AutoSDT: Scaling Data-Driven Discovery Tasks Toward Open Co-Scientists

Yifei Li[1,†*], **Hanane Nour Moussa**[1,†*], **Ziru Chen**[1,†], **Shijie Chen**[1,†], **Botao Yu**[1,†], **Mingyi Xue**[6,⋄], **Benjamin Burns**[1,†], **Tzu-Yao Chiu**[3,†], **Vishal Dey**[1,†], **Zitong Lu**[3,†], **Chen Wei**[5,⋄], **Qianheng Zhang**[5,⋄], **Tianyu Zhang**[3,†], **Song Gao**[5,⋄], **Xuhui Huang**[6,⋄], **Xia Ning**[1,2,4,†], **Nesreen K. Ahmed**[∘], **Ali Payani**[∘], **Huan Sun**[1,†]

[1]Department of Computer Science and Engineering [2]College of Pharmacy
[3]Department of Psychology [4]Department of Biomedical Informatics
[5]Department of Geography [6]Department of Chemistry
∘Cisco Research ⋄University of Wisconsin–Madison
†The Ohio State University

Abstract

Despite long-standing efforts in accelerating scientific discovery with AI, building AI co-scientists remains challenging due to limited high-quality data for training and evaluation. To tackle this data scarcity issue, we present AutoSDT, an automatic pipeline that collects high-quality coding tasks in real-world data-driven discovery workflows. AutoSDT leverages the coding capabilities and parametric knowledge of LLMs to search for diverse sources, select ecologically valid tasks, and synthesize accurate task instructions and code solutions. Using our pipeline, we construct AutoSDT-5K, a dataset of 5,404 coding tasks for data-driven discovery that covers four scientific disciplines and 756 unique Python packages. *To the best of our knowledge, AutoSDT-5K is the only automatically collected and the largest open dataset for data-driven scientific discovery.* Expert feedback on a subset of 256 tasks shows the effectiveness of AutoSDT: 93% of the collected tasks are ecologically valid, and 92.2% of the synthesized programs are functionally correct. Trained on AutoSDT-5K, the Qwen2.5-Coder-Instruct LLM series, dubbed AutoSDT-Coder, show substantial improvement on two challenging data-driven discovery benchmarks, ScienceAgentBench and DiscoveryBench. Most notably, AutoSDT-Coder-32B reaches the same level of performance as GPT-4o on ScienceAgentBench with a success rate of 7.8%, doubling the performance of its base model. On DiscoveryBench, it lifts the hypothesis matching score to 8.1, bringing a 17.4% relative improvement and closing the gap between open-weight models and GPT-4o.[1]

1 Introduction

Accelerating scientific research and development has long been an aspirational goal in AI research (Langley et al., 1987; Waltz and Buchanan, 2009). Since the 1960s, there have been continuous efforts in developing AI methods for scientific discovery, e.g., by constructing massive knowledge bases (Buchanan and Feigenbaum, 1978), designing expert rules and heuristics (Langley et al., 1981), and learning representations from large-scale data (Jumper et al., 2021). Nonetheless, they are tailored for very specific tasks with constrained solution spaces. The realization of an AI assistant for open-ended scientific discovery still seems distant.

Recently, large language models (LLMs) bring new light to fulfill this ambition and have piqued significant interest in building "AI co-scientists" (Boiko et al., 2023; Gottweis et al., 2025) that assist in scientific discovery. Specifically, due to their digital nature, most AI co-scientist agents focus on data-driven discovery workflows (Hey et al., 2009; Majumder et al., 2024), including scientific computation and analysis (Tian et al., 2024; Chen et al., 2025), symbolic regression (Shojaee et al., 2025a,b), and hypothesis generation (Majumder et al., 2025; Mitchener et al., 2025). While some agents have shown promising results (Gottweis et al., 2025; Lu et al., 2024), they mostly rely on proprietary LLMs, which impedes their adoption in subjects that require transparency and data privacy, such as social science (Étienne Ollion et al.,

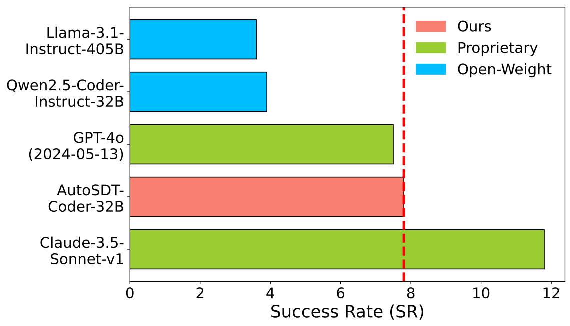


Figure 1: Performance of our AutoSDT-Coder in comparison to other open-weight and proprietary LLMs on ScienceAgentBench. AutoSDT-Coder-32B achieves the same level of performance as GPT-4o (2024-05-13).

*Equal Contribution. See contribution statement for details.

[1]Our code and data are publicly available at https://osu-nlp-group.github.io/AutoSDT/.

2024) and medicine (Zhang et al., 2024). Thus, there is a strong need for AI co-scientists powered by open-weight LLMs.

To build AI co-scientists, one critical bottleneck is the absence of large-scale, high-quality data for training and evaluation. Commonly formulated as code generation problems (Chen et al., 2025; Majumder et al., 2025; Mitchener et al., 2025), data-driven discovery tasks require AI agents to derive scientific insights by processing, analyzing, and visualizing data. On one hand, automatically mining data-driven discovery tasks is particularly challenging. Unlike software engineering tasks (Jimenez et al., 2024), which can often be extracted from code changes in pull requests, *data-driven discovery tasks require complete, file-level code that operates on real-world scientific datasets and solves domain-specific problems*, which cannot be directly crawled from existing code repositories. On the other hand, manual task annotation is quite time-consuming. It takes trained graduate students at least 2.5–3 hours to annotate one task (Chen et al., 2025; Majumder et al., 2025), not to mention extra time for paper searching and task validation.

In this paper, we present AutoSDT, a pipeline for automatically scaling data-driven discovery tasks to tackle the data scarcity issue from three aspects. **(1) Source Diversity**: Our pipeline overcomes the lack of source diversity in manually annotated datasets (Chen et al., 2025; Majumder et al., 2025) by using LLM-based query augmentation to systematically search for code repositories that contain data-driven discovery tasks. **(2) Task Ecological Validity**: We exploit LLMs' parametric knowledge to locate programs that resemble real-world data-driven discovery tasks and generate scientifically accurate task instructions. **(3) Code Quality**: Each selected program goes through multiple rounds of adaptation and validation by an LLM to ensure standalone executability and functional equivalence to the original code.

We use our pipeline to create AutoSDT-5K, a dataset of 5,404 data-driven discovery tasks, which costs only 0.55 USD per task on average. To our best knowledge, AutoSDT-5K is so far the largest coding dataset for data-driven discovery, covering 756 unique Python packages of computational tools in four disciplines: Bioinformatics, Computational Chemistry, Geographical Information Science, and Psychology and Cognitive Neuroscience. We also engage 9 subject matter experts from these disciplines, including Ph.D. students and professors, to examine a subset of 256 tasks. The experts report that **93%** of the tasks are scientifically authentic and represent parts of their data-driven discovery workflows, and **92.2%** of the generated programs are deemed correct solutions to the tasks, validating the high quality of AutoSDT-5K.

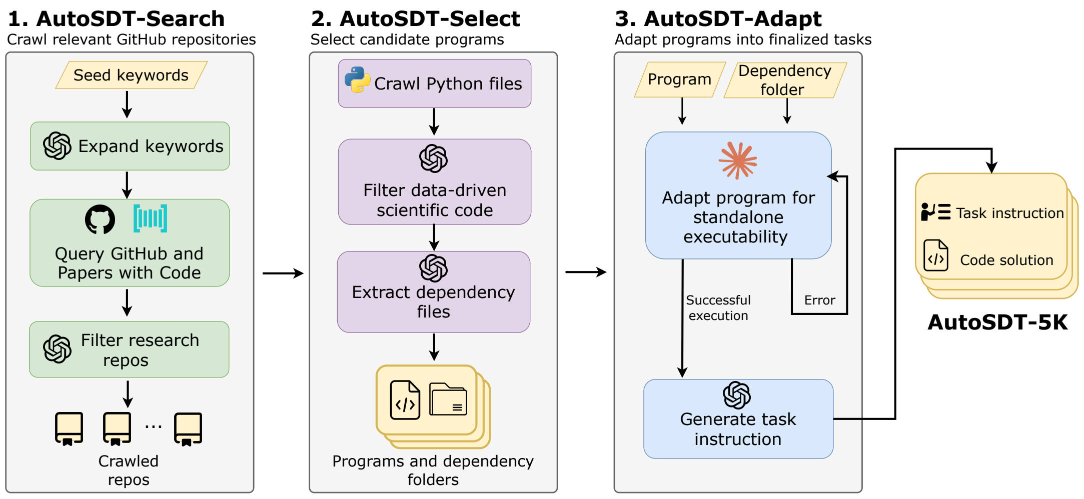
Through comprehensive experiments, we further demonstrate the utility of AutoSDT-5K. We fine-tune Qwen2.5-Coder-Instruct (Hui et al., 2024a) on AutoSDT-5K and obtain AutoSDT-Coder, a series of LLMs with improved coding capabilities for data-driven discovery. We evaluate AutoSDT-Coder on two challenging data-driven discovery benchmarks, ScienceAgentBench (Chen et al., 2025) and DiscoveryBench (Majumder et al., 2025). As shown in Figure 1, AutoSDT-Coder-32B reaches the same level of performance as GPT-4o (2024-05-13) with a 7.8% success rate (SR) on ScienceAgentBench, double the performance of the base model (3.9% SR). On DiscoveryBench, AutoSDT-Coder-32B also brings a 17.4% relative improvement over its base LLM, lifting the hypothesis matching score from 6.9 to 8.1. These results illustrate how AutoSDT can propel the advancement toward open AI co-scientists by automatically scaling high-quality data-driven discovery tasks.

