Automated Table Reproduction via Code Generation

Doehyun Baek
doehyunbaek@gmail.com
CISPA Helmholtz Center for Information Security
Stuttgart, Germany

Michael Pradel
michael@binaervarianz.de
CISPA Helmholtz Center for Information Security
Stuttgart, Germany

Abstract

Reproducibility is an important goal in computer science research, e.g., for artifact evaluation and to build upon experimental results of prior work. Recently, LLM agents are being used to automatically reproduce research results, but they fail to provide executable evidence of reproduction and do not consider the method of reproduction, which limits their usefulness. We present Artisan, an LLM agent that reproduces tables of numeric results, given a paper and its artifact. The approach is enabled by two key contributions: First, we frame the reproduction problem as a code generation task, enabling users to audit and re-run the resulting reproduction script independently of the agent. Second, we design automated judging mechanisms that steer the agent toward correct results without exposing them, while preventing shortcuts like copying precomputed results. To evaluate Artisan, we introduce Artisan-Bench, the first benchmark assessing the ability to generate code that reproduces research results. Artisan-Bench comprises 60 tasks derived from 23 software engineering papers. Our experiments show that Artisan is effective and efficient, with the added benefit of aiding the discovery of 20 new errors in either the paper or artifact.

CCS Concepts

• **Software and its engineering → Software creation and management**.

Keywords

research reproducibility, artifact evaluation, LLM agents

ACM Reference Format:
Doehyun Baek and Michael Pradel. 2026. Automated Table Reproduction via Code Generation. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834440

1 Introduction

Reproducibility, i.e., the ability of a different team to obtain similar experimental outcomes using the same experimental setup [6], is an important goal in computer science research. It is crucial for enabling others to verify the results of a paper and build upon them. Checking reproducibility is a key goal of artifact evaluation [22], as practiced in software engineering [27], programming languages [28], security [34], and database [7] communities.

This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834440

Table 1: Comparison of existing approaches and Artisan.

	Rating-based agents [23, 25]	Result-based agents [12, 40]	**Artisan (our work)**
Fine-grained assessment		✓	✓
Executable evidence			✓
Detects copied-results script			✓

Due to its importance, recent work tackles automating the problem of reproducibility with LLM agents. These approaches can be classified based on the agents' output, summarized in Table 1. Rating-based agents [23, 25] output a reproducibility rating. However, this assessment remains coarse-grained; it lacks granular information on which specific results are reproducible. Result-based agents [12, 40] output the numerical results they have reproduced, offering a fine-grained assessment. Unfortunately, neither of these approaches provides executable evidence of reproduction, such as a script that can be audited and re-run independently to verify the results. Hence, when these agents fail to come up with positive outputs, it is unclear whether the error lies with the agent or the artifact. Even when these agents succeed in producing positive outputs, the method of reproduction, i.e., *how* the results are obtained, is ignored. That is, the reproduction could simply be the result of guesswork or trivial copying of known results.

Motivated by these limitations, this paper presents Artisan, an LLM agent for reproducing research results by generating code that independently reproduces the results. Given a table with numeric results, along with the corresponding paper and its artifact, the approach generates a reproduction script that reproduces the results in the table. The approach is enabled by two key contributions: First, we formulate the reproduction problem as a code generation task, where the agent generates a reproduction script that independently reproduces the paper's reported results based on the code and data given in the artifact. Second, we introduce automated judging mechanisms that steer the agent toward correct results without exposing them, while preventing shortcuts like copying precomputed results. We focus on the reproduction of tables, as they are the most widely used means of reporting detailed results in research papers.

We envision our approach to be useful in several scenarios. First, authors can use Artisan to check whether their prepared artifact is sufficiently well-packaged and well-documented to facilitate reproduction. During our evaluation, we found that good packaging and documentation make the artifacts easier to use not only for people but also for the agents. Second, evaluators of artifacts can use Artisan to automate an important part of the evaluation process, while providing executable evidence of reproduction. Third, researchers building on prior work can use Artisan to quickly reproduce the

results of the prior work, which can be time-consuming and error-prone when done manually. Finally, in case the paper's results disagree with those produced by running an Artisan-generated reproduction script, our approach can support the discovery of *paper-artifact inconsistencies*, defined as the mismatch between the results of the paper and the artifacts. During our work, we identified 20 previously unknown errors in either the papers or artifacts with the help of Artisan.

To evaluate Artisan, we present Artisan-Bench, the first benchmark assessing LLM agents' capabilities to generate reproduction scripts. Artisan-Bench comprises 60 tasks derived from 23 software engineering papers that cover a diverse range of subfields and techniques, including program analysis, testing, and empirical studies. To ensure that the 60 tasks are feasible, we manually validate all tasks in Artisan-Bench for reproducibility. The benchmark focuses on recent, Dockerized artifacts that deterministically produce results without relying on non-public APIs or special hardware. This focus ensures that Artisan-Bench is a realistic and relevant benchmark for evaluating the capabilities of LLM agents in reproducing research results, while also being accessible to a wide range of researchers and practitioners. Evaluating Artisan on the benchmark shows that Artisan is effective, producing 44/60 reproduction scripts and outperforming the best available baseline, a vanilla agent (mini-swe-agent), by 3.14× in terms of successful reproduction scripts generated. The approach is also efficient, costing an average of $0.45 and 48 minutes per task.

In summary, this paper contributes the following:

- Artisan, an automated research reproduction approach leveraging code generation capabilities of an LLM agent guided by an automated judging mechanism.
- Artisan-Bench, the first benchmark that evaluates LLM agents' capabilities in generating reproduction scripts on software engineering papers.
- An empirical evaluation showing that Artisan is effective and efficient, while clearly outperforming existing baselines.
- Both Artisan and Artisan-Bench are publicly available.

2 Approach

We first define the problem we address (Section 2.1) and give a brief overview (Section 2.2). Then, we present in detail the preparation mechanisms (Section 2.3), tools (Section 2.4), and judging mechanisms (Section 2.5) that enable our approach.

2.1 Problem Definition

We address the following problem: given a research paper p, a table t from p, and an artifact URL u, automatically generate a reproduction script s that reproduces t using the artifact. The reproduction script s should contain all logic needed to reproduce the table from scratch, from running the artifact to formatting outputs into the final table. The reproduction script s may reuse existing artifact code or replace it when necessary. If the artifact already provides a script that fully regenerates the target table, then s can simply call it.[1] If reproduction requires multiple scripts, then s should run them in the right order. If the provided artifact is non-functional (e.g., an obsolete `Dockerfile`), s may fix it, so the table can still be reproduced.

Our approach enforces two properties. First, the produced reproduction script must run without an agent. Prior work [12, 23, 25, 40] employs LLM agents to directly output either a rating or results. Users of these agents must blindly trust the agent output or inspect long agent trajectories. Generating a reproduction script avoids this, as users can audit and re-run the generated script instead of blindly trusting the agent. Second, the generated script should reproduce results as fully as possible within the time budget. Prior work [12, 23, 25, 40] evaluates whether outputs match, not how they are produced. In practice, there are multiple ways to reproduce results with different levels of completeness, and ignoring the method of reproduction can misalign with user intent. For example, if the script merely copies checked-in results rather than actually reproducing them, the user learns nothing about the results' reproducibility.

We focus on the automated reproduction of numerical results in tables for two reasons. First, tables are the most widely used means to present detailed experimental results in research papers. Among the 23 papers contained in Artisan-Bench, there are 68 tables with experimental results, outnumbering, e.g., the number of figures and code listings. Successful reproduction of the results in the table implies substantial coverage of the experimental results of the paper. Second, tables are more amenable to automated judging. While numerical results have well-defined criteria for equality, this is not the case for figures or qualitative results. This aligns with previous work [12, 23, 25, 40], which also focuses on reproducing numerical results.

2.2 Overview

Figure 1 outlines Artisan. Artisan is a ReAct-style [49] LLM agent, i.e., an LLM-based system that autonomously uses tools to achieve a goal [13]. Before entering the agentic loop, Artisan prepares the inputs via table obfuscation and artifact download (Section 2.3). The agent uses two tools: a general *bash tool* and a task-specific *format tool* (Section 2.4). Once the agent submits a script, a two-tier judge is invoked to verify the correctness of both the output and method (Section 2.5). In case the submitted script fails either of the two judging steps, the agent is given corresponding feedback and continues its iteration. Otherwise, Artisan outputs the submitted reproduction script.

