# CODEDISTILLER: Automatically Generating Code Libraries for Scientific Coding Agents

**Peter A. Jansen**[1,2], **Samiah Hassan**[1], **Pragnya Narasimha**[1]
[1]University of Arizona, [2]Allen Institute for Artificial Intelligence
pajansen@arizona.edu

## Abstract

Automated Scientific Discovery (ASD) systems can help automatically generate and run code-based experiments, but their capabilities are limited by the code they can reliably generate from parametric knowledge alone. As a result, current systems either mutate a small number of manually-crafted experiment examples, or operate solely from parametric knowledge, limiting quality and reach. We introduce CODEDISTILLER, a system that automatically distills large collections of scientific GITHUB repositories into a vetted library of working domain-specific code examples, allowing ASD agents to expand their capabilities without manual effort. Using a combination of automatic and domain-expert evaluation on 250 materials science repositories, we find the best model is capable of producing functional examples for 74% of repositories, while our downstream evaluation shows an ASD agent augmented with a CODEDISTILLER generated library produces more accurate, complete, and scientifically sound experiments than an agent with only general materials-science code examples. We also evaluate LLM-AS-A-JUDGE ratings against domain-expert ratings in an A/B testing paradigm, finding moderate agreement and suggesting that inexpensive proxy metrics may be feasible for evaluating scientific discovery systems at scale.[1]

## 1 Introduction

While automated scientific discovery has been explored for decades (Simon et al., 1981), particularly in the context of literature-based discovery (Swanson, 1986; Yang et al., 2024) and data-driven discovery (Langley and Zytkow, 1989; Majumder et al., 2024), recent advances in code generation models (e.g. Chen et al., 2021; Jiang et al., 2024) have spurred the development of experiment-driven

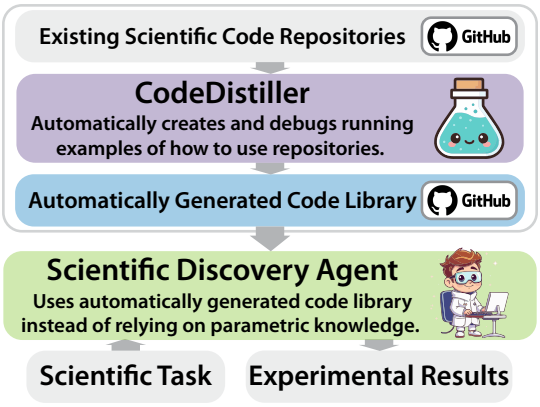


Figure 1: CODEDISTILLER distills a large collection of GITHUB repositories into a library of reusable scientific code, allowing CODE-RAG style scientific discovery agents to perform tasks beyond their parametric knowledge.

discovery agents in domains such as computer science where experiments can be run primarily through code. Typically these agents are supplied with a research task, then must generate (and iteratively debug) code that runs a computational experiment before writing a report describing the results and conclusions, as shown in Figure 1. These systems have recently been demonstrated to produce novel, if incremental, scientific discoveries (Weng et al., 2025; Zhang et al., 2025; Jansen et al., 2025).

A central challenge is that scientific experiments typically require highly specific methods, measurements, and protocols, whereas coding agents are limited to generating experiment code using whatever parametric knowledge they acquired during training. Existing automated scientific discovery agents therefore either mutate pre-existing experiment code (e.g. Lu et al., 2024) or require the experimenter to pre-generate a library of existing vetted code examples they can combine (e.g. Jansen et al., 2025). In this work we address this limitation by building a system that can automatically build a library of functional domain-specific code examples at scale, allowing systems to automatically increase

[1]Video: http://youtu.be/RQxSGFbGZSc, Repository: www.github.com/cognitiveailab/codedistiller



*Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 99–107
July 2-7, 2026 ©2026 Association for Computational Linguistics

their domain-specific capabilities while reducing their reliance on manual code construction.

The contributions of this work are:

1. CODEDISTILLER, a system for automatically converting GITHUB code repositories in specialized scientific domains into debugged, working examples suitable for incorporation in automated scientific discovery systems.
2. An evaluation in the materials science domain, showing that the best and worst performing base models are capable of successfully distilling between 26% to 74% of 250 repositories, with different price/performance tradeoffs.
3. A downstream evaluation showing that an automated discovery system augmented with a library of code automatically built with CODEDISTILLER generates more accurate, complete, and scientifically sound output than a baseline model with only general materials science code examples.
4. A detailed characterization of the agreement (or disparity) between LLM-AS-A-JUDGE vs domain-expert judgments in the materials science domain, showing moderate agreement for evaluating downstream automated scientific discovery quality using A/B testing, but mixed agreement (depending on the model) when evaluating the code distillation task.