2 AutoSDT

Since manual annotation requires extensive labor and high expertise, existing data-driven discovery datasets (Chen et al., 2025; Majumder et al., 2025) contain only a few hundred tasks for evaluation only. To enable LLM training with a reasonable amount of data, we propose AutoSDT, a fully automatic pipeline for collecting data-driven discovery tasks at scale. As shown in Figure 2, given a few high-level keywords, AutoSDT-Search first searches for related code repositories with keyword expansion (Section 2.1). After that, AutoSDT-Select identifies source code files that correspond to data-driven discovery tasks with multi-step filtering and extracts dependencies for their execution environments (Section 2.2). Lastly, AutoSDT-Adapt modifies the selected source code files into independently executable programs and generates task instructions accordingly (Section 2.3).

2.1 AutoSDT-Search

AutoSDT starts with searching for code repositories containing programs for data-driven discovery tasks. This process is initiated with user-provided



1. AutoSDT-Search
Crawl relevant GitHub repositories
Seed keywords
Expand keywords
Query GitHub and Papers with Code
Filter research repos
Crawled repos
2. AutoSDT-Select
Select candidate programs
Crawl Python files
Filter data-driven scientific code
Extract dependency files
Programs and dependency folders
3. AutoSDT-Adapt
Adapt programs into finalized tasks
Program
Dependency folder
Adapt program for standalone executability
Successful execution
Error
Generate task instruction
Task instruction
Code solution
AutoSDT-5K

Figure 2: AutoSDT collects data-driven discovery tasks in three steps: (1) **AutoSDT-Search** generates a list of keywords for each discipline and searches for relevant repositories. (2) **AutoSDT-Select** identifies programs that represent data-driven discovery tasks and extracts their execution dependency folders. (3) **AutoSDT-Adapt** modifies the selected programs to be independently executable and generates their corresponding task instructions.

keywords describing the topic of interest, such as "bioinformatics," which is the only human effort needed in AutoSDT. These keywords are subsequently expanded into a comprehensive set of related search queries (e.g., "genomics" and "molecular data") by an LLM[2], which significantly improves the coverage of the search results. For example, in our preliminary attempts, we can only find 332 repositories using the keyword "neuroscience" alone. However, after expanding the keyword into a list containing "neuroimaging," "neuroplasticity," and "neuroinformatics," AutoSDT-Search can double the number of its identified repositories in this discipline to 693.

We consider two popular code hosting platforms among researchers, GitHub and PapersWithCode, and use their search APIs to collect a list of repositories. After that, we use an LLM to judge whether each repository indeed hosts code related to a research paper in the targeted discipline according to the `README.md` file (prompt in Appendix Table A.2). *We then eliminate duplicate repositories and ensure that there is no overlap with the repositories utilized in existing benchmarks (Chen et al., 2025; Majumder et al., 2025) which we use for evaluation.* This process yields a large collection of code repositories related to the disciplines of interest, which are further processed by subsequent steps in the pipeline.

[2]The LLM here is GPT-4o (2024-11-20) unless otherwise indicated.

2.2 AutoSDT-Select

AutoSDT processes the crawled repositories with three sub-steps to identify source code files of data-driven discovery tasks and prepare their execution environments (workspaces).

Crawling Python Files. AutoSDT clones the identified repositories and first extracts all Python files. Then, it applies rule-based filtering to remove files exceeding 1,000 lines and directories that are unlikely to contain substantive data-driven discovery tasks (e.g., "config" and "tests").

Data-driven Scientific Code Filtering. We leverage the LLM's parametric knowledge to determine the filtered source code files' relevance to data-driven discovery. Specifically, the LLM assesses if each file meets three criteria: (1) its functionality is related to data-driven scientific workflows, such as model training, computational analysis, and data visualization; (2) it utilizes one or more datasets as program inputs; and (3) it generates scientific outputs such as numerical results, processed datasets, or visualizations. The prompt used for this step is in Appendix Table A.3.

Dependency Extraction and Workspace Preparation. The third sub-step focuses on extracting all necessary dependencies for executing the programs, including datasets, pre-trained models, and auxiliary utility code. This process leverages the LLM's code understanding capabilities to automatically identify required dependencies. The LLM analyzes both file content and repository structure to recognize all dependencies within the repository and returns a list of their paths. We provide the prompt used for dependency extraction in Appendix Table A.4. This step allows us to prepare compact workspaces by only storing necessary files. As a result, the average size of workspaces is only 40.42 MB as opposed to 264.98 MB of repositories.

2.3 AutoSDT-Adapt

Finally, AutoSDT-Adapt creates <task instruction, code solution> pairs by adapting the identified code snippets into independently executable programs and generating corresponding task instructions.

Program Adaptation. Program files taken directly from code repositories often fail to run locally for many reasons such as dependency issues, missing configurations, or other implementation bugs. Therefore, we introduce a code adaptation process to convert raw code files from the repository into standalone and executable code in three sub-steps: (1) We first prompt Claude-3.7-Sonnet (Anthropic, 2025) to generate an initial adaptation of the source code.[3] Given the source code and workspace structure, the LLM is supposed to make modifications to import statements, input/output routines, and hard-coded paths without changing the program's core functionality (Appendix Table A.5). (2) After obtaining the adapted program, we use `pipreqs`[4] to extract Python dependencies and prepare the conda environments for execution. (3) The adapted program is then executed in the configured conda environments for self-debugging (Chen et al., 2024). We repeat the program generation and execution loop for at most three iterations and discard those still having execution errors after the loop finishes.

Task Instruction Generation. Given an adapted program, we prompt an LLM to back-translate it into a clear task instruction (Appendix Table A.6) that explicitly includes the task goal, required input data and/or model files, and expected output files (examples in Appendix Table B.3). According to expert feedback (Section 3.2), most of the generated instructions are correctly expressed in subject-specific scientific language. With programs adapted and instructions generated, we obtain <task instruction, code solution> pairs that represent real-world tasks in data-driven scientific discovery. An example task is provided in Appendix B.

[3]In most stages of AutoSDT, we use GPT-4o (2024-11-20) for its general capabilities and lower cost but adopt Claude-3.7-Sonnet for its high coding performance.

[4]https://github.com/bndr/pipreqs

Statistics	**Value**
# Tasks	5,404
# Repositories	1,325
# Packages	756
Cost (USD)	2,955
Disciplines (# Tasks/ # Repositories):	
Bioinformatics	1,466 / 396
Computational Chemistry	1,345 / 311
Geo. Info. Science	1,541 / 341
Psy. & Cog. Neuroscience	1,052 / 277
Avg # Tasks/Repo	3.8
Avg # Subtasks/Task	4.3
Avg # of lines	262.8

Table 1: Detailed statistics of AutoSDT-5K.

3 AutoSDT-5K

3.1 Statistics

We apply AutoSDT to collect data-driven discovery tasks in four disciplines: Bioinformatics, Chemistry, Geographical Information Science, and Psychology and Cognitive Neuroscience. AutoSDT successfully retrieves 2,993 research-related repositories, selects files from 1,325 of them for further processing, and synthesizes 5,404 coding tasks to compose AutoSDT-5K (Table 1), a large-scale dataset for data-driven discovery.

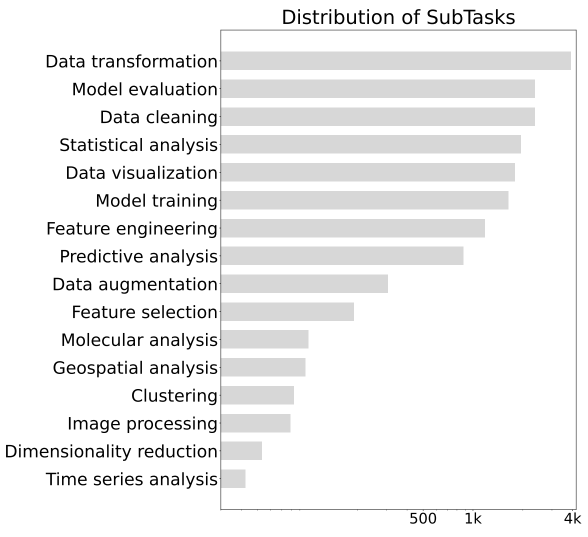
Coverage. AutoSDT-5K covers diverse types of data-driven discovery tasks, where each task comprises multiple subtasks representing specific workflow components (e.g., data transformation, model training, visualization), as illustrated in Figure 3a. In terms of required tools, in addition to common data analysis packages such as `sklearn` and `scipy`, AutoSDT-5K also includes a wide range of domain-specific packages, such as `ase` for atomic simulations, `nibabel` for reading neuroimaging files, and `geopandas` for handling geospatial data. Figure 3b shows representative general and domain-specific packages and their occurrence in AutoSDT-5K.

Cost. In processing all 2,993 repositories and generating 5.4K tasks, our pipeline incurs a total API cost of 2,955 USD, with a cost per task as low as 0.55 USD. The detailed breakdown of the cost is given in Appendix C. For reference, the human effort required to annotate a similar task is 2.5-3 hrs per Chen et al. (2025), which translates to at least 20 USD per task using minimum annotator rates[5].

