXiv preprint arXiv:2409.04109*, 2024.
- Z. S. Siegel, S. Kapoor, N. Nagdir, B. Stroebel, and A. Narayanan. Core-bench: Fostering the credibility of published research through a computational reproducibility agent benchmark. *arXiv preprint arXiv:2409.11363*, 2024.

- G. Starace, O. Jaffe, D. Sherburn, J. Aung, J. S. Chan, L. Maksin, R. Dias, E. Mays, B. Kinsella, W. Thompson, et al. Paperbench: Evaluating ai’s ability to replicate ai research. *arXiv preprint arXiv:2504.01848*, 2025.
- Z. Tang, Z. Lv, S. Zhang, F. Wu, and K. Kuang. Modelgpt: Unleashing llm’s capabilities for tailored model generation. *arXiv preprint arXiv:2402.12408*, 2024.
- B. Wang, C. Xu, X. Zhao, L. Ouyang, F. Wu, Z. Zhao, R. Xu, K. Liu, Y. Qu, F. Shang, et al. Mineru: An open-source solution for precise document content extraction. *arXiv preprint arXiv:2409.18839*, 2024a.
- L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6): 186345, 2024b.
- Q. Wang, D. Downey, H. Ji, and T. Hope. Scimon: Scientific inspiration machines optimized for novelty. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 279–299, 2024c.
- X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024d.
- Y. Wang and T. Derr. Tree decomposed graph neural network. In *Proceedings of the 30th ACM international conference on information & knowledge management*, pages 2040–2049, 2021.
- Y. Weng, M. Zhu, G. Bao, H. Zhang, J. Wang, Y. Zhang, and L. Yang. Cycleresearcher: Improving automated research via automated review. *arXiv preprint arXiv:2411.00816*, 2024.
- H. Wu, T. Hu, H. Luo, J. Wang, and M. Long. Solving high-dimensional pdes with latent spectral models. *arXiv preprint arXiv:2301.12664*, 2023.
- Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.
- Z. Yang, W. Zeng, S. Jin, C. Qian, P. Luo, and W. Liu. Autommlab: Automatically generating deployable models from language instructions for computer vision tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 22056–22064, 2025.
- D. Zhang, Y. Yu, J. Dong, C. Li, D. Su, C. Chu, and D. Yu. Mm-llms: Recent advances in multimodal large language models. *arXiv preprint arXiv:2401.13601*, 2024.
- S. Zhang, C. Gong, L. Wu, X. Liu, and M. Zhou. Automl-gpt: Automatic machine learning with gpt. *arXiv preprint arXiv:2305.02499*, 2023.
- B. Zhao, Q. Cui, R. Song, Y. Qiu, and J. Liang. Decoupled knowledge distillation. In *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*, pages 11953–11962, 2022.

A REPRODUCEBENCH Details

While our main text provides an overview of the research domain, datasets, and evaluation metrics integral to the REPRODUCEBENCH, this section offers a more detailed exposition of these elements.

A.1 Reference Code

We count the total lines of code for this reference implementation and its associated preprocessing code. Also, statistics about whether pretrained models are loaded and the reference performance rerun by ourselves are conducted. The details are shown in Table 7. Due to potential differences between our rerun settings and the experimental setup originally used by the paper authors, the reference performance may differ from the performance metric reported in their respective papers.

Table 7: More details about REPRODUCEBENCH. The Reference Performance is the rerun performance obtained by official implementations. We utilize them to calculate the performance gap.

Method	Total Lines	Preprocess Code	Preprocess Lines	Pretrained Model	Evaluation Metric	Reference Performance
<i>IEBins</i>	1877	✓	320	✗	$\delta < 1.25$	0.8854
<i>iTransformer</i>	610	✓	199	✗	MSE	0.4008
<i>DKD</i>	678	✗	0	✓	Accuary	74.56
<i>SimVP</i>	544	✓	161	✗	MSE	26.19
<i>HumanMAC</i>	1196	✓	412	✗	ADE	0.2195
<i>SFNet</i>	805	✓	121	✗	PSNR	39.68
<i>LSM</i>	403	✓	141	✗	MSE	0.0074
<i>Swin-UNET</i>	1218	✓	102	✓	DSC	0.7309
<i>TDGNN-w</i>	310	✓	21	✗	Accuary	0.7651
<i>TimeVAE</i>	711	✓	69	✗	Predictor	0.2083
<i>WCDM</i>	935	✓	117	✗	SSIM	0.7938
<i>BSPM</i>	713	✓	221	✗	Recall	0.1921
<i>DAT-S</i>	2047	✓	163	✗	PSNR	38.48