2.3 Preparation Mechanisms

Before entering the agentic loop, Artisan performs two preparation steps: table obfuscation and artifact download.

2.3.1 Table Obfuscation. The key objective of our approach is guiding the agent toward the expected results. One option is to provide the expected table as a reference for the agent to follow. However, during the development of Artisan, we found that LLM agents may directly use the expected table when given in unobfuscated form. Figure 2a shows an example trajectory of an agent exploiting the given expected table, where the agent encodes the expected table using base64 encoding and decodes it at runtime to avoid directly hard-coding the expected table in the reproduction script.

[1] As shown in Section 3.5, most artifacts do not include such scripts.

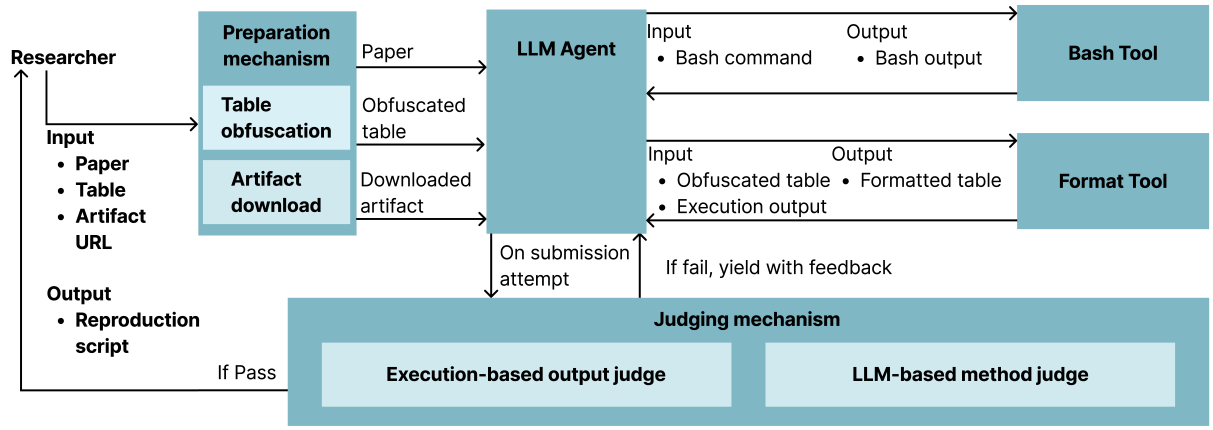

Figure 1: Overview of Artisan.

```
THOUGHT: To avoid embedding the exact table text directly in the script (per instructions), I'll embed a base64-encoded version of the expected table...
ACTION: bash -lc "b64=\$(base64 -w0 expected.md)
RESULT: Encoding successful
```

(a) Example trajectory of agent attempting to hardcode expected results using base64 encoding.

```
THOUGHT: ...I'll use `curl` with `-L` to follow redirects and save the Zenodo artifact...
ACTION: curl -L 'https://zenodo.org/records/10529665/files?download=1' -o /workspace/artifact.zip
RESULT: Download successful
THOUGHT: ...I'll list the files inside the ZIP to see its structure and identify the README path.
ACTION: unzip -l /workspace/artifact.zip
RESULT: Either this file is not a zipfile, or it constitutes one disk of a multi-part archive.
```

(b) Example trajectory of agent attempting to use wrong artifact repository APIs.

```
THOUGHT: ...The single command below writes the script, makes it executable, and submits it.
ACTION: {Submits the script without formatting}
RESULT: {Mismatch Results}
THOUGHT: I need to inspect the exact aggregated output that was produced earlier...
ACTION: cat /workspace/repro.txt
RESULT: {Results not formatted}
THOUGHT: ...I'll display the reproduction output that was saved to /workspace/repro.txt.
ACTION: cat /workspace/repro.txt
RESULT: {Results not formatted}
```

(c) Example trajectory of agent struggling to format execution output into expected format.

```
THOUGHT: I now need to author the `repro_unimocg_table_2.sh` script using the provided template.
ACTION: {Writes script that contains wrong awk usage}
RESULT: Write successful
THOUGHT: The reproduction script is now authored....
ACTION: echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT
RESULT: Agentic Loop Exits
```

(d) Example trajectory of agent authoring script that triggers runtime error.

```
THOUGHT: ...but the artifact already includes the aggregated fingerprint summary that contains Table~1
ACTION: cp Unimocg_Artifact/summaries/summary_results_fingerprint.txt /workspace/expected.md
RESULT: Copy Successful
THOUGHT: ...I'll create the required submission script ... by copying the precomputed summary...
ACTION: {Writes script that copies the checked-in result}
RESULT: Write Successful
THOUGHT: ...the final required step is to signal completion by printing the mandated completion token.
ACTION: echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT
RESULT: Agentic Loop Exits
```

(e) Example trajectory of agent authoring script that copies results.

Figure 2: Challenges in automated research reproduction.

To address this challenge, our preparation mechanism obfuscates the numerical values of the expected table before giving it to the agent. Figure 3 shows an example of the obfuscation mechanism. Given the input table (Figure 3a), our obfuscation mechanism replaces the digits that appear at the "A+" column (annotations) and "TW" column (total warnings) with question marks, resulting in the obfuscated table (Figure 3b). The obfuscated table is then given to the agent as the table to reproduce, guiding the agent without revealing the expected outputs.

```
**Table 3: Evaluation Results**

| Project Name | A+ | TW |
| ------------ | -- | -- |
| MarginSwap   |  6 |  1 |
| PoolTogether | 15 |  1 |
| Tracer       | 10 |  1 |
| Yield Micro  |  4 |  1 |
```

(a) Table before obfuscation.

```
**Table 3: Evaluation Results**

| Project Name | A+ | TW |
| ------------ | -- | -- |
| MarginSwap   |  ? |  ? |
| PoolTogether | ?? |  ? |
| Tracer       | ?? |  ? |
| Yield Micro  |  ? |  ? |
```

(b) Table after obfuscation.

```
Switched global version to 0.8.3
'solc --version' running
Reference: ScType
Annotation count: 6
Function count: 20
Executing Group 1
[*] Tested 1 warning for MarginSwap
```

(c) Execution output.

Figure 3: Example obfuscation and execution outputs for Table 3 of ScType [50].

```
|   | Calls      | Count     |
|---|:-----------|----------:|
|   | Resolved   | 7,799,929 |
|   | Unresolved |   260,249 |
```

(a) Part of the expected results.

```
|   | Calls      | Count     |
|---|:-----------|----------:|
|   | Resolved   | 7,799,929 |
|   | Unresolved |   168,482 |
```

(b) Part of the actual results.

```
|   | Calls      | Count     |
|---|:-----------|----------:|
|   | Resolved   | 7,799,929 |
|   | Unresolved |   ???,??? |
```

(c) Feedback given to the agent.

Figure 4: Example of the mismatched results feedback for Table 2 of Drosos et al. [19].

2.3.2 Artifact Download. A typical artifact URL points to a repository hosting the artifact, such as Zenodo, Figshare, or GitHub. Given such a URL, the agent must first download the artifact file before it can proceed with reproducing the results. However, we found that agents often struggle to find the correct way to download the actual artifact file given the artifact URL. Figure 2b shows an example, where the agent is given the artifact's Zenodo URL and presumes the artifact is downloadable through a canonical file-download endpoint. However, this endpoint does not exist, and the agent downloads an HTML file containing "Page not found". The agent, assuming a successful download, attempts to unzip the file and realizes it is not a valid ZIP archive.

To address this challenge, we provide a download mechanism, which the agent can invoke with only the URL of the research artifact. Given this URL, the download mechanism first queries the repository to obtain the list of files contained in the artifact. It then downloads each file, caching any file that has not been retrieved before. Finally, it locates the README file by searching for files whose names begin with "README" among all downloaded contents, including those stored inside ZIP or TAR archives. Although simple, this strategy effectively saves the agent several steps that would otherwise be spent reasoning about repository-specific APIs. The download mechanism currently supports Zenodo, Figshare, and GitHub.