## 2 Related Work

**Automated Scientific Discovery:** Agents for automated scientific discovery can be divided into those focusing on literature-based discovery (e.g. Swanson, 1986; Yang et al., 2024), data-driven discovery (e.g. Majumder et al., 2024; Mitchener et al., 2025), and experiment-driven discovery—the latter being the focus of this work. Several agents make use of solely computational experiments for discovery. AI SCIENTIST (Lu et al., 2024) mutates the code of existing experiment repositories to create novel experiments. AGENTLAB (Schmidgall et al., 2025) generates experiment code after reviewing relevant literature. Some approaches rely on expert-generated tool interfaces to domain-specific tools, such as in chemistry (CHEMCROW; Bran et al., 2024) or biology (BIOMNI; Huang et al., 2025). CODEDISTILLER addresses the approach taken by CODESCIENTIST (Jansen et al., 2025), where a CODE-RAG agent retrieves multiple relevant code examples from a vetted code library that it can combine to build complex computational experiments, and reduce reliance on parametric knowledge. While the 6 discoveries made by CODESCIENTIST relied on a code library built through a combination of manual generation and expert curation of LLM-generated examples, CODEDISTILLER allows for augmenting CODE-RAG libraries through automatically generated and vetted examples.

**Tasks:** Several existing tasks are similar to the example distillation task. The SUPER benchmark (Bogin et al., 2024) evaluates agents' capacity to set up and replicate specific research results from a GITHUB repository, with current best performance at 16%. REXBENCH (Edwards et al., 2025) and TM-BENCH (Wölflein et al., 2025) demonstrate automatically repurposing 12 and 15 research repositories (respectively) from PAPERS-WITH-CODE to new research tasks. ENVBENCH (Eliseeva et al., 2025) addresses LLM configuration of development environments. RESEARCHCODEBENCH (Hua et al., 2025) evaluates whether LLMs can implement 212 coding challenges drawn from 20 research articles using only parametric knowledge, finding the best model achieved less than 40% success, supporting the notion that distilling examples from repositories may increase recall. GISTIFY (Lee et al., 2025) requires generating fully self-contained examples from 5 GITHUB repositories whose output passes sets of predefined PYTESTS.

In contrast, CODEDISTILLER focuses on building a code library for CODERAG-style automated discovery systems, which can require integrating code across multiple repositories. CODEDISTILLER does not require repositories to contain unit tests, and is evaluated at greater scale than the above systems—both using automatic LLM-AS-A-JUDGE metrics (Zheng et al., 2023), as well as human domain expert evaluation in our domain of interest (materials science).

## 3 System Overview

The CODEDISTILLER agent workflow is shown in Figure 2. First, large-scale static information gathering steps search and identify code, documentation, and other files relevant to understand the repository and identify highly relevant files. Then, those highly-relevant files are used in a dynamic workflow to generate and iteratively debug a working example of the core repository functionality.