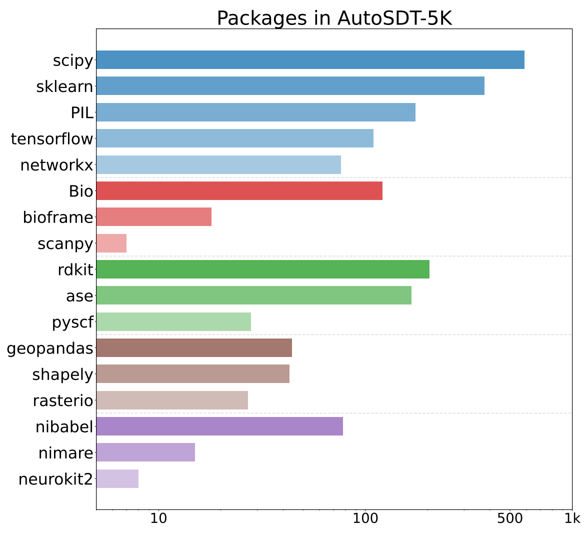
3.2 Expert Evaluation

To confirm the quality of the tasks in AutoSDT-5K, we involve nine subject matter experts to conduct a

[5]https://researcher-help.prolific.com/en/article/9cd998



(a)



(b)

Figure 3: (a) Distribution of subtasks in our dataset. Tasks in AutoSDT-5K are multi-step research workflows covering operations ranging from data preprocessing to more advanced analytics. (b) Examples of packages in AutoSDT-5K, including **general-purpose toolkits** and domain-specific packages for **bioinformatics**, **computational chemistry**, **geographic information science**, and **psychology and cognitive neuroscience**.

Difficulty	**%**	**Avg # of Lines**	**Avg # of Subtasks**
Easy	22.3	214.7	4.1
Medium	48.4	263.7	4.4
Hard	29.3	403.2	5.1

Table 2: Expert-rated task difficulty distribution and average lines of code and number of subtasks per task for each difficulty level.

rigorous evaluation: 3 in bioinformatics and computational chemistry, 3 in geographic information science, and 3 in psychology and cognitive neuroscience. We randomly sample 256 tasks from AutoSDT-5K (96 in bioinformatics/computational chemistry, 75 in geographic information science, and 85 in psychology & cognitive neuroscience). For each task, we prepare a folder containing the task instruction, the link to the original file on GitHub, the program solution, and the program dependencies. The questionnaire used for task evaluation is given in Appendix Table D.3.

Task Instruction Validity. As judged by domain experts, **93%** of the instructions in AutoSDT-5K describe meaningful tasks that scientists would encounter in their day-to-day research activities. In addition, **91.4%** of the task instructions are correctly expressed in the domain scientific language, adding to the ecological validity of the dataset. **73.4%** of the instructions are also clear and contain all the required information to solve the task, namely the task goal, input data, and expected output, showing the effectiveness of our instruction generation pipeline. For the 26.7% of task instructions in which the clarity is lacking, the expert feedback suggests that it is mainly due to the lack of detailed guidance about the methods to be used to solve the task. For example, a task in geographic information science specifies the goal of calculating the Rossby radius of deformation and provide the latitude and longitude of the geographic coordinate, however, in order to perform this calculation other parameters are needed such as buoyancy.

This issue may stem from limited context provided to the LLM for some code files found on GitHub that are often not well documented. A meaningful direction of future exploration would be the incorporation of additional information from the code repository or related publication in the instruction generation stage. However, careful consideration must be given to selecting the context that improves the model's understanding of the task background without overloading it with irrelevant information. We further discuss limitations and future work in the Limitations section.

Code Solution Correctness. The expert evaluation further confirms the effectiveness of AutoSDT in code adaptation. AutoSDT is able to successfully modify the code for standalone executability without altering its original functionality **84.4%** of the time. Based on expert feedback, many of the cases where the codes are not completely equivalent are due to the adapted code locally implementing missing dependencies. However, the correctness of the adapted code is still high, with **92.2%** of programs

Model Size	ScienceAgentBench						DiscoveryBench		
	SR(%, ↑)			VER (%, ↑)			HMS (%, ↑)		
	Base	SFT	Δ	Base	SFT	Δ	Base	SFT	Δ
7B	3.3 (±0.5)	2.3 (±0.9)	-1.0 (30%)	19.9 (±0.2)	27.5 (±3.3)	+7.6 (38%)	4.8 (±1.0)	6.3 (±1.3)	+1.5 (31%)
14B	4.3 (±0.5)	5.9 (±1.6)	+1.6 (37%)	26.5 (±2.1)	35.0 (±2.5)	+8.5 (32%)	6.4 (±0.2)	7.3 (±0.3)	+0.9 (14%)
32B	3.9 (±0.8)	**7.8** (±1.4)	+3.9 (100%)	28.4 (±0.8)	**36.0** (±5.3)	+7.6 (27%)	6.9 (±0.6)	**8.1** (±0.7)	+1.2 (17%)

Table 3: Results of different-sized Qwen2.5-Coder-Instruct models on ScienceAgentBench and DiscoveryBench. In general, fine-tuned models show substantial performance gains compared with base models. All results are generated with zero-shot direct prompting.

deemed correct solutions to their task instructions.
Task Difficulty. In judging the task difficulty, the experts were instructed to estimate how much time it would take them to write a code solution to the task instruction. Similar to Yang et al. (2025), tasks that have a completion time of $\leq$15 min are deemed *Easy*, those requiring 15 min – 1 hr are *Medium*, and tasks requiring 1+ hrs are *Hard*. As illustrated in Table 2, the expert ratings show varied difficulty levels, with more than **75%** of the tasks falling in the *Medium* to *Hard* range. On average, the *Hard* tasks contain considerably more lines of code and more subtasks.

Overall, AutoSDT-5K is a large-scale and high-quality dataset for data-driven discovery tasks, making it a valuable resource for developing future co-scientist agents. Compared to related datasets (Chen et al., 2025; Majumder et al., 2025; Gu et al., 2024; Mitchener et al., 2025), AutoSDT-5K covers a considerably larger set of tasks balanced across multiple disciplines and is the largest open dataset for data driven scientific discovery and the only automatically collected one to the best of our knowledge. *AutoSDT-5K is also the only dataset that is large enough to be used for training purposes, whereas other datasets are used for evaluation only.* AutoSDT-5K is adapted from naturally-occurring scientist-authored code, ensuring that the tasks represent genuine scientific workflows (see Appendix F).

4 Experiments

4.1 Experimental Setup

Datasets. We conduct experiments to show the effectiveness of training on AutoSDT-5K using two data-driven discovery benchmarks: ScienceAgentBench (Chen et al., 2025) and DiscoveryBench (Majumder et al., 2025). In ScienceAgentBench, given a task instruction and dataset information, a method is supposed to generate a complete Python program to solve the task in an end-to-end manner. This requires the generated program to correctly process the input data, implement correct functionality to model, analyze, or visualize the data, and finally save the results into the correct output path. In DiscoveryBench, given the description of input data schema and a scientific query, a method is supposed to first generate Python code to analyze the data based on the query and then generate scientific hypotheses.
Models. We choose Qwen2.5-Coder-Instruct series (Hui et al., 2024b) as our base models for supervised fine-tuning, due to their superior performance on existing coding benchmarks (Yang et al., 2025; Jain et al., 2025; Xie et al., 2025). On ScienceAgentBench, we evaluate: (1) five open-weight LLMs: Llama-3.1-Instruct-70B, 405B (Grattafiori et al., 2024), and Qwen2.5-Coder-Instruct-7B, 14B, and 32B (Hui et al., 2024a). On DiscoveryBench, we compare performance with GPT-4o (2024-11-20) and Qwen2.5-Coder-Instruct. We re-implemented some inference steps to decouple code generation from hypothesis generation, partially leading to different results reproduced in our paper. For all the inferences, we use a temperature of 0.2 and top_p of 0.95, and perform 0-shot direct prompting. More details of training and inference settings can be found in Appendix E.
Evaluation metrics. For ScienceAgentBench, we report (1) Success Rate (SR): a binary metric that examines whether a program output meets the human-annotated success criteria for each task goal and (2) Valid Execution Rate (**VER**): a binary metric which checks if the program can execute without errors and save its output to the correct location. In DiscoveryBench, we evaluate the generated hypotheses against gold hypotheses using Hypothesis Matching Score (**HMS**), which breaks them down into sub-hypotheses with GPT-4o (2024-11-20)[6] to calculate semantic matches. For both datasets, we sample 3 responses and report the average score

[6]The original evaluator LLM, gpt-4-preview-0125, is no longer available since 05/01/2025.

Models	SR (%, ↑)	VER (%, ↑)
Proprietary Reasoning Models		
Claude-3.7-Sonnet	18.6 (±0.8)	51.6 (±4.7)
OpenAI o1-preview	**23.9** (±0.5)	**56.2** (±1.7)
Proprietary Non-Reasoning Models		
GPT-4o (2024-05-13)	7.5 (±0.5)	42.2 (±1.6)
GPT-4o (2024-11-20)	11.4 (±1.2)	43.1 (±2.1)
Claude-3.5-Sonnet-v1	11.8 (±2.1)	36.0 (±1.2)
Open-Weight Models		
Llama-3.1-Instruct-70B	3.6 (±2.0)	22.2 (±0.9)
Llama-3.1-Instruct-405B	3.6 (±0.5)	32.0 (±0.5)
Qwen2.5-Coder-Instruct-32B	3.9 (±0.8)	28.4 (±0.8)
Fine-tuned Open-Weight Models (Ours)		
AutoSDT-Coder-7B	2.3 (±1.2)	27.5 (±3.3)
AutoSDT-Coder-14B	5.9 (±1.6)	35.0 (±2.5)
AutoSDT-Coder-32B	7.8 (±1.4)	36.0 (±5.2)

Table 4: Performance comparison among models on ScienceAgentBench.

(with standard deviation) for all metrics. We put more implementation details in Appendix E.

4.2 Main Results

Training on AutoSDT-5K effectively improves the performance on data-driven discovery tasks. Table 3 demonstrates that models trained on AutoSDT-5K achieve improved SR and VER on ScienceAgentBench. Specifically, we improve Qwen2.5-Coder-32B by 3.9% SR and 7.6% VER. Furthermore, we notice that the performance gains increase with model size; although the 7B model does not show SR improvements, the performance gains become more evident with the 14B and the 32B model. The performance drop for the 7B model is probably due to limited model capacity (Jain et al., 2025) or learnability gap (Li et al., 2025). Consistently, our ablation analysis in Section 4.3 demonstrates that while the 14B model saturates with more training examples, the 32B model is able to further leverage increased data for performance gains.