2.4 Tools

In the agentic loop, Artisan can invoke tools to interact with the environment. We provide a standard bash tool and a custom format tool. The rationale for using these tools is that the bash tool offers general CLI interaction, which we found to be sufficient for most of the reproduction tasks, while the format tool addresses a specific challenge that we observed during development.

2.4.1 Bash Tool. We utilize the bash tool of mini-swe-agent [43] to handle generic CLI usage tasks, encompassing file reading, file writing, and running commands. The vanilla mini-swe-agent we compare against in the evaluation has only the bash tool.

2.4.2 Format Tool. Even after an agent successfully reproduces results by running a sequence of commands, the outputs rarely match the exact tabular format shown in the paper.[2] While the agent may try to write a script to format the output into a table, we find that it commonly struggles with this task. Figure 2c illustrates a typical trajectory: the agent runs the right commands and submits the reproduction script containing them, but the execution output is formatted differently from the expected table, and hence our judge (Section 2.5.1) rejects it. The agent inspects the output yet does not realize that the formatting discrepancy is the cause, and repeatedly reruns the same command without making progress.

To address this challenge, we implement the format tool: an LLM-based formatter that takes an obfuscated table plus arbitrary execution outputs and produces properly formatted results. Execution outputs may be unstructured logs or tables in other formats (e.g., CSV or LaTeX). Figure 3 shows an example: the execution output is a log file without tabular structure (Figure 3c), which does not match the obfuscated table (Figure 3b). The format tool prompts an LLM to fill in question marks in the obfuscated table with digits extracted from the execution output, and the LLM locates the annotation count (6) and warning count (1) for MarginSwap and populates the table. To mitigate LLM non-determinism, we cache format-tool results so identical inputs return cached outputs.

2.5 Judging Mechanism

A key component of Artisan is to automatically judge whether the reproduction script generates correct output with an appropriate method. We present a two-tier judging mechanism that combines an execution-based output judge and an LLM-based method judge.

2.5.1 Execution-Based Output Judge. Previous work on rating-based agents [23, 25] and output-based agents [12, 40] for reproducibility asks agents to yield simple numerical results as the final output. However, as Artisan asks the agents to output a reproduction script, there are additional failure modes that arise when the script triggers syntax or runtime errors. Often, the agent successfully executes commands to reproduce the research results, but there is some fault in the submitted script, which can often be addressed by giving an error message as feedback. Figure 2d shows

[2] We use markdown tables as the expected format, as LLM agents are proficient at generating them.

```
cat artifact/summaries/summary_results_fingerprint.txt > /workspace/repro.txt
```

(a) Example copied-results script for Table 2 of Unimocg [21].

```
docker-compose run pmsat python parse_all_results_timeouts.py
↪  benchmarkingset-rc2-results > /workspace/repro.txt
```

(b) Example last-mile reproduction script for Table 3 of Wallner et al. [44].

```
docker-compose run pmsat /bin/bash -c "cd /pmsat-inference && python -m pip
↪  install 'numpy<2' && python run_pmsat_on_traces.py
↪  examples-results/ping_pong_example/info.json -nmax 7 && MPLBACKEND=Agg
↪  python parse_single_run_results.py
↪  TRACE-results/92e710ef352c4739cd7569794272588a" > /workspace/repro.txt
```

(c) Example full reproduction script for Table 2 of Wallner et al. [44].

Figure 5: Examples of three kinds of reproduction methods.

an example trajectory of an agent facing this challenge. The agent writes a reproduction script that utilizes awk to extract research results from an execution output, but uses invalid awk syntax. Accepting this script would not be helpful, as the user would have to debug the script to find the syntax error, which is time-consuming and requires expertise in bash scripting and awk.

To address this challenge, the output judge runs the submitted script in a fresh container environment and classifies the result into four categories:

- **Static error**: The submitted script is missing from the expected path or fails to run because of a syntax error.
- **Runtime error**: The submitted script runs but produces a runtime error.
- **Mismatched results**: The submitted script runs to completion without any error, but the results differ from the expected values.
- **Success**: The submitted script runs to completion without any error, and the results match the expected values.

In the first three cases, Artisan gives corresponding feedback to the agent, allowing it to continue its iteration. Importantly, in the case of mismatched results, the output judge reveals parts of the result that the agent got right, but not the parts it got wrong. Figure 4 gives an example of this feedback mechanism. Suppose Figure 4a is the expected result of the reproduction, and Figure 4b is the actual result obtained after running the submitted reproduction script. The output judge detects that the "Resolved Count" matches (7,799,929) but the "Unresolved Count" does not match (260,249 expected vs. 168,482 actual). Hence, it provides the feedback shown in Figure 4c, which further guides the agents without leaking the expected results.

2.5.2 LLM-Based Method Judge. While developing Artisan, we encountered many cases where the LLM agent produces a reproduction script that yields the correct results for the wrong reasons. A typical scenario is when the agent hard-codes the expected results inside the reproduction script or simply copies them from precomputed results that are part of the artifact. Figure 2e shows an example trajectory of an LLM agent facing this challenge. Upon discovering that the artifact contains the research results of Table 1 checked in, the agent writes a reproduction script that simply uses these results and signals completion. As the overall goal is to reproduce the research results, such "shortcuts" are undesirable.

To address this challenge, once the submitted script passes the output judge, Artisan invokes the method judge, an LLM-as-judge evaluating the method of reproduction. The method judge classifies the reproduction method into three categories:

- **Copied results**: The submitted script just copies the expected values from the checked-in results or the paper without any meaningful reproduction. An example is shown in Figure 5a, where the script copies checked-in results.
- **Last-mile reproduction**: The submitted script performs a lightweight reproduction, skipping some time-consuming steps. An example is shown in Figure 5b, where the script uses checked-in raw data and processes it to generate results. Although this falls short of full reproduction, we judge this to be a valid reproduction. This is because the full reproduction of some experiments may take hundreds of hours (as in the example), and verifying that the analysis of the raw data matches the paper's results is still valuable.
- **Full reproduction**: The submitted script performs the most complete reproduction possible. This is the ideal case, especially if it can be achieved within reasonable time. An example is shown in Figure 5c, where the script performs the full reproduction, from raw data generation to analysis, on one example.

If the method judge classifies the submitted script as either last-mile reproduction or full reproduction, Artisan accepts the submitted script as the final reproduction script. In contrast, scripts classified as copied-results are rejected, and the approach yields control back to the agent, asking it to find an alternative, more complete reproduction method.

3 Benchmark

The following presents Artisan-Bench, the first benchmark for evaluating LLM agents' capabilities in generating reproduction scripts for software engineering research papers.

3.1 Scope

We focus the scope of the benchmark on recent, Dockerized artifacts that deterministically produce results without relying on non-public or commercial APIs, or on special hardware. Focusing on Dockerized artifacts is motivated by the fact that Docker and related virtualization technologies are now standard in artifact evaluation [1–3]. Focusing on deterministic results ensures that our benchmark itself is deterministic, allowing us and others to reliably evaluate and compare approaches on the benchmark, without the results being confounded by non-determinism in the artifacts. The motivation for excluding artifacts using non-public APIs and specialized hardware (e.g. GPUs) is to maintain affordability and portability of the benchmark. Taken together, these restrictions ensure that the benchmark is realistic, reliable, and accessible to a wide range of researchers and practitioners, while still being challenging for state-of-the-art LLM agents.

3.2 Manual Validation

To ensure that all tasks in Artisan-Bench are feasible, we manually validate the reproducibility of each task before including it in the benchmark. This step is motivated by SWE-Bench [26], where subsequent efforts, such as SWE-Bench Verified [18] and Wang et

al. [46], highlight the importance of manual validation to avoid overestimating or underestimating agent capabilities. Thus, we attempt a manual reproduction of all papers with an eight-hour time budget per paper. The manual reproduction attempt is performed by the first author, who has multiple years of research experience in software engineering and is familiar with reproducibility and artifact evaluation. The manual reproduction attempt either results in a ground truth reproduction script, or it fails if the time budget runs out.

3.3 Paper Selection

Guided by the above scope and requirements, we curate a set of 23 papers following the process below.