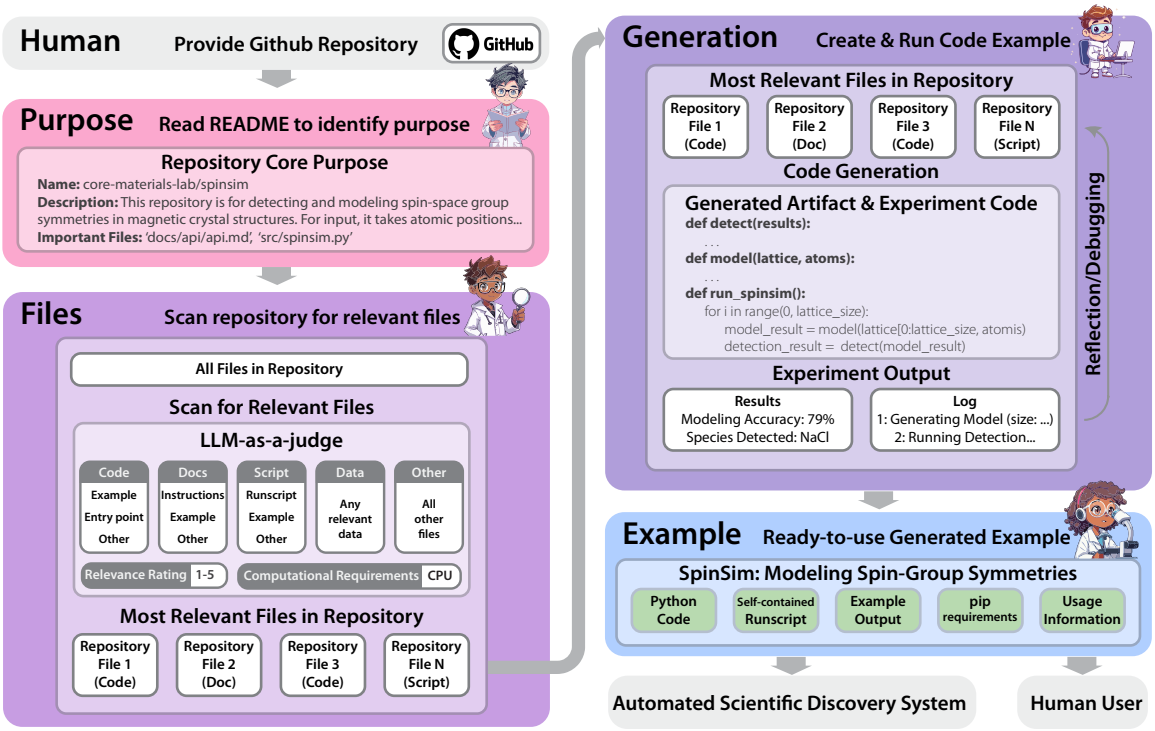


Figure 2: An overview of the core stages of the CODEDISTILLER workflow, including identifying the core purpose of the repository, identifying files relevant for building an example, and the example generation and debugging process.

### 3.1 Relevant File Classification

Repositories can contain hundreds of nested files, but only a small subset of those files – such as top-level APIs, documentation, or existing examples (if available) – tend to be relevant for building example code that demonstrates a repository's core functionality. Similarly, as a pragmatic consideration, the base language model used for generating and debugging the code example has a limited context window, and by providing only the most task-relevant files in the context, we increase the likelihood of success.

To this end, each file in a repository is individually provided to a prompt, which classifies the file contents into several coarse-grained types (shown in Figure 2), such as *code, documentation, scripts, data, or other*. Three of the classes (code, documentation, and scripts) are further subdivided into finer-grained categories (e.g., *existing examples*, *instructions*, *entry points*) to help determine whether a file is likely to have utility. In addition to classification, this scanning procedure also assigns an overall relevance score for each file (on a scale of 1–5) and generates complementary metadata, such as whether the file mentions special computational requirements (e.g., GPUs) or contains critical task-relevant information (e.g., configuration instructions) needed by the code-generation tool.

### 3.2 Code Generation

The code generation step provides the highest-ranked files and the repository's core purpose to a code generation system, whose task is to build, execute, and iteratively debug a working example of the repository's core functionality. First, a prompt (including the most relevant files) is provided to a base LLM, which generates four components: (1) executable PYTHON code, (2) *Python* library requirements, (3) a BASH runscript with CONDA environment setup, and (4) associated metadata, including a description; inclusion/exclusion criteria (i.e., what purposes the example is suited for or inappropriate for); computational resource requirements such as CPU cores, GPUs, RAM, and disk; and whether the repository's code is stand-alone or requires user interaction. This functionality is provided by a modified version of the CODESCIENTIST automated scientific discovery agent, adapted for the example-distillation task, as CODESCIENTIST includes detailed instrumentation for iteratively debugging LLM-generated code.

Once an initial example is generated, the code is automatically run in an UBUNTU cloud container, and its output instrumented and captured. The output includes debugging-oriented instrumentation (such as a timestamped log file recording each major operation, an overall JSON results file, and raw

| Metric | Agent Base Model | | |
|---|---|---|---|
| | GPT-OSS-120B | GPT-5 | Claude Sonnet 4-5 |
| **Overall Performance** | | | |
| *(Auto)* Successfully Completed (LLM-as-a-judge) | 61.6% | 70.4% | 75.6% |
| *(Manual)* Code Executed without Error† | 29.6% | 69.0% | 75.6% |
| *(Manual)* Demonstrates Repository Functionality† | 29.6% | 69.0% | 75.6% |
| *(Manual)* Correct Functionality† | 25.9% | 60.5% | 74.1% |
| **Runtimes** | | | |
| Avg. Runtime (successful cases) | 13.8 mins | 20.3 mins | 19.0 mins |
| Avg. Runtime (unsuccessful cases) | 16.3 mins | 47.0 mins | 21.9 mins |
| Avg. Debug Iterations (successful cases) | 2.4 | 2.2 | 1.9 |
| **Costs** | | | |
| Avg. Cost (successful cases) | $0.09 | $0.70 | $1.71 |
| Avg. Cost (unsuccessful cases) | $0.13 | $1.68 | $2.26 |

