

# SUPER: Evaluating Agents on Setting Up and Executing Tasks from Research Repositories

Ben Bogin<sup>1,2</sup> Kejuan Yang<sup>2</sup> Shashank Gupta<sup>1</sup> Kyle Richardson<sup>1</sup> Erin Bransom<sup>1</sup>  
Peter Clark<sup>1</sup> Ashish Sabharwal<sup>1</sup> Tushar Khot<sup>1</sup>

<sup>1</sup>Allen Institute for AI <sup>2</sup>University of Washington  
{benb, shashankg, kyler, erinbransom, peterc, ashishs, tushark}@allenai.org

## Abstract

Given that Large Language Models (LLMs) have made significant progress in writing code, can they now be used to autonomously reproduce results from research repositories? Such a capability would be a boon to the research community, helping researchers validate, understand, and extend prior work. To advance towards this goal, we introduce SUPER, the first benchmark designed to evaluate the capability of LLMs in setting up and executing tasks from research repositories. SUPER aims to capture the realistic challenges faced by researchers working with Machine Learning (ML) and Natural Language Processing (NLP) research repositories. Our benchmark comprises three distinct problem sets: 45 *end-to-end problems* with annotated expert solutions, 152 *sub-problems* derived from the expert set that focus on specific challenges (e.g., configuring a trainer), and 604 *automatically generated problems* for larger-scale development. We introduce various evaluation measures to assess both task success and progress, utilizing gold solutions when available or approximations otherwise. We show that state-of-the-art approaches struggle to solve these problems with the best model (GPT-4o) solving only 16.3% of the end-to-end set, and 46.1% of the scenarios. This illustrates the challenge of this task, and suggests that SUPER can serve as a valuable resource for the community to make and measure progress. <sup>1</sup>

## 1 Introduction

Research and scientific discoveries often rely on the reproducibility of experiments conducted by other researchers and the ease with which scientists can build upon each other’s work. In the context of empirical ML and NLP research, it is often crucial for researchers to be able to execute and reproduce diverse research experiments from

<sup>1</sup><https://github.com/allenai/super-benchmark>



Figure 1: An illustration of a research task and some of the steps an agent would need to complete it, including updating data loading configuration, resolving dependency issues (due to unlisted version dependencies in the repository), running the training script and reporting metrics.

open-source repositories, whether to verify existing results or to test them under new conditions.

In practice, even when research code is available, running code from arbitrary repositories is often non-trivial and time-consuming (Samuel and Mietchen, 2022; Storks et al., 2023). Experimentation frequently requires substantial effort to *set up and execute* them: installing the environment, making non-trivial configuration changes, resolving outdated package dependencies, fixing bugs, and determining the correct execution commands, among other tasks. All of this requires a considerable understanding of the documentation and repository code, knowledge about fixing issues (e.g., CUDA errors), as well as the ability to modify the

code appropriately. These steps are especially time-consuming for research repositories “in-the-wild”, as support and documentation may not be available.

In this work, we ask: *Can LLMs automate the set up and execution of tasks in research repositories?* Consider the research task in Fig. 1 where the agent is asked to use a research code repository to train a model with a new optimizer, and evaluate its performance on a custom dataset. A successful agent would need to set up the experiment by installing dependencies, downloading the provided data, and making code changes to load it (first three cells in the figure), then execute the training script while responding to unexpected issues such as an incompatible dependency (fourth and fifth cell), and finally report the result metrics (last cell).

While LLM-based agents have recently been used to produce execution commands from popular research repositories (Liu et al., 2023b), execute popular ML repositories (Huang et al., 2024), or resolve repository issues (Jimenez et al., 2024), no existing benchmark evaluates agents on the common problem faced by many researchers: both *setting up* and *executing* experiments using research repositories *in-the-wild*, i.e., less popular repositories that are not typically well-documented or maintained, which make experiments harder to configure and execute. As a recent study shows (Storks et al., 2023), both novice and advanced researchers find the challenge of “setting up the code base” to be the most difficult part of reproducing experiments.

To encourage research on this problem, we introduce **SUPER** (Setting **UP** and Executing tasks from **R**esearch repositories), a benchmark focusing on lower-profile research repositories. SUPER consists of three distinct problem sets. The *Expert* set contains 45 manually curated problems solved by experts. The *Masked* set includes 152 sub-problems derived from the expert set through our proposed “Code Masking” mechanism, which removes parts of the expert-written code to create diverse sub-problems targeting challenges like dependency installation, data configuration, hyperparameter setup, runtime error resolution, script execution and more. Finally, the *Auto* set contains 604 automatically generated tasks with an even more diverse set of repositories, offering potential use in development, fine-tuning, or training with environment feedback.

To evaluate agents on the Expert and Masked sets, for which we have gold solutions, we compare their answers (e.g., metrics to be reported) to

the gold solutions. To allow for partial credit, we also measure the progress of the agents by checking if they reach specific ‘landmark’ states in their solutions, such as completing a training stage. For the automatically generated problems, for which we have no gold solutions, we simply check if a key script (e.g., the training or evaluation script) was run successfully without exceptions, which we found to be an effective approximate metric.