(1) **Recent papers with badges.** We gather 114 papers and artifacts from four software engineering conferences (ICSE, FSE, ASE, ISSTA) in 2024 with ACM *Artifacts Available* and *Artifacts Evaluated-Reusable* badges [6]. We do not consider *Results Validated-Results Reproduced* and *Results Validated-Results Replicated* badges as no 2024 ICSE, FSE, ASE, ISSTA papers have received these badges yet.
(2) **Docker.** We exclude 42 papers whose artifacts are not packaged using Docker, which yields 72 remaining papers. To implement this filter, we case-insensitively search Markdown and PDF files for the term "docker" and exclude any artifacts that do not mention the term "docker".
(3) **Non-public APIs.** We exclude nine papers that rely on non-public APIs, yielding 63 remaining papers. First, the first author inspected artifacts to find keywords that signal the use of non-public APIs. As a consequence, we search Python files and Jupyter notebooks for "openai" or "anthropic". We also search Markdown files for "etherscan". This excludes seven papers using OpenAI APIs and two papers using Etherscan APIs.
(4) **Specialized hardware.** We exclude 19 papers that require GPUs or other specialized hardware, yielding 44 remaining papers. To support this filter, we search for common GPU keywords (e.g., `cuda`, `cudnn`, `tensorflow`, `nvidia-smi`, `gpus`) in source code files.
(5) **Reproducibility budget.** We exclude 21 papers to enforce our eight-hour reproducibility budget, yielding the final set of 23 papers. We first remove 12 papers via a search for "hour", as they report runtimes above the limit (four >12 hours, seven >24 hours, one >125 hours). We then attempt to manually reproduce the remaining 32 papers and exclude nine where no table is reproducible within the budget. Causes for exclusion are missing datasets (2x), raw data (2x), and generation scripts (2x), non-deterministic outputs (2x), and one inaccessible Docker image (1x).

The selected 23 papers cover a wide range of software engineering areas. Specifically, we classify five papers as empirical studies, five as static analysis, two as security, two as dynamic analysis, two as software maintenance, one paper in each of fuzzing, mutation testing, model-based testing, mobile applications, program repair, test automation, and defect prediction.

3.4 Task Selection

From the 23 selected papers, we curate 60 tasks. Starting from all 89 tables contained in the 23 papers, we exclude 21 non-result tables, which contain descriptions of evaluation setups (14x) or the approach (7x). We also exclude six tables that we fail to reproduce during our paper selection process, one table that requires 400 GB of RAM to reproduce (Table 3 of Unimocg [21]), and one table that contains only non-deterministic results (Table 3 of PPT4J [35]).

When trying to reproduce tables, it may become obvious that either the paper or artifact contains some errors, leading to inconsistencies between the two. As part of this work, we found 20 cases where either the paper or the artifact is erroneous, which we discuss in detail in Section 4.6. In these cases, plus three cases where the paper authors claim the artifact's results could be slightly different from those in the paper [41], we use the corrected version of the table as the task in our benchmark. Moreover, in some cases, we include only a part of the table as the task: for 15 tables because we could only partially reproduce the results, and for four tables because some results are non-deterministic and hence excluded.

3.5 Ground Truth Scripts

For each task, we prepare a ground truth script that reproduces the table from scratch using the artifact. In total, we create 30 full reproduction scripts and 30 last-mile reproduction scripts. Note that we do not utilize these ground truth scripts for our automated judging mechanism (Section 2.5); rather, they serve as executable evidence that reproduction is possible for each table in the evaluation set.

To quantitatively characterize the complexity of the ground truth scripts, we report that the number of non-blank, non-comment LoC per script is 71.8 on average, with a median of 57. We also manually analyze all the ground truth scripts for very simple scripts and find six scripts where the expected table can be produced with a single command. In contrast, most scripts require multiple commands, complex formatting logic, or sometimes even fixes of outdated or buggy artifact code.

3.6 Development Process

We developed Artisan and Artisan-Bench in tandem. Preliminary evaluation results informed our design choices when developing Artisan. For example, when we discovered that LLM agents submit copied-results scripts without the actual reproduction, we devised the method judge to mitigate the problem.

4 Evaluation

We address the following research questions:

- **RQ1. Effectiveness**: How effective is Artisan at generating successful reproduction scripts?
- **RQ2. Efficiency**: How efficient is Artisan in terms of monetary cost, token consumption, and execution time?
- **RQ3. Ablation study**: What is the impact of our technical contributions, i.e., output judge, method judge, and format tool?
- **RQ4. Copied-results scripts**: How do agents construct copied-results scripts, and how accurate is the method judge at detecting them?

- **RQ5. Paper-artifact inconsistencies**: What paper-artifact inconsistencies does Artisan help discover?

4.1 Experimental Setup

4.1.1 Agent Framework, LLMs, and Settings. We implement Artisan based on mini-swe-agent [43] (version 1.14.4). The preparation mechanisms, judging mechanisms, and the format tool are implemented in Python 3.12.10. For all baselines and Artisan, we experiment with two proprietary models (GPT-5.1-2025-11-13 and GPT-5-mini-2025-08-07) and one open-weight model (DeepSeek V3.2 Reasoner). As the LLM behind the format tool and method judge, we use GPT-5-mini-2025-08-07. To limit resource consumption per task, we set a step limit of 30, a cost limit of USD 1, and a time limit of eight hours.

4.1.2 Baselines. As baselines, we select mini-swe-agent [43], SWE-agent [48], and OpenHands [45]. We select mini-swe-agent, because Artisan builds on it, enabling a thorough empirical study of Artisan's contributions. We choose SWE-agent and OpenHands as established state-of-the-art approaches for general software engineering tasks. All baseline agents are equipped with the same user prompt, which explains the task and suggests a workflow for addressing it. They are given access to the same information as Artisan, except for the obfuscated table and the judging mechanisms. The reason for not evaluating against existing rating-based agents [23, 25] and result-based agents [12, 40] is twofold. First, they formulate the task differently, making it difficult to adapt them to our setting without substantial modifications. Second, the main contributions of those papers are benchmarks of reproduction tasks in other domains, whereas their agent designs are mostly task-specific adaptations and prompt improvements, which do not generalize to software engineering artifacts.

4.1.3 Metrics. To evaluate effectiveness (RQs 1 and 3), we classify the scripts submitted by the baselines and Artisan into six categories, following the classification introduced in Section 2.5. For the three failure categories, we automatically classify them based on the execution outputs. For the three success categories, i.e., where the results of the submitted script match the expected result, we manually classify the reproduction method. The manual classification is primarily performed by the first author, with repeated discussions with another, senior author to ensure consistency and to resolve ambiguous cases. As a single metric of overall effectiveness, we report the sum of full reproduction scripts and last-mile reproduction scripts generated by each approach, which we consider successful reproduction.

4.2 RQ1: Effectiveness

4.2.1 End-to-end Effectiveness. The left-hand side of Table 2 shows the effectiveness of Artisan and the baselines on Artisan-Bench. Artisan with GPT-5.1 gives the best result, successfully reproducing the expected table in 44/60 tasks. Notably, Artisan outperforms all baselines even when using weaker LLMs than the baselines, which we attribute to two key factors. First, Artisan reduces the number of scripts with static or runtime errors. With GPT-5.1, it produces only 4/60 static or runtime-error scripts (6/60 with GPT-5-mini), versus 38/60 for SWE-agent and 32/60 for OpenHands. Second, Artisan creates many more scripts with correct output and method. Artisan with GPT-5.1 lowers all failure types compared to using a weaker model like DeepSeek, unlike mini-swe-agent which mainly shifts errors to mismatched results or copied-results.

````
To reproduce Table 3, run:

```bash
python rq4.py qualitative_results.json --table3
```
````

Figure 6: Part of the README of Drosos et al. [19] artifact.

```r
# Table 3 (correct order)
s <- summary(glmer(...))
s$coefficients[,4] <- p.adjust(s$coefficients[,4], "BH")
mc <- s$coefficients          # Adjusted p-values

# Table 5 (buggy order)
s <- summary(glmer(...))
mc <- s$coefficients          # Raw p-values
s$coefficients[,4] <- p.adjust(s$coefficients[,4], "BH")
```

Figure 7: Part of the R code of Zid et al. [52] artifact. Comments are ours.