Table 1: Summary statistics of CODEDISTILLER performance as a function of using different base models. † Manual analysis was completed by a domain expert on a subsample of 50 repositories that were marked as *successfully completed* by the agent, and normalized to have a common denominator with automatic results.

STDOUT/STDERR logs). The code is also encouraged to generate figures and other human-readable demonstrations useful for verifying functionality. When execution is complete, or exceeds preset runtime thresholds, the output is provided to an LLM-AS-A-JUDGE prompt that determines whether the code functioned correctly or contains issues. If issues are found, the code is reflected – using the current version and execution logs as input – and the modified code is executed. This process repeats until the LLM judges the example to have run successfully, at which point the example generation process ends and the example is provided to the user (or ASD system). To control for costs and runtime, if debugging continues for too many iterations (typically 8), the process is terminated and the example is marked as unsuccessful.

# 4 Evaluation

## 4.1 Materials Science Repositories

We evaluate the performance of CODEDISTILLER in a materials science use case, where GITHUB repositories in the materials science domain are provided, and CODEDISTILLER must distill examples of using those repositories suitable for downstream human or automated scientific discovery agent use.

**Materials Science Libraries:** A materials science subject matter expert assembled a list of 30 popular materials science libraries in PYTHON, including PYMATGEN (a materials analysis library), ASE (tools for atomic simulation), LAMMPS (a molecular dynamics simulator), and PYCALPHAD (a computational thermodynamics modeler).

**Github Repositories:** Using the GITHUB API, we identified all active GITHUB repositories whose source files mentioned at least one of the list of materials science libraries, and whose repository was licensed under a permissive open source license. This resulted in 3,802 unique repositories. To minimize cost, for this evaluation we randomly subsampled this to a list of 250 repositories.

## 4.2 Evaluation Metrics

We evaluate the performance of CODEDISTILLER on three main sets of metrics: automatic assessments of task performance, manual assessments completed by a materials science domain expert, and summary statistics of costs and runtimes.

**Successfully Completed (Auto):** The LLM-as-a-judge (Zheng et al., 2023) that makes a binary assessment of whether CODEDISTILLER has executed correctly and faithfully to the task purpose, and marks the example as complete.

**Code Executed without Error (Domain Expert):** As above, but a domain expert manually inspects the output of the repository to verify that the code executed, and that log files are present.

**Demonstrates Repository Functionality (Domain Expert):** A domain expert examines the final generated code, and makes a binary judgment as to whether the code does demonstrate the core functionality of the input GITHUB repository. For example, if the input repository is for a molecular dynamics simulator, the generated example must perform a molecular dynamics simulation, rather

than simply generating plots, configuration files, or other non-core functionality.

**Correct Functionality (Domain Expert):** A domain expert examines the output of the code, and makes a binary assessment as to whether the output appears correct and faithful to the expected output.

### 4.3 Experiments

**Models:** We evaluated CODEDISTILLER using three popular LLM base models that represent different price/performance points: GPT-OSS-120B, a popular open-weight model, GPT-5, a popular reasoning model that includes web search tool calls, and CLAUDE SONNET 4.5, a popular reasoning model for code generation. The CODEDISTILLER pipeline can use different models for the file classification step (which can make use of inexpensive, fast models) and the code generation and debugging steps (which requires more capable models). As such, the file classification step used the less-expensive version of each model family (e.g. GPT-5-MINI, CLAUDE HAIKU 4.5), with the exception of GPT-OSS-120B, which was both fast and inexpensive enough to use for all parts of the pipeline.

**Domain Expert Evaluation:** A subset of 50 repositories were selected for manual evaluation by the domain expert, based on the criterion that they each were marked as successful by each of the 3 base model's automated *successfully completed* metric. The domain expert has a graduate degree in materials science.