AutoSDT-Coder models can handle different types of data-driven coding tasks. The performance improvement on DiscoveryBench showcases the generalization capability of models trained on AutoSDT-5K. In addition to improved performance on the tasks in ScienceAgentBench which give explicit instructions about the type of analysis to be conducted, AutoSDT-Coder models perform better at handling the open-ended hypothesis generation questions in DiscoveryBench.

Training Data	Bio.	Chem.	Geo.	Psy & Neu
Bio-only	**18.5**	10.0	0.0	**7.1**
Chem-only	11.1	**15.0**	0.0	**7.1**
Geo-only	14.8	**15.0**	3.7	**7.1**
Psy & Neu	11.1	5.0	3.7	**7.1**
Full	11.1	**15.0**	**14.8**	**7.1**

Table 5: Cross-disciplinary generalization results (SR %) of the 14B model on ScienceAgentBench. Each row indicates the discipline-only training data, while each column reflects performance measured on ScienceAgentBench specific to the target discipline. The SR is computed by counting a case as successful if at least one out of three independent runs is successful.

AutoSDT-Coder-32B outperforms larger open-weight models and rivals proprietary models. As shown in Table 4, we also compare the performance of AutoSDT-Coder models against larger open-weight and proprietary models on ScienceAgentBench. Our models outperform open-weight models of significantly larger size; for example, AutoSDT-Coder-32B achieves more than double the performance of the Llama-3.1-Instruct-405B model. Moreover, AutoSDT-Coder-32B outperforms GPT-4o (2024-05-13) and rivals the performance of Claude-3.5-Sonnet-v1 and GPT-4o (2024-11-20). These results show the effectiveness of AutoSDT in generating high-quality scientific data and its potential to train truly open-weight and open-data co-scientist agents. However, as shown in Table 4, we observe that there is still a large gap with reasoning models[7] like OpenAI-o1 (OpenAI, 2024) and Claude-3.7-Sonnet (Anthropic, 2025). We believe that boosting our data with high-quality reasoning trajectories could be a promising direction to explore and leave it as a future work.

4.3 Ablation Studies

Cross-disciplinary Generalization. Table 5 shows that AutoSDT-Coder-14B achieves decent performance not only on its in-discipline tasks but also generalizes across disciplines to some extent. For example, the Bioinformatics model is able to solve problems requiring specialized tools for cell and molecular analysis (Wolf et al., 2018; Gowers et al., 2016), but is also able to generalize to chemistry tasks due to their shared usage of common scientific libraries and tools. Most disciplines do not benefit from adding training data from other

[7]Models that incorporate a "thinking" or "chain-of-thought" process before generating the final answer.

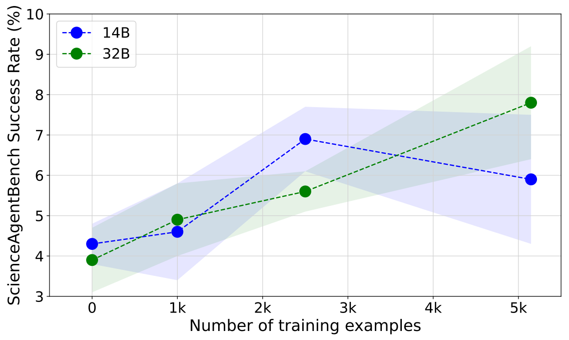


Figure 4: Impact of training set size on ScienceAgentBench performance for AutoSDT-Coder-14B and AutoSDT-Coder-32B. Shaded areas indicate standard deviation across three runs.

domains, thus discipline-specific data collection and training could be a promising direction of future work to build effective specialized models. An exception to this in our experiment is Geographic Information Science, where the specialized model is able to solve domain-specific raster data analysis problems (Rasterio Development Team, 2024), but training on other disciplines allows it to solve a broader set of problems requiring more general tools. Overall, these results suggest that discipline-specific data might be effective in training highly specialized models but multi-discipline training can help one single model tackle a wider range of scientific tasks.

Scaling Training Examples. We analyze the impact of training set size and model size on ScienceAgentBench in Figure 4. Specifically, we fine-tune both Qwen2.5-Coder-14B-Instruct and Qwen2.5-Coder-32B-Instruct using 1k, 2.5k, and 5k training examples. The 14B model exhibits noticeable gains up to 2.5k examples, after which further scaling does not yield improvement, suggesting the onset of performance saturation. In contrast, the 32B model continues to benefit from additional data, achieving higher success rates as the training set increases to 5k+. This analysis suggests that performance gains from scaling up training data become limited for smaller models, while larger models are able to better utilize increased data for further improvement. Such scaling behavior is consistent with previous findings (Jain et al., 2025), which report that the 14B model saturates at approximately 800 training trajectories for addressing software engineering issues, while the 32B still benefits from more training trajectories.

Model	Successful	Avg LoC	Max LoC
Qwen2.5-Coder-Instruct-32B	5	39.0 ± 18.8	62
AutoSDT-Coder-32B	12	39.8 ± 36.3	148

Table 6: Complexity of solved SAB tasks measured by gold program lines-of-code (LoC). AutoSDT solves more and longer cases.

Domain	Data Analy.	Data Proc.	Info Vis.	Model Dev.
Bio.	0.14→0.12	0.52→0.48	0.25→0.18	0.48→0.43
Chem.	0.17→0.17	0.30→0.30	0.14→0.12	0.35→0.26
Geo.	0.31→0.34	0.04→0.04	0.35→0.37	0.04→0.04
Psy & Neu	0.34→0.34	0.09→0.09	0.22→0.20	0.09→0.09
Avg. Δ	**-0.01**	**+0.04**	**+0.09**	**+0.15**

Table 7: Error ratios (base → AutoSDT) on SAB by domain/subtask. Positive Δ indicates reduced error for AutoSDT.

4.4 Additional Analysis

To further understand the behavior of fine-tuned models, we compare successful and failed cases in ScienceAgentBench generated by the base Qwen2.5-Coder-32B-Instruct model and AutoSDT-Coder-32B. We analyzed two aspects: (1) the complexity of successful tasks, and (2) the failed tasks broken down by different disciplines and sub-task categories. We estimate the complexity of tasks by calculating the number of lines in their corresponding gold programs. The case studies are based on a merged set of successful examples from 3 independent runs, where the base 32B model solved 5 cases and AutoSDT-Coder-32B solved 12 cases.

AutoSDT-5K not only improves the overall success rate but also enhances the model's ability to solve more complex tasks. As shown in Table 6, AutoSDT-Coder-32B solves tasks with gold programs of up to 148 lines of code, compared to only 62 for the base model. In contrast, the base model's successful cases are concentrated around shorter code lengths (39.0 + 18.8 = 57.8), falling below the benchmark's average of 58.6. This indicates that the base model tends to succeed only on simpler tasks, while fine-tuning on AutoSDT-5K equips the model with greater capacity to address more complex and diverse data-driven problems.

Across most task categories and domains, AutoSDT-Coder-32B consistently achieves lower or comparable error ratios. As shown in Table 7, in Bioinformatics, it reduces the error rate in Model Development from 0.48 to 0.43, and in Info Visualization from 0.25 to 0.18. Similarly, in Computational Chemistry, it improves performance in Model Development (0.35 $\rightarrow$ 0.26) and Info Visualization (0.14 $\rightarrow$ 0.12). These gains are consistent and suggest that AutoSDT-Coder-32B generalizes better across task types, mostly without sacrificing performance in any domain. This reinforces the earlier observation that SFT helps the model better navigate complex and diverse tasks, not only increasing overall success rates but also reducing failure rates in different domains and task categories.

5 Related Work

Scientific Coding Datasets. Multiple benchmarks have been proposed to measure the growing capabilities of LLMs in scientific domains such as SciCode (Tian et al., 2024), ScienceAgentBench (Chen et al., 2025), DiscoveryBench (Majumder et al., 2025), BLADE (Gu et al., 2024), and BixBench (Mitchener et al., 2025). These works rely on human curation of task instances, which is often inefficient and results in small datasets. In contrast, our work is the first to adopt auto-collection, enabling us to create a dataset at a much larger scale. The closest datasets in scope to AutoSDT-5K are ScienceAgentBench and DiscoveryBench. Both these benchmarks focus on the assessment of agents' abilities to analyze data and write corresponding code solutions and are grounded in well-defined scientific disciplines. We compare AutoSDT to related datasets in Table A.1 in Appendix F.

Automatically Collected Coding Datasets. Our work is the first to address the challenging task of collecting data-driven scientific discovery programs at scale. However, multiple automatic approaches have been introduced for software engineering tasks. RepoST (Xie et al., 2025) presents a sandboxing approach to build scalable training data for function-level coding. Concurrent to our work, R2E-Gym (Jain et al., 2025) proposes synthetic training instances by backtranslating commits, while SWE-smith (Yang et al., 2025) introduces a mostly automatic approach to generate large-scale data for software engineering agents by synthesizing task instances that break the repositories' test cases. Our work significantly differs from these by focusing on code for data-driven scientific discovery. In terms of data collection, finding suitable repositories for our purpose requires added effort. In contrast to works that select their repositories from the most popular PyPI packages (Yang et al., 2025), or use SEART GitHub search [8] with straightforward criteria such as recency, number of stars, etc. (Pan et al., 2025), our data collection process requires initial filtering to source suitable repositories followed by file-level checks to ensure the code is related to data-driven coding tasks. Moreover, our focus on well-defined scientific disciplines requires rigorous human evaluation by domain experts to ensure the relevance of the tasks collected through our pipeline. Such evaluation is often unnecessary for more generic software engineering datasets. Lastly, collecting coding tasks from research repositories introduces a new set of challenges related to verification since such code does not come with associated unit tests to be leveraged. We further discuss the verification challenge in the Limitations section.

6 Conclusion

We introduce AutoSDT, a fully automatic pipeline to collect data-driven scientific coding tasks at scale. Using AutoSDT, we collect AutoSDT-5K and conduct rigorous evaluation with subject matter experts to confirm the quality of its task instances. We train AutoSDT-Coder-32B, which shows substantial performance gains on two recent challenging data-driven scientific discovery benchmarks. Through AutoSDT, we aim to get closer to the ultimate goal of building truly open AI co-scientists.