A.2 Automatic Evaluation

For the evaluation metric in the align-score, we utilize LLMs to judge the generated code from three aspects, including paper-level, code-level and mixed-level. The prompts provided to the judge LLM are generally long, requiring a powerful model for robust evaluations. Consequently, we employ o1 for this purpose. For each level, each reproduction requires approximately \$0.5 for evaluation. Thus, an evaluation run to determine the align-score costs approximately \$20 on REPRODUCEBENCH. When evaluating the generated code in the repository Qian et al. [2023], all the Python files are concatenated to form the generated code. Figure 2 denotes the prompt to extract 5 key points from the paper by utilizing o1. Figure 3 shows the prompt for high-level evaluation and Figure 4 5 show the prompt for low-level evaluation. Figure 6 7 show the prompt for mixed-level evaluation. The placeholder in each prompt is replaced with the corresponding content.

A.3 Human Evaluation

For the human evaluations, the evaluators are given the original papers, instructions, official implementations, and generated ones. To reduce the workload of human assessment, we simultaneously generated summaries of the papers using LLMs, which helps to accelerate the evaluators’ understanding of the papers. The human evaluators should score the generated code through the completeness of the method, parameters and experiment pipeline with a maximum of 10, 5, and 5. The score criteria are similar to the mixed-level evaluations in Figure 6 and 7. The minimum and maximum scores only occur in extreme circumstances where the code is totally wrong or exactly correct. For each implementation, the evaluators need around 15-25 minutes for the evaluation.

B AUTOREPRODUCE Details

For all the experiments conducted in the experiment section, AUTOREPRODUCE utilises o3-mini and claude-3-5-sonnet-20240620 as the o3-mini and claude-3.5-sonnet respectively. In the code development phase of AUTOREPRODUCE, the code agent debugs the execute errors. We set the maximum debug tries to 20 for all the subphases in the code development phase. However, further debug attempts generally do not produce more executable code, as bugs are typically fixed within 5-8 iterations. The results are also denoted in Table 6. On the REPRODUCEBENCH benchmark, the average cost for AUTOREPRODUCE to reproduce a single experiment is \$1.87 when utilising the o3-mini as the LLM backbone.

C Further Experiments

C.1 Statement

Concurrently, Paperbench Starace et al. [2025] introduces a benchmark for repo-level paper replication, emphasising including all paper-mentioned details in the generated repositories. However, REPRODUCEBENCH concentrates on reproducing the experiment results, which facilitates a more direct and robust evaluation of the execution results of the generated code. A key distinction also lies in our fundamental objectives. REPRODUCEBENCH is primarily concerned with the capability to automatically reproduce model architectures and training methodologies, with research papers serving as the medium through which we test this generative ability, rather than being the end focus themselves. Consequently, our paper selection prioritises innovations specifically in model architectures and training methodologies. Conversely, Paperbench Seo et al. [2025] focuses more on the comprehensive replication of content and details within the target papers, drawing its paper selection from high-scoring ICML publications. This difference in focus means Paperbench is not centred on our interest in automatic model construction.

C.2 PaperBench Code-Dev Results

Although differing from our original intention, we conduct additional experiments on the PaperBench Code-Dev benchmark. This benchmark has distinct requirements from REPRODUCEBENCH. Given the requirement of PaperBench for full details reproduction and the Code-Dev set does not consider code execution guarantees. We adapt the setting of AUTOREPRODUCE. In these scenarios, AUTOREPRODUCE is configured to address all experiments of a target paper, with its debugging capabilities deactivated, thereby operating in a refinement-only mode for generating the implementation tasks. All reproduced code is implemented in a single Python file. This structural choice does not affect the final evaluation results because the evaluation procedure involves concatenating all source files, rendering the input for assessment functionally equivalent to a single, comprehensive file. Without debugging, the average number of refinement turns is approximately 2.3 for the method replication phase and 1.1 for the experiment execution phase.