### 4.4 Results

The results are shown in Table 1. Overall, automatic measures of success increase with model cost and complexity, ranging from 62% of repositories being marked as successfully having an example generated by GPT-OSS-120B, up to 76% by CLAUDE SONNET 4.5. This increase in performance comes with a steep increase in cost – with the average CLAUDE-generated example costing 19 times more than the GPT-OSS version.

**Overcounting:** Manual analysis shows a significant mismatch between automatic LLM-AS-A-JUDGE and manually-measured performance, with each model overestimating its performance. This is most stark for GPT-OSS-120B, which reports that it has successfully made examples for 62% of repositories, while the domain expert evaluation suggests the actual proportion with correct functionality is 26%. This undercounting is less for GPT-5, whose manually-validated performance decreases from 70% to 61%, and CLAUDE 4.5, that decreases modestly from 76% to 74%.

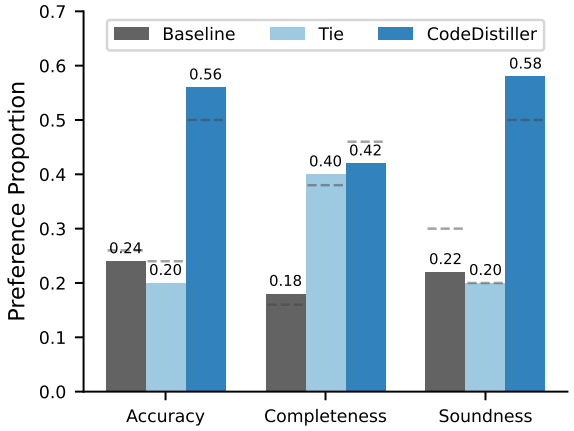


Figure 3: Results of A/B testing, showing the proportion of times the domain expert *(main results)* or LLM-as-a-judge model *(dashed lines)* preferred the experimental output from the baseline model (with generic materials science code examples) versus the model augmented with a CODEDISTILLER-generated library. Values represent the average of 50 experimental tasks implemented using CODESCIENTIST.

**Resource Cost:** Overall, the models share similar runtimes, and similar numbers of debug iterations. In this materials science domain, the example generation process tends to take between 13 and 21 minutes for successful examples, while taking between 1.9 and 2.4 debug iterations, on average. Unsuccessful attempts are a significant resource cost in terms of both LLM API cost and time. This is particularly true in that the model tends to continue to iterate the code until it hits a hard limit on the number of debug iterations, which necessarily costs more than successful examples, which tend to complete after an average of only two to three debug iterations. Here we mitigate this cost by limiting to 8 debug iterations, such as not to incur large cost in the face of diminishing returns as the number of debug iterations increases.

## 5 Downstream Evaluation

Here we evaluate whether incorporating automatically generated code examples from CODEDISTILLER improves the performance of a downstream automated scientific discovery system.

### 5.1 Task Framing

To understand whether incorporating CODEDISTILLER-generated examples into a CODE-RAG agent would improve performance, we constructed a set of materials science tasks centered specifically

| Better Data |
|---|
| **Solution 1 [Baseline]** uses synthetic fabricated data (20 hand-picked molecules replicated) instead of actual Tox21, producing predictions with no scientific validity or real-world grounding.<br>**Solution 2 [CodeDistiller]** uses the actual Tox21 dataset with 6,258 real compounds across 12 toxicity assays, producing scientifically grounded predictions from properly trained models. |
| **Better Modeling** |
| **Solution 1 [Baseline]** uses a generic Lennard-Jones potential not parameterized for Ge, Sb, or Te, resulting in extreme, unphysical volume collapses (80-93%) that do not represent accurate materials behavior.<br>**Solution 2 [CodeDistiller]** uses CHGNet, a materials-specific ML potential trained on DFT data, producing physically reasonable volume changes (-16% to +75%) that reflect realistic structural relaxation behavior. |
| **More Canonical Solutions** |
| **Solution 1 [Baseline]** provides numerical values with proper units for all parameters, but implements calculations from scratch with a manual database, leading to some discrepancies in atomic size difference values (e.g., AlTiVNb δ3b4 = 3.60% vs 5.428% in Solution 2).<br>**Solution 2 [CodeDistiller]** uses established computational materials science libraries (pymatgen and Parameter-Calculator-for-CCA) that are peer-reviewed and widely validated, likely providing more reliable atomic property data and calculation implementations. |

Table 2: Example A/B test preference ratings, highlighting more accurate science from CODEDISTILLER.

around the primary function of a subset of the materials science repositories generated in Section 4. This includes tasks on magnetic symmetry, cheminformatics, condensed matter, and other related tasks. An automated scientific discovery agent ran with and without the code examples, and the output was compared.