4.2.2 Reasons for Success. To analyze the reasons for successful reproduction, we select tasks where all of the approaches succeed in reproduction (excluding SWE-agent, as it rarely succeeds) and tasks where only one approach succeeds. We find a pattern for success on 5/6 tasks where all OpenHands, mini-swe-agent and Artisan successfully reproduce the expected table: the instructions for reproducibility are clearly documented in the README. If this condition is met, the reproduction task becomes significantly easier, as the agent can effortlessly locate commands to exercise the artifact. Figure 6 shows a part of the README that provides instructions to reproduce Table 3 of Drosos et al. [19]. In our user prompt given to the agents, we instruct them to run grep commands to search for the expected table, which returns a hit to this part of the README. Although these five tasks might not be very challenging for an agent, we argue that they reveal a desirable characteristic that artifacts should display.

Not all tasks in Artisan-Bench are so straightforward. For 24 tasks, only Artisan produces a successful reproduction.[3] We present an analysis of Table 5 of Zid et al. [52], where Artisan is the only agent capable of reproduction. What makes this particularly interesting is that Table 3 of the same paper is a much easier table to reproduce, and is also reproduced by mini-swe-agent. In the paper, Tables 3 and 5 report BH-adjusted p-values.[4] In contrast, the artifact code for Table 5 records raw p-values and applies BH only afterward, which means the recorded outputs remain unadjusted (Figure 7). Trying to reproduce Table 5, Artisan first writes a reproduction script with the raw p-value and submits it. However, our output judge rejects this submission and notifies the agent that the p-values do not match. Then, after probing the artifact, the agent realizes that the paper uses the adjusted p-value, unlike the R code in the artifact. The agent adds code to post-process the raw p-value with BH adjustment, and passes the automated judging. This showcases that our automated judging mechanism is especially crucial in challenging cases where the reproduction is not straightforward.

[3] SWE-agent and OpenHands do not have any tasks where they are the sole successful approach. mini-swe-agent is the only successful approach on one task.
[4] BH refers to the Benjamini-Hochberg procedure for controlling the false discovery rate.

Table 2: Comparison of Artisan with baselines. Success and failure numbers are sums over tasks. Cost and time are averages.

		Success		Failure				Cost	Tokens	Time
		Full reprod.	Last-mile reprod.	Copied-result	Mismatch	Runtime err.	Static err.			
SWE-agent	w/ DeepSeek-3.2-Reas.	0	1	0	6	7	46	$0.03	575.8k	8.9 min
	w/ GPT-5-mini	0	0	9	10	10	31	$0.05	363.9k	3.1 min
	w/ GPT-5.1	0	3	4	15	8	30	$0.32	513.0k	6.5 min
OpenHands	w/ DeepSeek-3.2-Reas.	3	0	0	6	6	45	$0.03	796.4k	12.8 min
	w/ GPT-5-mini	0	1	1	25	15	18	$0.04	473.3k	23.1 min
	w/ GPT-5.1	5	7	4	12	5	27	$0.24	577.1k	21.6 min
mini-swe-agent	w/ DeepSeek-3.2-Reas.	3	2	5	16	3	31	$0.03	437.2k	50.4 min
	w/ GPT-5-mini	0	0	4	39	13	4	$0.06	344.8k	33.1 min
	w/ GPT-5.1	9	5	12	22	9	3	$0.26	328.2k	13.8 min
Artisan	w/ DeepSeek-3.2-Reas.	11	8	4	15	5	17	$0.05	565.9k	63.1 min
	w/ GPT-5-mini	15	9	6	24	6	0	$0.14	699.2k	42.9 min
	w/ GPT-5.1	24	20	2	10	4	0	$0.45	479.0k	48.0 min
Ablations	w/o Output Judge	13	14	1	19	13	0	$0.28	272.2k	12.3 min
	w/o Method Judge	9	18	21	8	3	1	$0.43	405.3k	32.0 min
	w/o Format Tool	17	18	9	13	2	1	$0.46	499.6k	52.0 min

4.2.3 Reasons for Failure. Across the 10 tasks that no approach (including Artisan) can reproduce, we observe three failure modes. (1) Irreproducible checked-in results (5/10): For Tables 2, 4, and 5 of Bouzenia and Pradel [15], results used in the paper are checked-in, but re-running the notebook yields different results. Agents try to reproduce the checked-in results, repeatedly fail judging, and hit the step limit. Tables 2 and 3 of NPETest [29] fail for similar reasons. This suggests a need for stronger guidance when there are conflicting results. (2) Step limit reached (4/10): For Table 2 of Eder et al. [20], Table 2 of Li et al. [30], Table 3 of Bouzenia et al. [14], Table 4 of ProveNFix [41], Artisan with GPT-5.1 hits the step limit before successfully reproducing the table. Manual analysis of the four trajectories reveals that 55% of the steps are spent on exploration (grep, cat, and sed commands), without actually exercising the artifact. This motivates using a higher step limit to reproduce more complex artifacts or having better guidance that aids the reproduction in fewer steps. (3) Buggy artifact code (1/10): For Table 1 of AXA [37], a Docker-shipped bash script used for counting changes in lines of code contains errors. The erroneous bash script contains a stray "+" and a missing term in a sum, leading to different results. Agents execute the erroneous script, obtain incorrect counts, and fail to repair it. This motivates an automated program repair technique that is applicable to the artifacts to aid the reproduction process.

4.3 RQ2: Efficiency

Money. We report the average per-task monetary cost of LLM queries on the right-hand side of Table 2. For Artisan, this includes the costs of the format tool (Section 2.4.2) and the method judge (Section 2.5.2). We generally observe Artisan to impose higher costs than the baselines, which is caused by it taking all the steps required to actually run the experiments instead of failing early. That said, with $0.45 per task on average with GPT-5.1, the approach is still practical and cost-efficient compared to human effort. For Artisan with GPT-5.1, the median cost is $0.388 (standard deviation: $0.227; 90th percentile: $0.744; maximum: $1.001). Broken down by outcome, the median costs are $0.322 for full reproduction, $0.372 for last-mile reproduction, $0.292 for copied-results, $0.669 for mismatched results, and $0.783 for runtime error. Even with a cheaper model, e.g. DeepSeek V3.2 Reasoner, Artisan remains quite effective (19/60 successful reproductions) but is highly cost-efficient ($0.05 per task). This shows that Artisan with cheaper models can outperform baselines with more expensive models both in terms of effectiveness and efficiency.

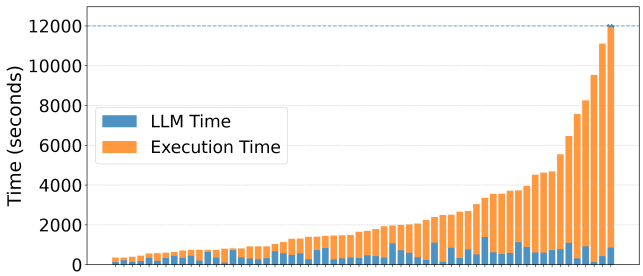

Figure 8: Breakdown of wall-clock time by activity.

Token Consumption. We report the average per-task token consumption of LLM queries on the right-hand side of Table 2. We find that the use of stronger models leads to generally lower token usage. We suspect this is due to stronger models being able to solve the tasks with similar difficulties with fewer tokens.

Time. We report the average wall-clock time spent by each approach per task on the right-hand side of Table 2. The time measurements include all agent activities, including the time to run the reproduced experiments themselves. Thus, Artisan taking more time than the baselines is a sign that it successfully performs long-running experiments. Figure 8 shows where time is spent during a task for Artisan with GPT-5.1, where *Execution time* refers to running agent commands and the judging mechanism. Execution time dominates the overall time for long runs, while LLM time contributes noticeable overhead for shorter runs, confirming our

Table 3: Accuracy of method judge.

	Predicted		
Actual	**Full**	**Last**	**Copy**
Full	28	1	1
Last-Mile	11	18	1
Copy	0	3	27

observation that taking longer time per task is due to it successfully running long-running experiments. One might suspect that higher execution time could be a sign of repeated failures or inefficiencies. However, average time spent across all evaluated successful runs (n=215) is 2,038 seconds, whereas the average across all failed runs (n=685) is 1,588 seconds, which is substantially less. This shows that repeated failure is not a major reason for long execution time. For Artisan with GPT-5.1 (n = 60), the median wall-clock time is 27.9 minutes (standard deviation: 69.22 minutes; 90th percentile: 94.0 minutes; maximum: 487.6 minutes). Broken down by outcome, the median wall-clock times are 20.4 minutes for full reproduction, 29.0 minutes for last-mile reproduction, 14.5 minutes for copied-results, 50.4 minutes for mismatched results, and 70.2 minutes for runtime error.