Table 8: The evaluation results of AUTOREPRODUCE on PaperBench Code-dev benchmarks. The BasicAgent and IterativeAgent are the implementations in the PaperBench.

Baselines	LLM	Replication Score (%)
BasicAgent	o3-mini	6.4
IterativeAgent	o3-mini	17.3
IterativeAgent	o1-high	43.4
PaperCoder	o3-mini	45.1
AUTOREPRODUCE (w/o Paper Lineage)	o3-mini	44.1
AUTOREPRODUCE	o3-mini	48.5
AUTOREPRODUCE (w/ Diagram Understanding)	o3-mini	49.6

The results show that although AUTOREPRODUCE is not primarily focused on repo-level paper reproduction, its performance is still better than that of other systems when utilizing the same LLM. Furthermore, we observe performance improvements on the PaperBench Code-Dev benchmarks, attributable to advancements in our paper lineage algorithm and diagram understanding capabilities.

Our analysis reveals that the paper lineage algorithm enables AUTOREPRODUCE to establish comparable baseline implementations, while diagram understanding provides richer contextual information regarding methods and experimental setups. As the o3-mini does not support visual input, we utilize Claude-3.5-Sonnet to generate the image comprehension text.

D Limitations

In this work, we investigate the automatic reproducibility of AI experiments, a critical aspect we believe will significantly advance automation within the AI field. However, our current approach has certain limitations that present avenues for future research. Primarily, our method is specialised for replicating individual experimental tasks rather than performing broader code generation at the repository level. Consequently, enhancing execution capabilities within the context of repo-level code warrants further exploration. Furthermore, the inherent complexity of raw datasets often necessitates preliminary data processing. Automating this data preprocessing stage itself represents a substantial research challenge that needs to be addressed to achieve more comprehensive automation.

E Ethics Statement

The primary objective of this work is to automate the replication of experiments detailed in existing, publicly available research papers. While the methodologies themselves are drawn from the public domain, which generally implies transparency, it is important to acknowledge the potential for data leakage associated with the use of our system. Users should therefore be mindful of this possibility, particularly when the replication process might involve sensitive datasets or generate intermediate results that could inadvertently disclose information.

Prompt for summarizing 5 key points proposed in the paper

TASK = MovingMnist
METRIC = MSE
TITLE = SimVP: Simpler yet Better Video Prediction
PATH = bench/simvp/source
INSTRUCTION = You are assigned an arXiv paper **{TITLE}** to replicate. You need to replicate the experiment conducted for **{TASK}** dataset in the paper. Training and testing datasets are under the folder **{PATH}**. Code related to processing the data is provided to you to obtain the dataset. Utilise **{METRIC}** as the evaluation metric.

The provided text above is the full text of the paper. The instructions for reproducing the experiment are **{INSTRUCTION}** with the model **{MODEL}**.
 Now you will evaluate whether the code has replicated the instructions about **{TASK}** experiments in the paper.
 Please summarise 5 key points. These 5 key points will be used to assess whether the code has completely replicated the model, methods, and experimental setting in the paper. Specifically, you are preferred to use 3 key points to summarise the proposed method, 1 for hyperparameters and 1 for training setup. Do not include the dataset generation process as a point, as the dataset has been preprocessed.
 You should regard each part of the proposed method in the paper as a separate key point. If there are formulas in the paper, you need to extract them in LaTeX format and use them as the criteria for judging whether the code has been replicated.
 Do not include some common content as the key points to be compared. Only include the key points related to the **{TASK}** task. If all these 5 points are replicated exactly, the code will fully replicate the paper. These key points are very important and should be as detailed as possible, which could reflect the key points to reproduce the paper.

Figure 2: Prompt for key points summarisation.