Limitations

We recognize the following limitations and future work directions:

Verification of task instances. AutoSDT creates data-driven coding datasets composed of task instructions and code solutions but does not generate evaluation scripts for each code solution, thus limiting its usability in some settings such as reinforcement learning. Unlike software engineering datasets that rely on unit tests that are readily available on popular GitHub repositories, the main challenge in creating evaluation scripts for the tasks in AutoSDT-5K is that they should be *outcome-based*, which means that the output of the model should be compared against ground-truth results. However, based on our preliminary attempts, it is non-trivial to ensure the complete correctness of the programs

[8] https://seart-ghs.si.usi.ch/

adapted through AutoSDT without human or even subject matter expert intervention. Relying on alternative methods such as LLM-as-judge or applying heuristics may lead to evaluation scripts that do not measure the true correctness of the program solution and thus be unusable as a signal for reinforcement learning or rejection sampling. An interesting direction of future work would be to implement an automatic and reliable framework to generate instance-specific evaluation scripts which would greatly enhance the use cases of our dataset.

Reasoning Models and Agent Frameworks. In our work, we mainly focus on improving the performance of base models. We do not train reasoning models on AutoSDT-5K due to the challenges in generating effective long chain-of-thought (CoT) rationales at scale. Some recent works have investigated generating CoT rationales from code by explaining solution programs (Li and Mooney, 2024), which presents a promising direction for future research. Likewise, we leave the experimentation with agent frameworks such as OpenHands CodeAct (Wang et al., 2025) and self-debug(Chen et al., 2024) for future work.

Dataset Scale. Due to resource constraints (e.g., API costs and accessibility of human experts), we only crawl 2,993 repositories to generate 5.4k task instances. However, given that there are more repositories on GitHub and PapersWithCode to crawl, future work can use our pipeline to generate even larger datasets.

Discipline and Programming Language Diversity. In creating AutoSDT-5K, we focus on four disciplines that have a wealth of open-source code and experts we can easily contact. However, our pipeline can be used to collect data-driven coding datasets for any discipline that hosts research code on GitHub simply by providing discipline-specific seed keywords. Likewise, our pipeline is geared toward Python since it is the most common programming language in the disciplines of interest. However, AutoSDT can be easily extended to other languages commonly used in data analysis such as R and Stata.

Ethical Considerations

AutoSDT creates tasks based on open-source code and data, and we respect the creators' ownership and intellectual property. We have made our best effort to ensure that the repositories included in AutoSDT-5K have permissive licenses allowing for academic use. We provide more details in Appendix G.

Author Contributions

Y. Li co-led the project, designed and implemented the AutoSDT pipeline, ran AutoSDT to create AutoSDT-5K, implemented and conducted all experiments on ScienceAgentBench and DiscoveryBench, and wrote Section 4 of the manuscript. H. N. Moussa co-led the project, designed and implemented the repository searching in AutoSDT-Search, contributed to testing and optimizing AutoSDT-Select and AutoSDT-Adapt, jointly ran AutoSDT to collect AutoSDT-5K, coordinated the expert evaluation, and wrote Sections 2, 3, 5, 6 of the manuscript. Z. Chen contributed to the core ideas of the project, gave suggestions to optimize experimental designs, and drafted the abstract and section 1 of the manuscript. S. Chen provided feedback on the core ideas and experimental details and revised the manuscript. B. Yu helped run evaluation experiments on ScienceAgentBench. M. Xue, B. Burns, T. Chiu, V. Dey, Z. Lu, C. Wei, Q. Zhang, T. Zhang, S. Gao are subject matter experts who validated 256 randomly sampled tasks. X. Huang and X. Ning are senior subject matter experts in this project. N. Ahmed and A. Payani are senior authors who provided constructive feedback on the project during bi-weekly discussions. H. Sun is the senior lead author who conceived and oversaw the research project, contributed to the core ideas, and revised the manuscript.

Acknowledgements

The authors would like to thank colleagues from the OSU NLP group and the Amazon AGI team for constructive feedback. The authors thank Frazier N. Baker for participating in a preliminary round of expert evaluation. This research was sponsored in part by NSF OAC 2112606, Amazon, Cisco, and Ohio Supercomputer Center (Center, 1987). The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notice herein.

References

Anthropic. 2025. Claude 3.7 sonnet and claude code.

Daniil A. Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. 2023. Autonomous chemical research with large language models. *Nature*, 624:570–578.

Bruce G. Buchanan and Edward A. Feigenbaum. 1978. Dendral and meta-dendral: Their applications dimension. *Artificial Intelligence*, 11(1):5–24. Applications to the Sciences and Medicine.

Ohio Supercomputer Center. 1987. Ohio supercomputer center.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. 2025. Mle-bench: Evaluating machine learning agents on machine learning engineering. *Preprint*, arXiv:2410.07095.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. 2025. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. In *The Thirteenth International Conference on Learning Representations*.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Anil Palepu, Petar Sirkovic, Artiom Myaskovsky, Felix Weissenberger, Keran Rong, Ryutaro Tanno, Khaled Saab, Dan Popovici, Jacob Blum, Fan Zhang, Katherine Chou, Avinatan Hassidim, Burak Gokturk, Amin Vahdat, Pushmeet Kohli, and 15 others. 2025. Towards an ai co-scientist. *Preprint*, arXiv:2502.18864.

Richard J Gowers, Max Linke, Jonathan Barnoud, Tyler JE Reddy, Manuel N Melo, Sean L Seyler, David L Dotson, Jan Domanski, Sébastien Buchoux, Ian M Kenney, and Oliver Beckstein. 2016. Mdanalysis: A python package for the rapid analysis of molecular dynamics simulations. In *Proceedings of the 15th Python in Science Conference*, pages 98–105, Austin, TX. SciPy.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Ken Gu, Ruoxi Shang, Ruien Jiang, Keying Kuang, Richard-John Lin, Donghe Lyu, Yue Mao, Youran Pan, Teng Wu, Jiaqian Yu, Yikun Zhang, Tianmai M. Zhang, Lanyi Zhu, Mike A Merrill, Jeffrey Heer, and Tim Althoff. 2024. BLADE: Benchmarking language model agents for data-driven science. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 13936–13971, Miami, Florida, USA. Association for Computational Linguistics.

Tony Hey, Stewart Tansley, Kristin Tolle, and Jim Gray. 2009. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research.

Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang, Xiao Liu, Jun Zhao, and Kang Liu. 2024. DA-code: Agent data science code generation benchmark for large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 13487–13521, Miami, Florida, USA. Association for Computational Linguistics.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024a. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024b. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *Preprint*, arXiv:2504.07164.

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*.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. 2025. DSBench: How far are data science agents from becoming data science experts? In *The Thirteenth International Conference on Learning Representations*.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, and 15 others. 2021. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E.

Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Pat Langley, Gary L. Bradshaw, and Herbert A. Simon. 1981. Bac0n.5: the discovery of conservation laws. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'81, page 121–126, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Patrick W. Langley, Herbert A. Simon, Gary Bradshaw, and Jan M. Zytkow. 1987. *Scientific Discovery: Computational Explorations of the Creative Process*. The MIT Press.

Jierui Li and Raymond Mooney. 2024. Distilling algorithmic reasoning from llms via explaining solution programs. *Preprint*, arXiv:2404.08148.

Yuetai Li, Xiang Yue, Zhangchen Xu, Fengqing Jiang, Luyao Niu, Bill Yuchen Lin, Bhaskar Ramasubramanian, and Radha Poovendran. 2025. Small models struggle to learn from strong reasoners. *arXiv e-prints*, pages arXiv–2502.

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 *Forty-first International Conference on Machine Learning*.

Bodhisattwa Prasad Majumder, Harshit Surana, Dhruv Agarwal, Bhavana Dalvi Mishra, Abhijeetsingh Meena, Aryan Prakhar, Tirth Vora, Tushar Khot, Ashish Sabharwal, and Peter Clark. 2025. Discoverybench: Towards data-driven discovery with large language models. In *The Thirteenth International Conference on Learning Representations*.

Ludovico Mitchener, Jon M Laurent, Benjamin Tenmann, Siddharth Narayanan, Geemi P Wellawatte, Andrew White, Lorenzo Sani, and Samuel G Rodriques. 2025. Bixbench: a comprehensive benchmark for llm-based agents in computational biology. *Preprint*, arXiv:2503.00096.

OpenAI. 2024. Introducing openai o1-preview.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training software engineering agents and verifiers with SWE-gym. In *ICLR 2025 Third Workshop on Deep Learning for Code*.

Rasterio Development Team. 2024. *Rasterio: Geospatial raster data access for Python*. Rasterio Project.

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K. Reddy. 2025a. LLM-SR: Scientific equation discovery via programming with large language models. In *The Thirteenth International Conference on Learning Representations*.

Parshin Shojaee, Ngoc-Hieu Nguyen, Kazem Meidani, Amir Barati Farimani, Khoa D Doan, and Chandan K Reddy. 2025b. Llm-srbench: A new benchmark for scientific equation discovery with large language models. *Preprint*, arXiv:2504.10415.

Minyang Tian, Luyu Gao, Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, HAO TONG, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Shengzhu Yin, and 10 others. 2024. Scicode: A research coding benchmark curated by scientists. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

David Waltz and Bruce G. Buchanan. 2009. Automating science. *Science*, 324(5923):43–44.

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, and 5 others. 2025. Openhands: An open platform for ai software developers as generalist agents. *Preprint*, arXiv:2407.16741.

Hjalmar Wijk, Tao Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Josh Clymer, Jai Dhyani, Elena Ericheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Sato, William Saunders, and 3 others. 2024. Re-bench: Evaluating frontier ai r&d capabilities of language model agents against human experts. *Preprint*, arXiv:2411.15114.