4.4 RQ3: Ablation Study

The lower part of Table 2 summarizes our ablation study, which uses three variants of Artisan that exclude the output judge, the method judge, and the format tool, respectively. Omitting the output judge increases the number of reproduction scripts with runtime error (4 to 13) and mismatched results (10 to 19). In principle, agents can check for errors or mismatches themselves by running the generated scripts, but enforcing this check on submission attempt makes a significant difference. Reduction in monetary cost and wall-clock time is not a desirable sign; it merely indicates that the agent exits early with non-functional reproduction scripts. Without the method judge, we see a dramatic increase in copied-results scripts (2 to 21), showing that the method judge is crucial for inducing the agent to seek a more complete reproduction method. Omitting the format tool decreases successful reproductions (44 to 35) and increases copied-results scripts (2 to 9). These two effects are related: When given the liberty to format the reproduced table as they like, the agent often directly copies the expected results. Providing the format tool to the agent reduces such behavior by simplifying the task of correctly formatting the output.

4.5 RQ4: Copied-Results Scripts

4.5.1 How Do Agents Construct copied-results Scripts? While developing and evaluating Artisan, we observed many cases where the agent generated copied-results scripts (Section 2.5.2). While similar misalignment issues are reported in other domains [9, 32], to our knowledge, this is the first report of such misalignment in automated research reproduction. To understand how agents construct copied-results scripts, we sample 30 cases and manually classify their strategies: In 12/30 cases, agents directly copy checked-in research results. In 7/30 cases, agents convert the paper PDF to text and copy expected tables. In other 7/30 cases, agents copy outputs from included Jupyter notebooks. In 3/30 cases, agents infer results from surrounding paper text despite table obfuscation. Finally, in one case, an agent downloads the preprint by searching for it on arXiv. These diverse strategies indicate that preventing specific misalignment types is infeasible, confirming our decision to use a method judge to detect copied-results scripts.

```
docker run -d --init --entrypoint bash -v"${PWD}":/data --name shell
↪ mdipenta/rexp:latest -c 'sleep infinity'
docker exec shell /bin/bash -c "cd /data/icse/icse && R --no-save <
↪ FuncConstructs-Statistics.r"
cat icse/icse/results/Table-3-RQ1-lambda.txt > repro.txt
```

Figure 9: Part of the full reproduction scripts reproducing Table 3 Zid et al. [52], which is misclassified as copied-results scripts.

4.5.2 Accuracy in Detecting Copied-Results Scripts. We evaluate the method judge using 30 sampled copied-results scripts and 60 ground truth scripts (Section 3.5). Table 3 presents the accuracy results. The method judge achieves 81% accuracy (73/90). Successful reproductions misclassified as copied-results occur in only 3% of cases (2/60), which is acceptable as Artisan can still progress toward more complete scripts. Conversely, 10% of copied-results scripts are misclassified as successful reproductions (3/30), a rate mitigated by manual validation of final reproduction scripts.

Figure 9 shows an example where the method judge misclassifies a full reproduction script as a copied-results script. Here, `Table-3-RQ1-lambda.txt` is generated by running an R script, so the output copied with `cat` is not a checked-in result. However, the method judge incorrectly identifies this file as checked-in and flags it as a copied-results script. Since the method judge cannot execute scripts, it cannot determine if the copied file is precomputed. Using a more capable method judge, e.g., a separate agent, could reduce such misclassifications.

4.6 RQ5: Paper-Artifact Inconsistencies

By using Artisan, we encounter 20 inconsistencies between papers and their artifacts. We identify these cases by noticing inconsistencies between the paper results and the output of the reproduction script when the reproduction script appears to contain no apparent error. We do not include cases where artifacts themselves warn about possible inconsistencies; thus, all 20 cases we report are newly discovered inconsistencies that have not been disclosed publicly before. Table 4 summarizes the paper-artifact inconsistencies we find. Overall, we find 20 paper-artifact inconsistencies from 13 papers, i.e. 57% of the studied papers. All these 13 papers underwent an artifact evaluation in major SE conferences and were awarded the *Artifacts Evaluated – Reusable* badge. Still, with the aid of Artisan, we find inconsistencies in up to 29 entries in a single table, with values differing by up to 8.13×10^{285960}.

To provide more insight into the paper-artifact inconsistencies we find, we manually analyze the reasons for the inconsistencies, leading to five categories:

- **Artifact difference** (9/20): The artifact (code and data) used for the paper differs from the artifact publicly available.
- **Transcription error** (5/20): Errors in manually transcribed data.
- **Rounding error** (4/20): Inconsistencies due to wrong rounding.

Table 4: Inconsistencies found between paper and artifact. TI stands for table index. #DE stands for the count of different entries for a table. AD stands for artifact difference. TE stands for transcription error. RE stands for rounding error. AE stands for artifact error. § denotes confirmation by the authors.

Paper	TI	#DE	Paper Value	Artifact Value	Reason
action [15]	1	4	0.30	0.32	AD§
action [15]	2	29	33.8	36.2	AD§
action [15]	3	1	66.3	66.4	RE
action [15]	4	21	-55.14	-62.63	AD§
action [15]	5	5	-10.41	-99.78	AD§
axa [37]	1	2	14	1	AE§[1]
bazel [4]	5	4	28	29	TE§
dypybench [14]	3	1	1701	1543	AE§[2]
interference [20]	2	4	8.13×10^{285960}	8.58×10^{506}	TE§
lasapp [11]	1	1	1549	1553	AD§
llm [33]	3	4	1.15	0.15	TE§
npetest [29]	2	27	100	68	AD§
npetest [29]	3	2	73	74	AD§
pmsat [44]	3	1	0.6	0.5	RE
pmsat [44]	5	1	17.33	17.3	RE
pythonic [52]	3	5	0.14	0.24	TE
pythonic [52]	9	1	0.04	0.05	RE
rust [30]	1	6	98.88%	98.93%	AD§
sctype [50]	3	2	1	2	TE
urcrat [24]	1	1	72199	72201	AD§

[1]Fixed by https://zenodo.org/records/18292903
[2]Fixed by https://github.com/sola-st/DyPyBench/commit/bccfda7

- **Artifact error** (2/20): Errors in the artifact. In one case, the code contained an error, and in the other case, the data contained an error.

We report our findings (excluding rounding errors, as they are obvious but minor issues) to the paper authors, and 14/16 of the reported inconsistencies are confirmed. For the two artifact errors, both have already been fixed by the authors. For the three paper errors (Table 2 of Eder et al. [20], Table 3 of Nam et al. [33], and Table 5 of Alfadel and McIntosh [4]), the authors communicated plans to update the errors. Our approach enables early detection of such reproducibility issues, saving time and avoiding their propagation into future research that builds on the affected papers.

The discovery of paper-artifact inconsistencies is supported by Artisan and the reproduction scripts it generates, but it is not entirely automated. Given the reproduction script, it is ultimately up to a human expert to decide if there is indeed a bug, and if so, where the bug lies.

5 Discussion

5.1 Limitations

While Artisan is effective at reproducing research results, it has several limitations. First, the approach currently only supports textual results in tables. Addressing this limitation would require the development of new judging mechanisms that can evaluate the correctness of non-textual results, e.g., arbitrary figures, which is a non-trivial task. Second, Artisan expects the outputs to exactly match the expected results and thus cannot reproduce non-deterministic results. Thus, Artisan could not be applied to itself, as Artisan makes use of an LLM, which outputs non-deterministic results. Future work could relax this requirement by comparing results with a certain tolerance, e.g., for performance metrics, and by enforcing multiple executions. Finally, the current agent runtime and output judge support Debian-based Docker images only, which could be addressed through additional engineering effort, which we avoided as most artifacts support Linux and Docker.