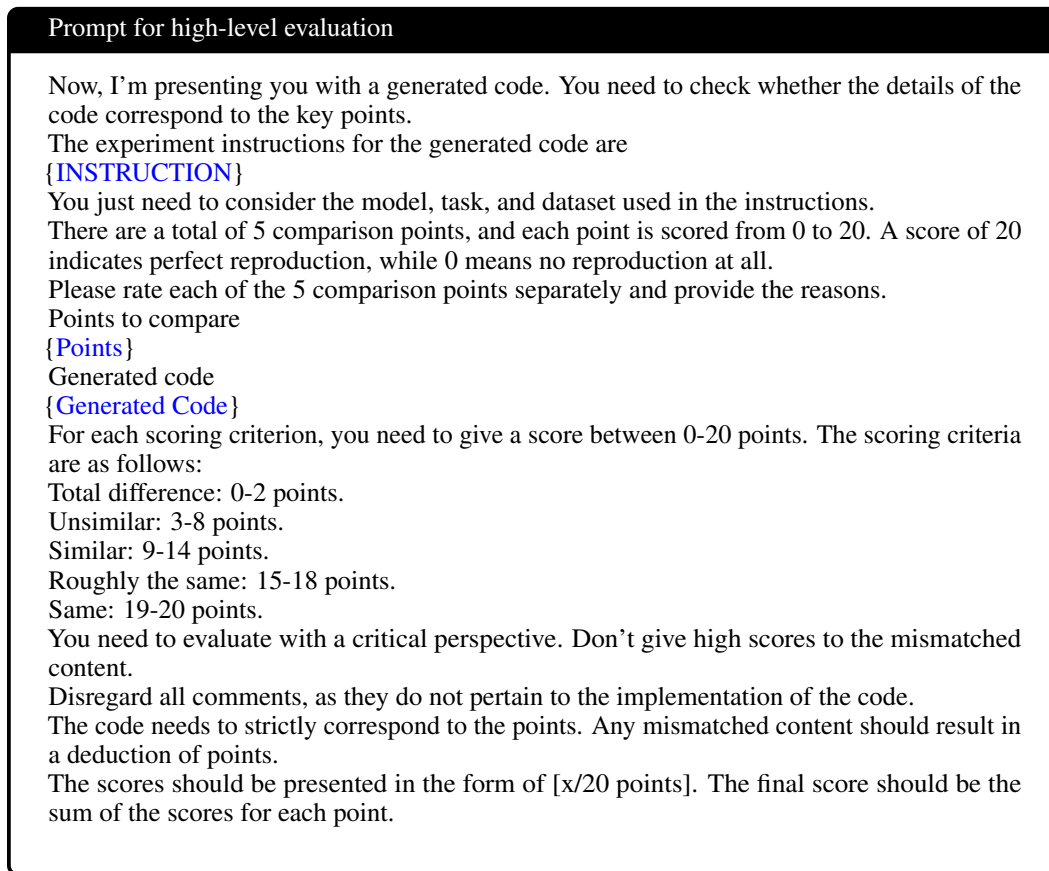


Figure 3: Prompt for high-level score. The **Points** are the 5 key points generated by the LLM judge, and the **Generated Code** is the code generated by LLM Agents.

Prompt for low-level evaluation

You are an expert proficient in code analysis, specialising in comparing and evaluating code structure and functionality. Now you need to judge whether the replicated code is the same as the official code.

Please analyse the following two code segments: the first is model-generated code, and the second is standard training code. You don't need to pay attention to the contents related to saving, printing and logging. Focus on the model itself and its training process instead.

Conclusion: Summarise whether the two code segments are completely equivalent in terms of model definition and Experimental Integrity, and briefly explain the significance of the scoring results.

First Code Segment (Standard Experiment Code):

{ [Reference Code](#) }

Second Code Segment (Model-Generated Code):

{ [Generated Code](#) }

Your task is to conduct a detailed comparison of these two code segments, focusing on the following aspects:

Overall Structure:

The overall structure of the model code includes the data flow in the forward function and the overall model structure.

Only consider the model-related code, such as the encoder-decoder structure, and ignore the data loading and other irrelevant code.