F. Alexander Wolf, Philipp Angerer, and Fabian J. Theis. 2018. Scanpy: large-scale single-cell gene expression data analysis. *Genome Biology*, 19:15.

Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn Rose. 2024. Codebenchgen: Creating scalable execution-based code generation benchmarks. *Preprint*, arXiv:2404.00566.

Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn Rose. 2025. Repost: Scalable repository-level coding environment construction with sandbox testing. *Preprint*, arXiv:2503.07358.

John Yang, Kilian Leret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. Swe-smith: Scaling data for software engineering agents. *Preprint*, arXiv:2504.21798.

Gongbo Zhang, Qiao Jin, Yiliang Zhou, Song Wang, Betina Idnay, Yiming Luo, Elizabeth Park, Jordan G. Nestor, Matthew E. Spotnitz, Ali Soroush, Thomas R. Campion, Zhiyong Lu, Chunhua Weng, and Yifan Peng. 2024. Closing the gap between open source and commercial large language models for medical evidence summarization. *npj Digital Medicine*, 7(1):239.

Yaowei Zheng, Richong Zhang, Junhao Zhang, YeYanhan YeYanhan, and Zheyan Luo. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 400–410.

Étienne Ollion, Rubing Shen, Ana Macanovic, and Arnault Chatelain. 2024. The dangers of using proprietary llms for research. *Nature Machine Intelligence*, 6(1):4–5.

Appendix

We provide more details omitted from the main text in the Appendix as follows:

- Appendix A: Details and Prompts of AutoSDT
  - Appendix A.1: AutoSDT-Search
  - Appendix A.2: AutoSDT-Select
  - Appendix A.3: AutoSDT-Adapt
- Appendix B: Example Tasks
  - Appendix B.1: Task Instructions
  - Appendix B.2: Full Task Example
- Appendix C: Cost Breakdown
- Appendix D: Expert Evaluation
- Appendix E: Training Details
- Appendix F: Related Datasets
- Appendix G: Repository Licenses

A Details and Prompts of AutoSDT

A.1 AutoSDT-Search

In order to search for suitable repositories we use the GitHub GraphQL API[9] and the PapersWithCode API[10].

After seed keyword expansion using GPT-4o, we obtain the following expanded keywords per discipline: bioinformatics (*genomics, biomarkers, proteomics*), computational chemistry (*molecular dynamics, cheminformatics, catalysis*), psychology (*psychometrics, neuropsychology, cognition*), neuroscience (*neuroimaging, neuroplasticity, neuroinformatics*), geographic information science (*geoscience, geospatial, cartography*).

The seed and expanded keywords are then used to query the GitHub GraphQL API, which searches for repositories containing these keywords within their README.md files or descriptions, alongside terms commonly indicative of research-oriented repositories (e.g., “citation,” “doi,” and “arxiv”). In order to control the quality of retrieved repositories, we restrict results to Python-based repositories with a minimum of 10 stars.

[9]https://docs.github.com/en/graphql
[10]https://paperswithcode.com/api/v1/docs/

Dataset	**Task Instances**	**Subject Domains**	**Scientific Dataset**	**Naturally Occurring Code**	**Auto Collection**
DA-Code (Huang et al., 2024)	500	0	✗	✓	✗
DSBench (Jing et al., 2025)	540	0	✗	✗	✗
MLE Bench (Chan et al., 2025)	75	1	✗	✗	✗
REBench (Wijk et al., 2024)	7	1	✗	✗	✗
ScienceAgentBench (Chen et al., 2025)	102	4	✓	✓	✗
DiscoveryBench (Majumder et al., 2025)	239	6	✓	✓	✗
BLADE (Gu et al., 2024)	12	6	✓	✓	✗
BixBench (Mitchener et al., 2025)	296	1	✓	✗	✗
AutoSDT-5K (Ours)	5404	4	✓	✓	✓

Table A.1: Dataset statistics of AutoSDT-5K compared to related datasets. Columns show the number of instances, number of subject domains, and whether the dataset is oriented towards scientific disciplines, based on naturally-occurring code, and automatically collected.

You are an expert at reading GitHub `README.md` files thoroughly and determining whether the repository hosts code related to a research paper or not, and you are also skilled at correctly extracting the link to the related paper.

Your answer should be based on your thorough understanding of the content of the `README.md` file. Does the `README.md` file indicate that the repository hosts code related to a research paper in the discipline of `{keyword}`? Answer by 'YES' or 'NO' in the 'RESEARCH'. If your answer to the previous question is 'YES', extract the link to the related research paper. Make sure to extract the link to the research paper that this repository implements only, this should be the link to the paper that people would cite if they used the code in the repository for their work, ignoring all other irrelevant links that might be referenced in the `README.md` file. Put the link(s) in front of the 'LINKS': as a list of links.

`README.md` file: `{readme}`

You should strictly follow the format below:

RESEARCH:
LINKS:

Table A.2: Prompt for AutoSDT-Search Repository Filtering Stage.

After a repository is identified via GitHub or PapersWithCode, it goes through an LLM filtering stage using GPT-4o which checks that the repository hosts research code related to the discipline of interest and if so extracts the links to the papers. The prompt used for filtering is given in Table A.2. For PapersWithCode the link extraction stage is skipped since the arXiv paper links can be obtained directly from the API.

A.2 AutoSDT-Select

After identifying suitable repositories using AutoSDT-Search, we locally clone them for further processing. This is because we would quickly reach the GitHub GraphQL rate limit if we perform these operations via API. Once the repositories are cloned and all Python files are identified, we first perform rule-based filtering to eliminate files that are excessively lengthy (i.e., more than 1000 lines) and those located in directories unlikely to contain substantive scientific programs (e.g., "utils," "config," "tests"). The remaining files then undergo LLM-based filtering using GPT-4o to judge whether they host code for data-driven scientific discovery. The prompt containing the detailed criteria is given in Table A.3.

In order to locate dependencies, we use GPT-4o with the prompt given in Table A.4. The LLM is given both the code and the file structure of the repository and is asked to return the paths of the dependencies contained within the repository. These can be dataset files, models, local modules, etc.

A.3 AutoSDT-Adapt

In the program adaptation stage of AutoSDT-Adapt, we prompt Claude-3.7-Sonnet with the original program and the structure of the dependency folder and explicitly instruct it to only make minimal changes required to ensure the executability of the program and not alter the original functionality. The prompt that we use is given in Table A.5. In order to generate instructions, we use GPT-4o with the prompt given in Table A.6.

You are an expert at determining whether a program contains scientific code or not. Given a code file, you need to verify if the current code is a scientific task. Several conditions should be satisfied:

1. Functionality: the functionality of the given program should be related to tasks in a scientific workflow. These tasks include but are not limited to feature engineering, machine learning, deep learning, computational analysis, data visualization, model training, numerical calculation/analysis, statistical methods, domain-specific analysis/simulation, etc.
2. Input: the program should receive at least one or multiple datasets as input. In other words, the program is dealing with a dataset and conducting analysis or experiments on top of the data. The data can either be loaded through built-in functions or be loaded from local files. If the current program does not receive and process any data, it cannot be considered as "a scientific task" here.
3. Output: the program should output numerical or visualization results that can be further evaluated.

A code file is considered a scientific task ONLY IF it completely satisfied the three dimensions above. For example, code files that purely contain modeling, training/testing, data pre-processing, or only consist of utility functions or class definitions, are not considered a scientific task.

Program name: `{file_name}`
Program code: `{code}`

After reasoning about the problem, output your final answer strictly based on the following format:
VERDICT: `{YES/NO}`

Table A.3: Prompt to Verify Data-driven Scientific Discovery Code.

You are an expert software engineer who is very skilled at analyzing Python code files and their repositories to extract dependencies.

In this task you will be given a Python file and the GitHub file tree of the repository it belongs to, your job is to thoroughly understand the code and all the in-repository dependencies it needs. This is because we would like to run this code in a standalone environment and we have to make sure that all the dependencies that the code needs are copied in that environment. Hence, it is very important that you have a thorough understanding of the code and extract all in-repository dependencies needed.

Specifically, your job is to do the following:

1. Recognize whether the code makes use of a dataset. The dataset can either be loaded via built-in library functions (e.g., data = MNIST ()) or loaded from a local file in the repository (csv, jsonl, xls, txt, parquet, or any other file type). If the dataset(s) used in the code are either loaded through built-in library functions or contained within the repository, you should output "Yes" in DATASET_LABEL field. Otherwise, you should output "No".

2. In the case where the dataset used in the code is contained within the repository, you also have to find the relative path to the dataset file, based on the GitHub file tree that will be given to you. You will list the paths to all datasets used in the code as a list of paths after the field DATASET_PATHS.

3. Besides the dataset, now you have to identify all other in-repository dependencies that the code uses, and extract their relative paths based on the file tree given to you. These can be modules, classes, models, or any other dependency that the code imports from a folder within the repository. If you identify that there are in-repository dependencies used, you should put a "Yes" in the MODULE_LABEL. Otherwise, output a "No".

4. In the case of a "Yes", make sure to put the relative paths to all dependencies as a list of paths in the MODULE_PATHS field, based on the GitHub file tree given to you.

5. If based on the code alone you can only identify the folder that contains the dependency but not the exact file only return the path to the folder. This is because you might sometimes not be able to know which file the dependency is exactly located in based on only looking at the file tree. Thus, to stay on the safe side, just give the path to the folder that contains the dependency.

Python code: `{code}`

Project directory: `{directory}`

Table A.4: Prompt to Locate Dependencies.

You are an excellent coder at adapting existing files for standalone executability. You will be given a code file from a Github Repo. Your task is to modify the code into a self-contained program that can be run locally and separately.

Please do not change the original functionality of the code. You must keep the original logic and functionality of the code as much as possible. You should never include dummy/pass statements or empty/mock functions in your response.