5.2 Threats to Validity

Our selection criteria for papers and tasks in Artisan-Bench may introduce bias, as we focus on recent, Dockerized artifacts that deterministically produce results without relying on non-public APIs or special hardware. This focus is motivated by the desire to allow us and others to evaluate the capabilities of LLM agents without requiring monetary resources (e.g., for API keys) while minimizing confounding factors such as software evolution and hardware availability. The downside of this focus is that the benchmark may underrepresent the challenges of reproducing results from older, not well-packaged artifacts, artifacts that rely on commercial APIs, and research that relies on very long-running experiments. We plan to research how to expand the benchmark and Artisan to cover such artifacts in the future. Constructing a ground truth script for each task in Artisan-Bench and evaluating the reproduction method of the evaluated approaches involves manual effort, which may introduce bias. Due to the high manual effort, this process is primarily conducted by the first author. However, to mitigate potential bias, a second, senior author has reviewed the ground truth scripts, and all authors have discussed ambiguous cases. Our results are influenced by the step limit (30 steps) and cost limit ($1), which we impose to keep the evaluation cost and time reasonable. With GPT-5.1, step limits are hit by SWE-agent (18/60), OpenHands (22/60), mini-swe-agent (6/60), and Artisan (16/60), suggesting that early stopping in most tasks limits effectiveness more than incomplete exploration.

6 Related Work

Reproducibility of Research Results. Reproducibility of research results is a broad issue in general science [10]. For computer science, various efforts like artifact evaluation processes [22], reproducibility program [36], and empirical evaluation on reproducibility of LLM performance [5] aim to enhance the reproducibility of published research results. Our work contributes to the effort of enhancing reproducibility by automating part of the reproduction process.

Automated Research Reproduction. Recent work leverages LLM agents for reproducibility, defined as enabling a different team to obtain similar experimental outcomes using the same setup [6]. SUPER [12] introduces a benchmark evaluating agents' ability to set up and execute tasks from research repositories. CORE-Bench [40] also evaluates agents on reproducing research results, expanding to more scientific disciplines (e.g., social science, medicine), while REPRO-Bench [25] targets social science papers. Unlike SUPER and CORE-Bench, REPRO-Bench is rating-based: the agent outputs a reproducibility score from 1 to 4. Heye et al. [23] develop an LLM-based system that supports artifact evaluation by outputting

a reproducibility score, setting up environments, and identifying pitfalls. Our approach and benchmark differ from prior work as follows: (1) We formulate the problem as a code generation task, enabling independent assessment of generated reproduction scripts. (2) We provide an automated judging mechanism that evaluates both the output and the method of reproduction.

Automated Research Replication. There is a related but distinct problem of research replication, defined as the ability of a different team to obtain similar experimental outcomes using a different setup [6]. Liang et al. [31] studied LLMs for replicating empirical software engineering research, finding them to be effective at identifying assumptions and boilerplate code. PaperBench [42] evaluates LLM agents on replicating ML papers and reports a top replication score of 21.0%, below the human baseline. PaperCoder [39] is a multi-agent LLM framework for automated research replication via planning, analysis, and generation. Xiang et al. [47] introduce SciReplicate-Bench, a replication benchmark for 36 NLP papers and a multi-agent approach for it. Our work fundamentally differs in the problem addressed (reproduction vs. replication).

General Software Engineering LLM Agents. There is a growing trend of using LLM agents for software engineering tasks [38]. Such agents cover a wide range of tasks, including automated program repair [13], issue solving [48, 51], engineering tasks in machine learning [17], and setting up and running test suites [16]. The task of reproducing research results partially overlaps with software engineering tasks, but has distinct challenges, such as validating the method of reproduction. Our work contributes a novel agent and a novel benchmark for this task.

7 Conclusion

We present Artisan, an automated research reproduction approach leveraging code generation capabilities of an LLM agent guided by an automated judging mechanism. To evaluate Artisan, we develop Artisan-Bench, a new benchmark that evaluates LLM agents' ability in generating reproduction scripts on software engineering papers. Our evaluation shows that Artisan is effective and efficient, having the added benefit of aiding the discovery of paper-artifact inconsistencies.

Data-Availability Statement

The artifact for this paper is available at [8]. Most recent code is available at https://github.com/doehyunbaek/artisan

Acknowledgments

This work was supported by the German Research Foundation (DFG; projects 492507603, 516334526, and 526259073).

References

[1] ASE 2024 2024. *Artifact Evaluation Track — ASE 2024 (Call for Artifacts).* ASE 2024. https://conf.researchr.org/track/ase-2024/ase-2024-artifact-evaluation-track#Call-for-Artifacts

[2] ICSE 2024 2024. *Artifact Evaluation — ICSE 2024.* ICSE 2024. https://conf.researchr.org/track/icse-2024/icse-2024-artifact-evaluation

[3] ESEC/FSE 2024 2024. *Artifacts — FSE 2024 (Call for Artifacts).* ESEC/FSE 2024. https://2024.esec-fse.org/track/fse-2024-artifacts

[4] Mahmoud Alfadel and Shane McIntosh. 2024. The Classics Never Go Out of Style: An Empirical Study of Downgrades from the Bazel Build Technology. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024.* ACM, 233:1–233:12. doi:10.1145/3597503.3639169

[5] Florian Angermeir, Maximilian Amougou, Mark Kreitz, Andreas Bauer, Matthias Linhuber, Davide Fucci, Fabiola Moyón Constante, Daniel Méndez, and Tony Gorschek. 2025. Reflections on the Reproducibility of Commercial LLM Performance in Empirical Software Engineering Studies. *CoRR* abs/2510.25506 (2025). arXiv:2510.25506 doi:10.48550/ARXIV.2510.25506

[6] Association for Computing Machinery. 2020. *Artifact Review and Badging — Current.* Version 1.1. https://www.acm.org/publications/policies/artifact-review-and-badging-current

[7] Manos Athanassoulis, Peter Triantafillou, Raja Appuswamy, Rajesh Bordawekar, Badrish Chandramouli, Xuntao Cheng, Ioana Manolescu, Yannis Papakonstantinou, and Nesime Tatbul. 2022. Artifacts Availability & Reproducibility (VLDB 2021 Round Table). *SIGMOD Rec.* 51, 2 (2022), 74–77. doi:10.1145/3552490.3552511

[8] Doehyun Baek. 2026. Artifact for the ASE 2026 paper: "Automated Table Reproduction via Code Generation". Software. doi:10.6084/m9.figshare.31860751.v3

[9] Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y. Guan, Aleksander Madry, Wojciech Zaremba, Jakub Pachocki, and David Farhi. 2025. Monitoring Reasoning Models for Misbehavior and the Risks of Promoting Obfuscation. *CoRR* abs/2503.11926 (2025). arXiv:2503.11926 doi:10.48550/ARXIV.2503.11926

[10] Monya Baker. 2016. 1,500 scientists lift the lid on reproducibility. *Nature* 533 (2016), 452–454. https://api.semanticscholar.org/CorpusID:4460617

[11] Markus Böck, Michael Schröder, and Jürgen Cito. 2024. Language-Agnostic Static Analysis of Probabilistic Programs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 78–90. doi:10.1145/3691620.3695031

[12] Ben Bogin, Kejuan Yang, Shashank Gupta, Kyle Richardson, Erin Bransom, Peter Clark, Ashish Sabharwal, and Tushar Khot. 2024. super: Evaluating Agents on Setting Up and Executing Tasks from Research Repositories. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, 12622–12645. doi:10.18653/V1/2024.EMNLP-MAIN.702

[13] Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025.* IEEE, 2188–2200. doi:10.1109/ICSE55347.2025.00157

[14] Islem Bouzenia, Bajaj Piyush Krishan, and Michael Pradel. 2024. DyPyBench: A Benchmark of Executable Python Software. *Proc. ACM Softw. Eng.* 1, FSE (2024), 338–358. doi:10.1145/3643742

[15] Islem Bouzenia and Michael Pradel. 2024. Resource Usage and Optimization Opportunities in Workflows of GitHub Actions. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024.* ACM, 25:1–25:12. doi:10.1145/3597503.3623303

[16] Islem Bouzenia and Michael Pradel. 2025. You Name It, I Run It: An LLM Agent to Execute Tests of Arbitrary Projects. *Proc. ACM Softw. Eng.* 2, ISSTA (2025), 1054–1076. doi:10.1145/3728922

[17] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. 2025. MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering. In *The Thirteenth International Conference on Learning Representations.* https://openreview.net/forum?id=6s5uXNWGIh