**Tasks:** A diverse set of 60 discovery problems targeted at the primary purpose of the code examples were generated using CLAUDE SONNET 4.5. The list of repositories was filtered by the domain expert to a set of 12 with minimal external data requirements, as materials science tasks can often require closed-source data that is difficult to acquire, and 5 problems were generated for each repository.

**Discovery Agent:** The tasks were provided to CODESCIENTIST in two configurations. In the *baseline* configuration, CODESCIENTIST was given access to general materials-science domain code examples, including popular simulation packages for materials analysis, atomic simulation, molecular dynamics, and computational thermodynamics, as well as associated openly accessible

| Rating Dimension | Cohen's $\kappa$ |
|---|---|
| Accuracy | 0.77 |
| Soundness | 0.70 |
| Completeness | 0.62 |

Table 3: Agreement between manual (domain-expert) and LLM-as-a-judge A/B preference ratings across each rating dimension for the downstream discovery task.

data. In the *experimental* configuration, the set of code examples was expanded to include the relevant CODEDISTILLER generated example appropriate for a given task. We used CLAUDE SONNET 4.5 as a base model. Each CODESCIENTIST run was given a maximum of 15 debug iterations, 6 hours of total runtime (capped at 60 minutes per debug iteration), and up to $5 USD of LLM-associated costs. Because automated discovery is still challenging and low-recall, we submitted each of the baseline and experimental tasks twice, and for a given discovery problem, used the first run that successfully completed (if any) in the subsequent evaluation.

**Expert and Automated Evaluation:** To measure whether CODEDISTILLER examples provide benefit over the baseline system using only generic materials-science domain examples combined with parametric knowledge, we evaluated output using an A/B testing paradigm (Kohavi et al., 2020; Fisher, 1935). Specifically, for a given discovery problem, we provided the generated *code*, *raw experimental results*, and *experiment report* from both models, and asked the *domain expert* (manual condition) and an LLM-AS-A-JUDGE (automatic condition) to rate which performed better on three dimensions: *accuracy*, *completeness*, and *soundness*. The output of each model was rated blind, and for LLM-as-a-judge each discovery problem was rated twice, counterbalancing presentation order (Wang et al., 2024; Zheng et al., 2023). The A/B test allowed three ratings: a preference for A, B, or a tie, and was framed in a chain-of-thought paradigm requiring independent textual evaluations of each system before the final A/B rating. To control for problem difficulty, we only examined problems where both *baseline* and *experimental* models produced a solution (50 problems).

### 5.2 Downstream Results

The results of the A/B test are shown in Figure 3. Overall across metrics, in both domain-expert and LLM-as-a-judge conditions, the results show a strong preference for the system that includes

CODEDISTILLER generated examples. While the baseline system is preferred in between 18% to 24% of cases across *accuracy*, *completeness*, and *soundness*, the CODESCIENTIST system augmented with CODEDISTILLER is generally preferred in more than half of cases, while approximately a quarter of problems resulted in ties. Example reasoning traces for these preferences are shown in Table 2, highlighting the utility of automatically including vetted community-generated GITHUB repositories for scientific computation over the parametric knowledge stored within the base model.

**Expert and LLM-as-a-judge Agreement:** Table 3 reports interannotator agreement between manual (domain-expert) and LLM-as-a-judge ratings on the A/B task, measured using Cohen's Kappa (Cohen, 1960), broken down by each of the three dimensions (*accuracy*, *completeness*, and *soundness*). Overall results range between a Kappa of 0.62 and 0.77, which is interpreted as overall *moderate* agreement (McHugh, 2012) – though we note that the *accuracy* dimension approaches *strong* agreement, while the *completeness* dimension borders being classified as *weak* agreement. This suggests that for *accuracy* and *soundness* evaluation dimensions, a strong LLM-as-a-judge model may serve as an approximate inexpensive proxy metric for expert ratings in this paradigm, as expert ratings were expensive to collect, requiring several weeks to examine reports. It also suggests that a domain expert is still particularly important when evaluating the *completeness* of experimental reports.