You need to slightly modify the source code's input/output logistics and intermediate steps to make it a stand-alone program that can be executed locally. The modified code will then be executed in a local environment. If there are errors, you need to debug the code based on the execution feedback. All the datasets and dependency files are located at `{dataset_path}`. If the original code has imported modules from local files, you can assume they exist and do the same imports in your modified code. Here is the directory structure of the dataset and dependency files: `{dataset_structure}`

Make sure that the code you generate uses the same input files as the original code. Do not generate dummy input files or input data.

For the output of the programs, your code should save the results to a file named "`pred_results/pred_[code_file_name].[extension]`", depending on the type of data such as `csv`, `txt`, `jsonl`, etc. ALL outputs of the program should be saved in the directory `pred_results/`. You should never create new folders or files outside of the specified directory.

Code to be modified:

`{code_file_name}`

`{code}`

The user may execute your code and report any exceptions and error messages. You should address the reported issues and respond with a fixed, complete program. Note that, when addressing bugs, you should ONLY focus on addressing the errors and exceptions and MUST NOT change or delete the main functionality and logic of the original program just to make it executable.

Keep your response concise and do not use a code block if it's not intended to be executed. Do not suggest a few line changes, incomplete program outline, or partial code that requires the user to modify. Your response should include a complete, standalone, executable program.

Do not use any interactive Python commands in your program, such as '!pip install numpy', which will cause execution errors.

Regardless of the iterations of self-debugging, make sure to wrap your program in a code block that specifies the script type, python. For example: "'python print("Hello World!") "'

Table A.5: Prompt for Code Adaptation.

You are a helpful agent for generating task instructions based on a code snippet for solving scientific data processing tasks. You need to provide a clear and concise instruction that best describes the functionality of the given code. The instruction should be written in plain English and should be detailed enough so that a person who has no knowledge of the code can understand the task and implement code for it. The instruction should not reveal too many implementation details but also should be precise and not vague. It should be a high-level description of the code's functionality.

You should thoroughly read the scientific data processing code snippet provided, understand the underlying domain-specific concepts behind it, and generate a task instruction that makes correct use of the domain-specific language. In other words, your task instructions should be written as if they are from a domain scientist giving instructions to a junior researcher in their lab.

The structure of the instruction should be as clear as possible: you should clearly specify the goal of the task, clearly name the exact input file/files that should be used, and the output files that should be created and the path to which they should be saved. Additionally, if the output of the program is written to a file, you should specify the format that the output should be written in, based on the implementation given in the code snippet. In cases where, based on your understanding of the code, you deem that the instruction needs more details - for example, if a certain program can use different computational methods to reach a solution - you can add guidelines about the specific method to use in the instruction. In all cases, ensure that the instruction does not include too many implementation details but also that it is precise and does not invite ambiguity or confusion. The format of your instruction should be a concise paragraph of a few lines without any sections. Keep the instruction focused on the high level scientific goal of the task and do not make reference to unnecessary details like "ensure the directory or so and so files exist". Such low level implementation details should never be part of the instruction.

Please generate the instruction based on the code snippet below.

```
{code}
```

Table A.6: Prompt for Instruction Generation.

Stage	**Cost (USD)**
AutoSDT-Search:	
Repository Crawling	32
AutoSDT-Select:	
Scientific Task Filtering	459
Dependency Locating	828
AutoSDT-Adapt:	
Program Adaptation	1,210
Instruction Generation	426
Total Cost	2,955

Table A.1: AutoSDT Cost Breakdown.

Model Size	**HMS(%)**
Re-implementation Results	
Qwen2.5-Coder-7B-Instruct	4.8
AutoSDT-Coder-7B	6.3
Qwen2.5-Coder-14B-Instruct	6.4
AutoSDT-Coder-14B	7.3
Qwen2.5-Coder-32B-Instruct	6.9
AutoSDT-Coder-32B	8.1
GPT-4o (2024-05-13)	10.4
Results copied from DiscoveryBench	
Llama-3-70B	12.1
GPT-4o (2024-05-13)	15.5

Table A.2: Direct Prompting results on Discovery-Bench.

B Example Tasks

B.1 Task Instructions

We provide examples of task instructions for each of the disciplines covered in AutoSDT-5K in Table B.3.

B.2 Full Task Example

We provide an example of a (task instruction, code solution) pair in Geographic Information in Listing B.1.

C Cost Breakdown

We show the detailed breakdown of the API cost for each stage of AutoSDT in order to build AutoSDT-5K. For AutoSDT-Search, AutoSDT-Select, and the instruction generation in AutoSDT-Adapt, we use GPT-4o. For code adaptation in AutoSDT-Adapt we use Claude-3.7-Sonnet.

D Expert Evaluation

E Training Details

Supervised Fine-tuning. We perform full parameter fine-tuning using the LlamaFactory library (Zheng et al., 2024) . For AutoSDT-Coder-7B/14B/32B, we train them with learning rate 1e-5, maximum 1 epoch, and a max context length of 8192. Warmup is turned off for 7B/14B and turned on for 32B. Training is done on 4 NVIDIA H100

Discipline	Task Instruction
Bioinformatics	*Predict circRNA-disease associations using the Random Walk with Restart (RWR) algorithm. Utilize the circRNA-disease association data in "circrna_disease.txt", along with circRNA and disease lists from "circ_list.csv" and "dis_list.csv" respectively. Perform 5-fold cross-validation to evaluate prediction performance, calculating metrics such as accuracy, recall, precision, F1-score, AUC, and AUPR. Save the results to "RWR.csv" in CSV format, including metrics and their values.*
Computational Chemistry	*Cluster molecular structures based on their chemical fingerprints using the SMILES data in "smiles.csv". Compute Morgan fingerprints for the molecules, perform clustering using the Butina algorithm with a similarity cutoff of 0.72, and identify the centroid molecule for each cluster. Save the clustering summary, including the number of clusters and centroid SMILES, to "clustering.txt" and generate SVG visualizations of the centroid molecules for each cluster and save them as "centroid.svg".*
Geographic Inf. Sci.	*Match geo-tagged drone images to corresponding satellite map images using geographic coordinates. Use the satellite map data from "map.csv" and the drone photo metadata from "metadata.csv". For each drone image, determine its location on the satellite map by comparing its geographic coordinates with the boundaries of the satellite images. Calculate the drone image's precise geographic position within the matched satellite image and compare it to the ground truth coordinates. Save the results, including the calculated coordinates, errors, and matching status, to "results.csv".*
Psy. and Cog. Neuroscience	*Process MRI data to calculate the incidence sizes of parental brain regions. Use the MRI in mri.nii.gz and the Allen Brain annotation file allen.nii.gz. Apply a threshold to identify stroke-affected regions and generate the following outputs: (1) a labeled NIfTI file highlighting affected regions saved as "affected_regions_parental.nii.gz", (2) a text file summarizing stroke volume and affected region percentages saved as "summary.txt", and (3) a MATLAB file with detailed region labels and metrics saved as "label_count.mat".*

Table B.3: Representative examples of task instructions for each discipline.

License	Repositories
MIT	449
GNU	247
Apache	145
BSD	84
CC	57
Boost	4
Public Domain	3
ISC	1
Eclipse	1
PolyForm	1
Mulan	1
Other	15

Table C.1: License information for repositories used in AutoSDT-5K.

Repositories
GabrieleLozupone/AXIAL
fhalab/MLDE
snacktavish/TreeToReads
usnistgov/SDNist
ruppinlab/CSI-Microbes-identification
fenchri/edge-oriented-graph
SNU-LIST/QSMnet
Ramprasad-Group/polygnn
gdalessi/OpenMORe
svalkiers/clusTCR
AI-sandbox/SALAI-Net
pixelite1201/agora_evaluation
jsunn-y/PolymerGasMembraneML
spectrochempy/spectrochempy
usnistgov/atomgpt

Table C.2: Repositories with other licenses.

96G GPUs (for 7B/14B) and 8 for 32B models.

Inference. We use the vLLM library (Kwon et al., 2023) to deploy LLM servers and conduct inference experiments. For all the inference in ScienceAgentBench, we use a default temperature=0.2, top_p=0.95, and max_tokens=2000. For the inference in DiscoveryBench, we use a default temperature=0.2, top_p=0.95, and max_tokens=1024.

Re-implementation of Inference of DiscoveryBench. We re-implemented the inference pipeline based on the original codebase provided by DiscoveryBench authors. Their original implementation was based on LangChain to build an end-to-end LLM agent, while our need is to decouple the code generation step. Therefore, the results of DiscoveryBench in this paper are based on our reproduction and are slightly different from the original paper. We compare our reproduced results and the numbers in the DiscoveryBench paper in Table A.2.

F Related Datasets

We compare AutoSDT-5K against existing science-oriented data analysis datasets in Table A.1. (1) AutoSDT-5K covers a considerably larger set of tasks balanced across multiple disciplines. (2) Unlike MLE-Bench and DSBench which derive code from competition platforms or RE-Bench and Bix-Bench which use human annotators to curate new coding tasks, AutoSDT-5K is based on naturally-occurring code authored by real-world scientists, ensuring the ecological validity of the tasks. (3) AutoSDT-5K is the only automatically generated dataset for coding tasks in scientific disciplines. While automatic generation approaches have been applied to software engineering datasets (Xie et al., 2025, 2024; Yang et al., 2025), this work is the first to address the challenging task of collecting high-quality data-driven scientific discovery programs at scale.

G Repository Licenses

We ensure that all 1325 repositories composing the final tasks in AutoSDT-5K allow for academic use. We list the licenses and the number of corresponding repositories in Table C.1. We manually checked the 15 repositories with custom licenses and ensured that they all allow academic and non-commercial use and list them in Table C.2. There are also 317 repositories without any license information. We assume that these repositories are permissive for academic purposes.

Task Instruction: Generate binary road masks by creating buffers around road geometries defined in GeoJSON files located in “geojson_roads_speed/”. Use the corresponding satellite imagery files from the “PS-RGB” subdirectory to rasterize the buffered road geometries into binary masks. Save the resulting masks as PNG files in the directory specified by “output_mask_path”. Additionally, save a list of unavailable imagery files (those without corresponding GeoJSON labels) to “pred_results/pred_unavailable_files.txt”. Ensure the buffer distance around roads is set to the value of “buffer_meters” (default: 2 meters), and assign a pixel value of “burnValue” (default: 255) for road areas in the masks.