We evaluate both proprietary and open-source LLMs on SUPER as the underlying models of strong baseline agents with access to file-editing tools. We find that agents struggle to correctly solve many of the problems, with the strongest agent solving only 46.1% of the Masked sub-problems. These agents are even further away from solving entire research tasks, completing correctly only 16.3% of the end-to-end *Expert* tasks. Open-source models substantially lag behind on both the sub-problems and end-to-end tasks. Moreover, we find that the ranking of the agents and models on the Auto set is mostly the same as it is on the curated sets, suggesting its potential usefulness for development.

Our analysis of model trajectories reveals that agents are better at resolving well-specified sub-problems, such as solving exceptions, bugs, and other issues, than tasks requiring repository and file exploration to understand code structure. These results underscore many of the core challenges facing LLM-based experiment execution systems, which our benchmark aims to help advance.

## 2 Related Work

**Coding benchmarks:** While early code benchmarks (Chen et al., 2021; Austin et al., 2021; Cassano et al., 2022) mainly focused on synthesizing simple functions from descriptions, recent benchmarks have shifted to more complex competitive programming problems (Li et al., 2022; Hendrycks et al., 2021; Jain et al., 2024) and evaluating proficiency with popular data science libraries (Lai et al., 2023). Unlike these, we follow the recent trend on evaluating LLMs in more natural programming scenarios, such as programming with external tools and APIs (Li et al., 2023; Shen et al., 2023; Wang et al., 2023b; Patil et al., 2023), code editing and debugging (Cassano et al., 2023; Tian et al., 2024; Haque et al., 2023), resolving GitHub issues (Jimenez et al., 2024) and understanding and coding within a repository context (Liu et al., 2023a; Ding et al., 2024; Zhang et al., 2023).

Resource	SUPER (this work)	DS-1000 (Lai et al., 2023)	ML-Bench (Agent) (Liu et al., 2023b)	MLAgentBench (Huang et al., 2024)	SWE-bench (Jimenez et al., 2024)
Repo. understanding	✓	✗	✓	✗	✓
Requires repository setup	✓	✗	✓	✗	✗
Outcome-based evaluation	✓	✓	✗	✓	✓
Low-profile repositories	✓	✗	✗	-	✗
↔ Median stars	14/14/23	35,309	9,632	-	12,557
# source repositories	45/45/604	8	18	-	12
# problems	45/152/604	1000	9641	15	2294

Table 1: Comparison of SUPER with four related code execution benchmarks across key challenges (rows 1-5) and the number of source repositories and problems in the dataset (rows 6-7). For SUPER, repository and problem counts refer to the Expert/Masked/Auto sets. "Repository understanding" involves navigating repository files to complete tasks, while "Repository setup" refers to installing dependencies and configuring the environment. "Outcome-based evaluation" assesses performance through unit tests or comparing outcome metrics to gold standards. "Low-profile repositories" are those with few GitHub stars.

In contrast to these works, SUPER focuses on the end-to-end task of setting up and executing research tasks in lower-profile repositories, presenting a unique set of challenges, with tasks that require repository comprehension and reasoning, editing multiple files, setting up the repository environment for execution while interactively running commands in the environment. Table 1 compares the four datasets most relevant to SUPER. ML-Bench (Liu et al., 2023b), specifically, its ML-Agent-Bench setup, evaluates LLMs’ ability to execute tasks but focuses on popular code repositories rather than low profile, and does not evaluate based on correctness of outcome, i.e., whether resulting metrics are correct. MLAgentBench (Huang et al., 2024) evaluates agents ability to run ML experiments but focuses on optimizing single-script ML experiments rather than comprehending and setting up arbitrary repositories for experimentation.

**LLM Agents:** Recent advancements in LLM-based agents have shown significant progress across various domains, including games (Wang et al., 2023a), web navigation (Yao et al., 2022; Zhou et al., 2023), human interaction simulation (Park et al., 2023), automating complex computer tasks (Xie et al., 2024), data science and machine learning (Guo et al., 2024; Hong et al., 2024; Liu et al., 2024; Yang et al., 2024b), open-ended discovery (Jansen et al., 2024), and coding (Wang et al., 2024; Yang et al., 2024a; OpenDevin Team, 2024). Our benchmark introduces an important new domain that encourages the development of LLM-based agents to assist researchers in their end to end research tasks with arbitrary repositories.

### 3 Benchmark Construction

In this section we describe the process of building the SUPER benchmark. The SUPER benchmark

consists of 3 sets (see Table 2) serving different purposes. The **Expert** set (§3.1) contains manually written problems, solved by experts. The **Masked** set (§3.2) contains sub-problems extracted from the Expert set using the gold solution, which provide easier and more focused sub-problems. Fig. 2 provides a high-level overview of the construction pipeline of these two sets. Finally, the **Auto** set (§3.3) contains automatically generated problems which can be used for development and improvement of agents.