## 6 Conclusion

We present CODEDISTILLER, a system for automatically converting scientific GITHUB repositories into ready-to-use examples of core domain-specific functionality for code-based scientific discovery agents. Through automated and manual evaluation in the materials science domain, we empirically demonstrate that the best base model can achieve 74% performance on this distillation task, while a system using these automatically constructed code examples outperforms a baseline system in the accuracy, completeness, and scientific soundness of the output. Our system is available as open source software.

## Limitations

**Domain expert:** It is difficult and time consuming to measure whether the generated example is genuinely correct – at the extreme, this would require the domain expert to generate a large set of tests, use the example code on several known problems, and verify the results match the literature. Here, the domain expert examines the code, results, and any generated data/figures, and makes a good-faith effort to judge whether the code appears to be executing correctly and faithfully to the main function of the repository. As such, while we use a domain-expert evaluation, it should be considered a time-restricted proxy evaluation rather than an exhaustive evaluation.

**Automatically identifying materials science repositories:** In this work we automatically identify materials science repositories on GITHUB by searching for repositories that contain at least one file with at least one import from a list of common materials science libraries. We estimate, based on our manual analysis, that approximately half of the repositories collected in this way are directly related to materials science, while the remainder are unrelated, but happened to import (for example) a data analysis library used in materials science. When conducting the manual analyses of CODEDISTILLER performance, we preferentially selected repositories that were directly related to materials science for the domain expert to evaluate.

**Multi-domain evaluation:** This work examines a materials science use case, which is a high-impact domain that makes use of significant computational experimentation. Some of this computational tooling is open source and potentially available for use by the system, while other tooling is closed and, in the experiment framing described here, unavailable. Similarly, tooling in materials science frequently relies on associated materials data, with similar challenges around open versus closed data availability. If transferred to another domain, domain-specific challenges (including, but not limite to issues of software and data availability) may affect overall code distillation performance, as well as performance on downstream discovery tasks.

**Purpose-built versus general-purpose agents:** A contemporary question in agent-based automated scientific discovery is whether purpose-built agents (such as CODEDISTILLER) or general-purpose agents (such as Claude Code, Codex, and related systems) are better for specific tasks. There are many challenges in resolving this question – systems have different input, budgets, models, architectures, ablations, and highly variable output –

making evaluating this in a controlled manner difficult, particularly in the present case where substantial manual evaluation by a domain expert is required. This work is agnostic to this open question of architecture, and (depending on the system or context it is used in), CODEDISTILLER could be used both as a purpose-built agent for pregenerating code libraries as a preprocessing step, but also as a tool or subagent that more general agents could call during their execution.

## Acknowledgments

This research was developed with funding from the Defense Advanced Research Projects Agency's (DARPA) SciFy program (Agreement No. HR00112520300) to PJ at the University of Arizona. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. This work was supported in part a Modal for Academics compute grant. PJ has an outside interest in the Allen Institute for Artificial Intelligence. This interest has been disclosed to the University of Arizona and reviewed in accordance with its conflict of interest policies. We thank the members of the DARPA Scientific Feasibility (SciFy) program and Peter Clark for thoughtful discussions.

## References

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*, pages 12622–12645, Miami, Florida, USA. Association for Computational Linguistics.

M. Bran, A. Cox, O. Schilter, and 1 others. 2024. Augmenting large language models with chemistry tools. *Nature Machine Intelligence*, 6:525–535.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 34 others. 2021. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46.

Nicholas Edwards, Yukyung Lee, Yujun (Audrey) Mao, Yulu Qin, Sebastian Schuster, and Najoung Kim. 2025. Rexbench: Can coding agents autonomously implement ai research extensions? *arXiv preprint*.

Aleksandra Eliseeva, Alexander Kovrigin, Ilia Kholkin, Egor Bogomolov, and Yaroslav Zharov. 2025. Envbench: A benchmark for automated environment setup. In *ICLR 2025 Third Workshop on Deep Learning for Code*.

Ronald A. Fisher. 1935. *The Design of Experiments*. Oliver and Boyd, Edinburgh, UK.

Tianyu Hua, Harper Hua, Violet Xiang, Benjamin Klieger, Sang T. Truong, Weixin Liang, Fan-Yun Sun, and Nick Haber. 2025. Researchcodebench: Benchmarking llms on implementing novel machine learning research code. *ArXiv*, abs/2506.02314.