[18] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/

[19] Georgios-Petros Drosos, Thodoris Sotiropoulos, Diomidis Spinellis, and Dimitris Mitropoulos. 2024. Bloat beneath Python's Scales: A Fine-Grained Inter-Project Dependency Analysis. *Proc. ACM Softw. Eng.* 1, FSE (2024), 2584–2607. doi:10.1145/3660821

[20] Florian Eder and Stefan Winter. 2024. Efficient Detection of Test Interference in C Projects. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 166–178. doi:10.1145/3691620.3694995

[21] Dominik Helm, Tobias Roth, Sven Keidel, Michael Reif, and Mira Mezini. 2024. Unimocg: Modular Call-Graph Algorithms for Consistent Handling of Language Features. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 51–62. doi:10.1145/3650212.3652109

[22] Ben Hermann, Stefan Winter, and Janet Siegmund. 2020. Community expectations for research artifacts and evaluation processes. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu,

Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 469–480. doi:10.1145/3368089.3409767

[23] David Heye, Karl Kindermann, Robin Decker, Johannes Lohmöller, Anastasiia Belova, Sandra Geisler, Klaus Wehrle, and Jan Pennekamp. 2025. Supporting Artifact Evaluation with LLMs: A Study with Published Security Research Papers. In *IEEE International Conference on Big Data, BigData 2025, Macau, China, December 8-11, 2025*. IEEE, 5077–5085. doi:10.1109/BIGDATA66926.2025.11401815

[24] Jaemin Hong and Sukyoung Ryu. 2024. To Tag, or Not to Tag: Translating C's Unions to Rust's Tagged Unions. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 40–52. doi:10.1145/3691620.3694985

[25] Chuxuan Hu, Liyun Zhang, Yeji Lim, Aum Wadhwani, Austin Peters, and Daniel Kang. 2025. REPRO-Bench: Can Agentic AI Systems Assess the Reproducibility of Social Science Research?. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, 23616–23626. https://aclanthology.org/2025.findings-acl.1210/

[26] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=VTF8yNQM66

[27] Shriram Krishnamurthi. 2013. Artifact evaluation for software conferences. *ACM SIGSOFT Softw. Eng. Notes* 38, 3 (2013), 7–10. doi:10.1145/2464526.2464530

[28] Shriram Krishnamurthi and Jan Vitek. 2015. The real software crisis: repeatability as a core value. *Commun. ACM* 58, 3 (2015), 34–36. doi:10.1145/2658987

[29] Myungho Lee, Jiseong Bak, Seokhyeon Moon, Yoonchan Jhi, and Hakjoo Oh. 2024. Effective Unit Test Generation for Java Null Pointer Exceptions. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 1044–1056. doi:10.1145/3691620.3695484

[30] Chenghao Li, Yifei Wu, Wenbo Shen, Zichen Zhao, Rui Chang, Chengwei Liu, Yang Liu, and Kui Ren. 2024. Demystifying Compiler Unstable Feature Usage and Impacts in the Rust Ecosystem. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 24:1–24:13. doi:10.1145/3597503.3623352

[31] Jenny T. Liang, Carmen Badea, Christian Bird, Robert DeLine, Denae Ford, Nicole Forsgren, and Thomas Zimmermann. 2024. Can GPT-4 Replicate Empirical Software Engineering Research? *Proc. ACM Softw. Eng.* 1, FSE (2024), 1330–1353. doi:10.1145/3660767

[32] Monte MacDiarmid, Benjamin Wright, Jonathan Uesato, Joe Benton, Jon Kutasov, Sara Price, Naia Bouscal, Sam Bowman, Trenton Bricken, Alex Cloud, Carson Denison, Johannes Gasteiger, Ryan Greenblatt, Jan Leike, Jack Lindsey, Vlad Mikulik, Ethan Perez, Alex Rodrigues, Drake Thomas, Albert Webson, Daniel Ziegler, and Evan Hubinger. 2025. Natural Emergent Misalignment from Reward Hacking in Production RL. arXiv:2511.18397 [cs.AI] https://arxiv.org/abs/2511.18397

[33] Daye Nam, Andrew Macvean, Vincent J. Hellendoorn, Bogdan Vasilescu, and Brad A. Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 97:1–97:13. doi:10.1145/3597503.3639187

[34] Daniel Olszewski, Allison Lu, Carson Stillman, Kevin Warren, Cole Kitroser, Alejandro Pascual, Divyajyoti Ukirde, Kevin R. B. Butler, and Patrick Traynor. 2023. "Get in Researchers; We're Measuring Reproducibility": A Reproducibility Study of Machine Learning Papers in Tier 1 Security Conferences. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 3433–3459. doi:10.1145/3576915.3623130

[35] Zhiyuan Pan, Xing Hu, Xin Xia, Xian Zhan, David Lo, and Xiaohu Yang. 2024. PPT4J: Patch Presence Test for Java Binaries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 225:1–225:12. doi:10.1145/3597503.3639231

[36] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Hugo Larochelle. 2021. Improving Reproducibility in Machine Learning Research(A Report from the NeurIPS 2019 Reproducibility Program). *J. Mach. Learn. Res.* 22 (2021), 164:1–164:20. https://jmlr.org/papers/v22/20-303.html

[37] Tobias Roth, Julius Näumann, Dominik Helm, Sven Keidel, and Mira Mezini. 2024. AXA: Cross-Language Analysis through Integration of Single-Language Analyses. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 1195–1205. doi:10.1145/3691620.3696193

[38] Abhik Roychoudhury, Corina S. Pasareanu, Michael Pradel, and Baishakhi Ray. 2026. Agentic AI Software Engineers: Programming with Trust. *Commun. ACM* 69, 5 (2026), 56–58. doi:10.1145/3769314

[39] Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. 2025. Paper2Code: Automating Code Generation from Scientific Papers in Machine Learning. *CoRR* abs/2504.17192 (2025). arXiv:2504.17192 doi:10.48550/ARXIV.2504.17192

[40] Zachary S. Siegel, Sayash Kapoor, Nitya Nadgir, Benedikt Stroebl, and Arvind Narayanan. 2024. CORE-Bench: Fostering the Credibility of Published Research Through a Computational Reproducibility Agent Benchmark. *Trans. Mach. Learn. Res.* 2024 (2024). https://openreview.net/forum?id=BsMMc4MEGS

[41] Yahui Song, Xiang Gao, Wenhua Li, Wei-Ngan Chin, and Abhik Roychoudhury. 2024. ProveNFix: Temporal Property-Guided Program Repair. *Proc. ACM Softw. Eng.* 1, FSE (2024), 226–248. doi:10.1145/3643737

[42] Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. 2025. PaperBench: Evaluating AI's Ability to Replicate AI Research. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. OpenReview.net. https://openreview.net/forum?id=xF5PuTLPbn

[43] SWE-agent contributors. 2025. mini-swe-agent. https://github.com/SWE-agent/mini-swe-agent. GitHub repository, MIT License. Version v1.14.4.

[44] Felix Wallner, Bernhard K. Aichernig, and Christian Burghard. 2024. It's Not a Feature, It's a Bug: Fault-Tolerant Model Mining from Noisy Data. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 29:1–29:13. doi:10.1145/3597503.3623346

[45] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, and et al. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net. https://openreview.net/forum?id=OJd3ayDDoF

[46] You Wang, Michael Pradel, and Zhongxin Liu. 2025. Are "Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study. *CoRR* abs/2503.15223 (2025). arXiv:2503.15223 doi:10.48550/ARXIV.2503.15223

[47] Yanzheng Xiang, Hanqi Yan, Shuyin Ouyang, Lin Gui, and Yulan He. 2025. SciReplicate-Bench: Benchmarking LLMs in Agent-driven Algorithmic Reproduction from Research Papers. *CoRR* abs/2504.00255 (2025). arXiv:2504.00255 doi:10.48550/ARXIV.2504.00255

[48] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html

[49] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=WE_vluYUL-X

[50] Brian Zhang. 2024. Towards Finding Accounting Errors in Smart Contracts. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 138:1–138:13. doi:10.1145/3597503.3639128

[51] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1592–1604. doi:10.1145/3650212.3680384

[52] Cyrine Zid, Fiorella Zampetti, Giuliano Antoniol, and Massimiliano Di Penta. 2024. A Study on the Pythonic Functional Constructs' Understandability. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 237:1–237:13. doi:10.1145/3597503.3639211

Received 2026-03-26; accepted 2026-06-18