Code Solution:

```python
import argparse
import os
import sys
import time
import numpy as np
from tqdm import tqdm
import cv2
import json
import rasterio
from rasterio import features
from shapely.geometry import shape, mapping
import matplotlib.pyplot as plt

# Create pred_results directory if it doesn't exist
if not os.path.exists('pred_results'):
    os.makedirs('pred_results')

def create_buffer_geopandas(geoJsonFileName, bufferDistanceMeters, bufferRoundness=1,
                            projectToUTM=True, verbose=False):
    """
    Create a buffer around the line segments of the geojson file.
    Return a buffered geometry.
    """

    # Load geojson file
    try:
        with open(geoJsonFileName, 'r') as f:
            geojson_data = json.load(f)
    except Exception as e:
        print(f"create_buffer_geopandas(): can't load GeoJSON file: {geoJsonFileName}, error: {e}
")
        return None

    if not geojson_data.get('features'):
        return None

    # Extract geometries
    geometries = []
    for feature in geojson_data['features']:
        if feature.get('geometry'):
            geom = shape(feature['geometry'])
            geometries.append(geom)

    if not geometries:
        return None

    # Create buffers
    buffered_geometries = []
    for geom in geometries:
        buffered_geom = geom.buffer(bufferDistanceMeters / 111000.0, bufferRoundness)  #
 Approximate conversion from meters to degrees
        buffered_geometries.append(buffered_geom)

    return buffered_geometries
```

Listing B.1: Road mask generation task and code solution (page 1 of 3)

```python
def get_road_buffer(geoJson, im_file, output_raster, buffer_meters=2,
            burnValue=150, bufferRoundness=6,
            plot_file='', figsize=(6, 6), fontsize=8, dpi=500,
            show_plot=False, verbose=False):
"""
Create buffer around roads defined in geoJson file, then burn values to an
output_raster
"""

    buffered_geometries = create_buffer_geopandas(geoJson, buffer_meters,
                                     bufferRoundness=bufferRoundness,
                                     projectToUTM=True, verbose=verbose)

    if not buffered_geometries:
        return None, None
    # Create the mask
    with rasterio.open(im_file) as src:
        src_profile = src.profile
        out_arr = np.zeros((src.height, src.width), dtype=np.uint8)

    # Prepare shapes for rasterization
    shapes = [(mapping(geom), burnValue) for geom in buffered_geometries]
    # Burn the shapes into the raster
    with rasterio.open(im_file) as src:
        out_arr = features.rasterize(shapes=shapes,
                                    out=out_arr,
                                    transform=src.transform)
    # Save the mask if output_raster is provided
    if output_raster:
        # Create output directory if it doesn't exist
        output_dir = os.path.dirname(output_raster)
        if output_dir and not os.path.exists(output_dir):
            os.makedirs(output_dir)

        # Write the mask using OpenCV
        cv2.imwrite(output_raster, out_arr)

    return out_arr, buffered_geometries
def create_masks(path_data, buffer_meters=2, is_SN3=True,
                 burnValue=150, make_plots=True, overwrite_ims=False,
                 output_mask_path='',
                 header=['name', 'im_file', 'im_vis_file', 'mask_file',
                         'mask_vis_file']):
    t0 = time.time()
    # set paths
    path_labels = os.path.join(path_data, 'geojson_roads_speed/')
    # output directories
    path_masks = output_mask_path
    # image directories
    if is_SN3:
        #old directory RGB-PanSharpen-u8
        path_images_vis = os.path.join(path_data, 'PS-RGB')
    else:
        path_images_vis = os.path.join(path_data, 'PS-RGB') \
            if os.path.isdir(os.path.join(path_data, 'PS-RGB')) else os.path.join(path_data, 'PS-
     RGB-u8')

    # Create output directory if it doesn't exist
    if not os.path.exists(path_masks):
        os.makedirs(path_masks)

    outfile_list = []

    # Check if the image directory exists
    if not os.path.exists(path_images_vis):
        print(f"Image directory does not exist: {path_images_vis}")
        return

    im_files = os.listdir(path_images_vis)
    nfiles = len(im_files)
    unavailabel = []

    for i, im_name in enumerate(tqdm(im_files)):
        if not im_name.endswith('.tif'):
            continue

        # define files
        name_root = os.path.basename(im_name)
        im_file_vis = os.path.join(path_images_vis, im_name)

        lab_name = name_root.replace('.tif', '.geojson').split('_')
```

Listing B.2: Road mask generation code solution (continued, page 2 of 3)

```python
        label_file = os.path.join(path_labels,
                                  ''.join(['_'.join(lab_name[0:-1]), '_geojson_roads_speed_', lab_name[-1]]))
        label_file_tot = label_file.replace('PS-RGB_', '')

        if os.path.isfile(label_file_tot):
            mask_file = os.path.join(path_masks, name_root.replace('.tif', '.png'))
            if not os.path.exists(mask_file) or overwrite_ims:
                try:
                    mask, gdf_buffer = get_road_buffer(label_file_tot,
                                                       im_file_vis,
                                                       mask_file,
                                                       buffer_meters=buffer_meters,
                                                       burnValue=burnValue,
                                                       bufferRoundness=6,
                                                       plot_file='',
                                                       figsize=(6, 6),
                                                       fontsize=8,
                                                       dpi=500,
                                                       show_plot=False,
                                                       verbose=False)

                    if mask is not None:
                        cv2.imwrite(mask_file, mask)
            except Exception as e:
                print(f"Error processing {im_name}: {e}")
            else:
                unavailabel.append(im_name)
    print(len(unavailabel), '  Unavilable out of ', nfiles)
    t4 = time.time()
    print("Time to run create_masks():", t4 - t0, "seconds")
# Save list of unavailable files to output
    with open('pred_results/pred_unavailable_files.txt', 'w') as f:
        for item in unavailabel:
            f.write(f"{item}\n")
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--path_data', type=str, required=True,
                        help='Folder containing imagery and geojson labels')
    parser.add_argument('--output_mask_path', required=True, type=str,
                        help='Path to save output masks')
    parser.add_argument('--buffer_meters', default=2, type=float,
                        help='Buffer distance (meters) around graph')
    parser.add_argument('--burnValue', default=255, type=int,
                        help='Value of road pixels (for plotting)')
    parser.add_argument('--overwrite_ims', default=1, type=int,
                        help='Switch to overwrite 8bit images and masks')
    parser.add_argument('--is_SN5', action='store_true', help='SN3 or SN5')

    args = parser.parse_args()
    output_mask_path = args.output_mask_path
    if not os.path.isdir(output_mask_path):
        os.makedirs(output_mask_path, exist_ok=True)
    is_SN5 = args.is_SN5
    print('is_SN5->', is_SN5)
    dir_path = args.path_data
    print('doing...', dir_path)
    create_masks(dir_path,
                 buffer_meters=args.buffer_meters,
                 is_SN3=not is_SN5,
                 burnValue=args.burnValue,
                 output_mask_path=output_mask_path,
                 make_plots=0,
                 overwrite_ims=1)
if __name__ == "__main__":
    # Set up the arguments for testing
    sys.argv = [
        'create_road_masks.py',
        '--path_data', 'benchmark/datasets/SpaceNet8/SpaceNet8/baseline/data',
        '--output_mask_path', 'pred_results/pred_road_masks'
    ]

    # Run the main function
    main()
```

Listing B.3: Road mask generation code solution (continued, page 3 of 3)

AutoSDT-5K Quality Evaluation

Hello domain experts and thank you for collaborating with us on AutoSDT! We have taken your feedback from the pilot study into account and aimed to improve the quality of the tasks generated through our pipeline.

In this round of validation, we aim to rigorously evaluate the quality of the tasks in the training set through three dimensions: the task instruction, code solution, and task difficulty.

You will be given a folder containing three components: the program (.py file), the task instruction and link to original GitHub file (metadata.jsonl), and zip file containing the program dependencies and output in the folder gold_results. After examining all these components please answer the following questions.

Expert Name. [ ]

Task ID. [ ]

Task Instruction.

In this section you will be assessing the quality of the task instruction by determining whether it is meaningful and realistic, correctly expressed in the domain scientific language, and clear.

1. Is this a meaningful and realistic scientific data analysis task that a scientist in your field would perform in their research? **[Yes/No]**

2. Is the instruction correctly expressed in the domain scientific language? **[Yes/No]**

3. Is the instruction clear and contains all required information needed to complete the task - goal, methods, input, and output? In other words, if you were given this instruction as a task, would you have the information you need to start writing a solution? **[Yes/No]**

4. If your answer to the previous question was “No”, what is missing?

Program Solution.

In this section you will answer two questions about the functionality equivalence with the original program on GitHub and the program correctness.

1. Does the program perform the same functionality as the original program on GitHub? There might be changes to the program in the adaptation process to make it executable in a standalone environment (e.g. changes to the import statements, to the input / output routines, etc.) Please ignore all these stylistic changes and focus on whether the core functionality of the program remains unchanged. **[Yes/No]**

2. Does the program represent a valid solution to the task? There could be multiple possible solutions to a task, here you should just determine whether the program is a valid solution and correctly addresses the goal of the task. **[Yes/No]**

Task Difficulty.

In this section you will answer rate the task difficulty.

1. How would you rate the difficulty level of the task? Your judgment about the task difficulty should be realistic (i.e. do not assume familiarity with certain libraries/methods/packages.). In other words, if you had to write a solution for the task right now, how long would that take you? As a general rule of thumb, tasks that can be completed within 15 min are considered easy, those requiring a duration from 15 min to 1 hr are considered medium, and those requiring 1+ hrs are hard. **[Easy/Medium/Hard]**

Table D.3: Questionnaire shared with domain experts for quality evaluation