Kexin Huang, Serena Zhang, Hanchen Wang, Yuanhao Qu, Yingzhou Lu, Yusuf Roohani, Ryan Li, Lin Qiu, Gavin Li, Junze Zhang, Di Yin, Shruti Marwaha, Jennefer N. Carter, Xin Zhou, Matthew Wheeler, Jonathan A. Bernstein, Mengdi Wang, Peng He, Jingtian Zhou, and 4 others. 2025. Biomni: A general-purpose biomedical ai agent. *bioRxiv*.

Peter Jansen, Oyvind Tafjord, Marissa Radensky, Pao Siangliulue, Tom Hope, Bhavana Dalvi Mishra, Bodhisattwa Prasad Majumder, Daniel S Weld, and Peter Clark. 2025. CodeScientist: End-to-end semi-automated scientific discovery with code-based experimentation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 13370–13467, Vienna, Austria. Association for Computational Linguistics.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*.

Ron Kohavi, Diane Tang, and Ya Xu. 2020. *Trustworthy Online Controlled Experiments: A Practical Guide to A/B Testing*. Cambridge University Press, Cambridge, UK.

Pat Langley and Jan M. Zytkow. 1989. Data-driven approaches to empirical discovery. *Artificial Intelligence*, 40(1):283–312.

Hyunji Lee, Minseon Kim, Chinmay Singh, Matheus Pereira, Atharv Sonwane, Isadora White, Elias Stengel-Eskin, Mohit Bansal, Zhengyan Shi, Alessandro Sordoni, Marc-Alexandre Côté, Xingdi Yuan, and Lucas Caccia. 2025. Gistify! codebase-level understanding via runtime execution. *Preprint*, arXiv:2510.26790.

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

Bodhisattwa Prasad Majumder, Harshit Surana, Dhruv Agarwal, Sanchaita Hazra, Ashish Sabharwal, and Peter Clark. 2024. Position: data-driven discovery

with large generative models. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3):276–282.

Ludovico Mitchener, Angela Yiu, Benjamin Chang, Mathieu Bourdenx, Tyler Nadolski, Arvis Sulovari, Eric C. Landsness, Daniel L. Barabasi, Siddharth Narayanan, Nicky Evans, Shriya Reddy, Martha Foiani, Aizad Kamal, Leah P. Shriver, Fang Cao, Asmamaw T. Wassie, Jon M. Laurent, Edwin Melville-Green, Mayk Caldas, and 18 others. 2025. Kosmos: An ai scientist for autonomous discovery. *Preprint*, arXiv:2511.02824.

Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Zicheng Liu, and Emad Barsoum. 2025. Agent laboratory: Using llm agents as research assistants. *ArXiv*, abs/2501.04227.

Herbert A. Simon, Pat Langley, and Gary L. Bradshaw. 1981. Scientific discovery as problem solving. *Synthese*, 47:1–27.

Don R. Swanson. 1986. Fish oil, raynaud's syndrome, and undiscovered public knowledge. *Perspectives in Biology and Medicine*, 30(1):7–18.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Lingpeng Kong, Qi Liu, Tianyu Liu, and Zhifang Sui. 2024. Large language models are not fair evaluators. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9440–9450, Bangkok, Thailand. Association for Computational Linguistics.

Yixuan Weng, Minjun Zhu, Qiujie Xie, Qiyao Sun, Zhen Lin, Sifan Liu, and Yue Zhang. 2025. Deepscientist: Advancing frontier-pushing scientific findings progressively. *ArXiv*, abs/2509.26603.

Georg Wölflein, Dyke Ferber, Daniel Truhn, Ognjen Arandjelovic, and Jakob Nikolas Kather. 2025. LLM agents making agent tools. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26092–26130, Vienna, Austria. Association for Computational Linguistics.

Zonglin Yang, Xinya Du, Junxian Li, Jie Zheng, Soujanya Poria, and Erik Cambria. 2024. Large language models for automated open-domain scientific hypotheses discovery. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13545–13565, Bangkok, Thailand. Association for Computational Linguistics.

Bo Zhang, Shi Feng, Xiangchao Yan, Jiakang Yuan, Zhiyin Yu, Xiaohan He, Songtao Huang, Shaowei Hou, Zheng Nie, Zhilong Wang, Jinyao Liu, Runmin Ma, Tianshuo Peng, Peng Ye, Dongzhan Zhou, Shufei Zhang, Xiaosong Wang, Yilan Zhang, Meng Li, and 5 others. 2025. Novelseek: When agent becomes the scientist - building closed-loop system from hypothesis to verification. *ArXiv*, abs/2505.16938.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.