Model Structure:

Are the model architectures defined in both code segments (e.g., number of layers, activation functions, input/output dimensions) completely identical?

Specifically compare the implementation details, such as if a spatio-temporal processing module is present in the code, including but not limited to input processing, feature extraction methods, temporal dimension handling, spatial dimension handling, connection methods (e.g., residual connections, attention mechanisms), and parameter settings. Apply this level of comparison to all modules. Check for any subtle differences (e.g., convolution kernel size, pooling methods, normalisation techniques).

Training Details:

Compare whether the model hyperparameters (e.g., learning rate, batch size, optimiser type, learning rate decay strategy) are consistent. Verify whether the loss function's definition and implementation are identical, including the loss calculation formula and weight assignments.

Experimental Integrity:

Compare the implementation of the training process, including the training pipeline, data preprocessing, and gradient update logic, to determine if they are equivalent. The most crucial thing you need to pay attention to is the integrity of the experiment. Check for any functional differences (e.g., initialisation methods, early stopping mechanisms). There is no requirement for checkpoint saving, logging information and multi-GPU training. Do not consider these contents.

Pay special attention to analysing the implementation of the module in the model structure, ensuring a detailed comparison of each layer's specific parameters and computational logic.

If differences are found, clearly indicate the specific code lines or modules where they occur and analyse their potential impact on model performance or behaviour.

Ignore differences in code style (e.g., variable naming, comments) and focus on functionality and implementation logic.

You need to focus on the code implementation and don't need to consider comments and function names, and other contents irrelevant to the implementation.

Figure 4: Prompt for low-level score. The [Reference Code](#) and [Generated Code](#) are our curated official implementations and Agents' generated code, respectively.

Prompt for low-level evaluation

Scoring Criteria (Total: 100 points):

Overall Structure (25 points): Consistency in the overall structure of the model-related code, including the overall data flow pipeline in the forward function, and the overall modules are structured.

Model Details (25 points): Consistency in the implementation of the model structure. All the layers should be compared. If the names are different but the internal functions are the same, you should consider them as the same.

Training Details (25 points): Consistency in hyperparameters, loss function, learning rate decay, etc.

Experimental Integrity (25 points): Consistency in training loop, data processing, and other pipeline. You just need to compare the training, testing pipeline included in the code.

You need to analyse the code details and implementation before giving the score. Firstly, analyse the overall structure of the model and then analyse it specifically for each module.

For custom blocks, the details of how custom blocks are implemented should be analysed.

For some official implementation, you need to analyse whether the implementation of custom is the same as the official one based on your understanding. Ignoring the differences of programming languages and only considering the code implementation.

For each scoring criterion, you need to give a score between 0-25 points. The scoring criteria are as follows:

Total difference: 0-4 points.

Unsimilar: 5-10 points.

Similar: 12-16 points.

Roughly the same: 17-22 points.

Same: 23-25 points.

The MLP and a single linear layer should be considered roughly the same.

Similar should be when the functions of two codes are the same, but the implementations are different. Unsimilar should be when the functions of two codes are different.

You need to analyse the specific implementation of the code, do not just focus on the name. Make detailed evaluations based on the details. Ignore all the comments and focus only on the code.

The scores should be presented in the form of [x/25 points]. The final score should be the sum of the scores for each point.

Figure 5: Prompt for low-level score. The [Reference Code](#) and [Generated Code](#) are our curated official implementations and Agents' generated code, respectively.

Prompt for mixed-level evaluation

You are an expert code reviewer specialising in evaluating the fidelity of research paper implementations. Your task is to meticulously compare the generated code against a set of key points from a research paper. Crucially, you will use the provided reference code as the ground truth to understand how each key point is specifically and correctly implemented.

The experiment instruction is {INSTRUCTION} You just need to consider the model, task, and dataset used in the instructions.

Inputs You Will Be Provided With

1. Points: A list of key concepts, mechanisms, algorithms, or architectural features from the research paper that the generated code is supposed to implement.
2. Reference code: The official source code accompanying the research paper. This code serves as the benchmark for understanding the precise, intended implementation details of each key point.
3. Generated code: The generated code that needs to be evaluated for its accuracy in reproducing the key points as they are implemented in the reference code.

Your Evaluation Process: 1. Understand Key Point via Reference Code: For the key point, first, thoroughly examine the reference code. Identify and describe the specific segments of the reference code (e.g., functions, classes, logic blocks) that implement this key point. Summarise how the reference code realises this key point. This understanding will be your basis for comparison.

2. Analyse Generated Code against Reference Implementation: Now, review the generated code (generated code) to find its implementation of the same key point. Compare this implementation directly against your understanding of how it was done in the reference code. Focus on whether the logic, structure, and functional outcome are equivalent.

3. Score the Replication: Based on your comparative analysis, assign a score from 0 to 20 to the generated code for its replication of this specific key point, using the scoring rubric below.

4. Provide Detailed Justification: Clearly articulate the reasons for your score. Specifically highlight matches and discrepancies between the generated code's implementation and the reference code's implementation of the key point. Explain why it matches or why it deviates.

Scoring Rubric:

0-2 points (Total difference): The core innovation point (as demonstrated in the reference code) is not replicated at all in the generated code, or the implementation is fundamentally flawed, missing major functional aspects, or entirely incorrect when compared to the reference code.

3-8 points (Unsimilar): The key point is replicated in the generated code, but not completely accurately or comprehensively when compared to the reference code. Some aspects of the reference code's implementation might be present, but there are noticeable inaccuracies, missing details, or differences in logic that might affect functionality or deviate from the paper's intended mechanism, as shown in the official code.

9-14 points (Similar): The key point is replicated completely and accurately in the generated code. The implementation in the generated code closely mirrors the logic, structure, and functional behavior of the corresponding implementation in the reference code, although there might be some non-critical differences or alternative approaches that achieve the same core outcome.

15-18 points (Roughly the same): The key point is replicated completely, accurately, and comprehensively in the generated code. The implementation is highly consistent with the reference code in logic, structure, and function, differing only in very minor, non-functional ways that do not impact the core mechanism.

19-20 points (Same): The implementation of the key point in the generated code is identical or functionally equivalent to the reference code, representing a code-level copy or a near-perfect replication of the relevant sections.

Figure 6: Prompt for mixed-level score. The **Reference Code** and **Generated Code** are our curated official implementations and Agents' generated code, respectively.

Prompt for mixed-level evaluation

Critical Evaluation Guidelines:

Conduct your evaluation with a stringent and critical perspective. Do not award high scores for superficial similarities or implementations that do not match the essence of the reference code's approach for a given key point.

Your evaluation must be based solely on the executable code logic. Disregard all comments in both the reference code and the generated code as they do not pertain to the functional implementation.

At the same time, you need to pay attention not only to the key points themselves, but also to all the details related to them. If the key points correspond but the related implementations are different, it will still affect the reproduction effect. Please evaluate each key point in the points with the generated code. You should both determine the overall similarity and concrete scores. For example, when you decide the two codes are similar, you should also determine the level of similarity.

When evaluating specific implementations, prioritise the equivalence of core structure and function over superficial differences, such as the number of modules. If generated code achieves the same functional outcome and structural design as the reference, it should be considered equivalent, irrespective of modular composition.

Begin Evaluation:

Points to compare: {points}

Reference Code: {Reference Code}

Generated code: {Generated Code}

Output Format:

For each key point, please provide: 1. Key Point: *[Name/Description of the key point being evaluated]

2. Reference Code Implementation Summary: *[Your summary of how this key point is implemented in the reference code]

3. Generated Code Analysis & Comparison: *[Your detailed analysis of the generated code's attempt to implement this point, comparing it directly to the reference code's approach]

4. Score: *[x/20 points]

5. Reasoning for Score: *[Detailed justification based on the comparison]

Sum the overall scores for each key point to provide a final score out of 100 points, and include a summary of the overall evaluation.

Overall Score: *[x/100 points]

Figure 7: Prompt for mixed-level score. The [Reference Code](#) and [Generated Code](#) are our curated official implementations and Agents' generated code, respectively.