**Environment Setup.** Running research-oriented repositories often necessitates both being able to run system shell commands (e.g. to install dependencies and run scripts) and stateful Python commands. Previous work and environments typically support only one of these (e.g., only system shell commands (Jimenez et al., 2024) or only Python commands (Huang et al., 2024)). Instead, we build an environment that allows running both of these commands with a Jupyter notebook as engine. In this setup, each execution code is equivalent to running a notebook *cell*, which contains Python code and/or bash commands, and where state is reserved between cell executions (e.g., each cell can use any of the previously defined Python variables). The execution of each cell returns an observation string.

#### 3.1 Expert Set

We construct the Expert set by (1) identifying a set of relevant code repositories from research papers and manually writing research-oriented tasks based on them and (2) asking human experts to provide end-to-end solutions for these tasks (§3.1.1). We then use the expert solutions as the basis for *outcome-based* evaluation, where we compare the agent’s answer to the gold answer, and a more lenient *landmark-based* evaluation that indicates



Figure 2: An overview of the construction pipeline for the Expert and Masked sets. The Expert set contains manually written tasks, along with expert solutions (Step 2). The Masked set contains problems extracted from the experts set (Step 3).

progress toward correctly solving the task, even if the solution is not entirely correct (§3.1.2).

### 3.1.1 Construction

**Tasks.** We create tasks motivated by the following two common settings: (1) reproducing numbers from research papers by running specific experiments, and (2) running modified experiments with different datasets, models, or configurations.

We start by collecting repositories from the “Papers With Code” ([github.com/paperswithcode/paperswithcode-data](https://github.com/paperswithcode/paperswithcode-data)) database, which contains research papers linked to their GitHub repositories, along with some additional metadata such as the modality of the datasets used. We only sample research papers with “Text” modalities and select repositories from 2021 or beyond.

We then manually review the sampled repositories and write tasks that involve running a single experiment that is mentioned either in the repository’s “readme” file or under a script available in the repository, if such can be found. Whenever possible, we make the task more challenging by requiring the experiment to be run on a new dataset or model, other than the one described in the available documentation. In these cases, we select either datasets available on HuggingFace Hub (<https://huggingface.co/datasets>) or provide a Google Drive link where the dataset can be found. The challenge of running on a specific dataset varies in difficulty: it could involve only a single configuration line change if the dataset is already supported, or creating a new dataset reader, adjusting column names, etc.

For each task, we define (1) the target Github repository, (2) the task definition (e.g., “train a model...”), (3) the metrics or output to be reported (e.g., “F1 metric on the validation set”), along with a specific structure of how the answer should be formatted, and (4) implementation instructions (e.g. specific hyper-parameters). The implementation instructions are important for two reasons: first, to allow fair evaluation of submitted answers by making sure the task is not under-specified, such

Set	#	Solutions	Evaluation	Purpose
Expert	45	✓	Solution-based	Benchmark
↔ Masked	152	✓	Solution-based	Benchmark, analysis
Auto	604	✗	Proxy	Development

Table 2: The different sets of SUPER.

that two agents that correctly complete the task get the same results. Second, to minimize computational requirements, as described next.

**Minimizing Computational Requirements.** To make SUPER faster and cheaper to run, we ensure tasks are executable without reliance on GPU machines and that they do not require more than 10 minutes of compute (e.g., for training models or installing packages) on basic compute instances (see §4 for compute details). We therefore create tasks that require minimal compute by only asking to train and evaluate small models (e.g., gpt2-small), and by adding implementation instructions to the task, such as “only load the first 10 examples of the dataset” or “run a single epoch”.

Note that these restrictions do not make the task any easier for the agent. In fact, they often add additional challenges that agents need to solve (e.g., configuring hyper-parameters, finding where data is loaded to limit its loading to the first 10 samples, being able to run experiments that were designed for GPU on a CPU, etc.).

**Expert Annotation.** We use Upwork (<https://www.upwork.com/>) to find and hire experts that have experience with running ML and NLP experiments. We filter the initial list of applications by testing them on a pilot task which we solved ourselves, to make sure they are able to correctly execute a task and effectively solve issues by comparing their solution and results to ours. We instruct workers to execute their solutions on Google Colab, allowing us to collect the solutions in a consistent notebook-like environment.

We ask the experts to submit their (1) solution notebook, (2) answers (i.e. metrics to be reported), the specific git commit hash of the repository that they have used, and the final version list of all dependencies that were installed throughout the notebook.<sup>2</sup> In addition, we instruct them to use default parameters whenever possible, and to report any decision that they had to make but was not specified in the original task (e.g., the selection

<sup>2</sup>The git hash and dependencies ensure that these solutions can be reproduced in the future even as repository and package versions change.

of a delimiter token or specific hyper-parameters when no default values are provided). We add these decisions to the task description or implementation instructions to ensure that any agent solving the same task would have all the necessary information needed to get the same results.

Finally, we manually review their solutions, making sure that (1) the solution correctly follows the task, (2) it can be executed in our environment, (3) all unspecified decisions have been recorded and (4) re-running the experiment multiple times yields the same results (up to an error of  $10^{-2}$ ). If needed, we ask the workers to make corrections, or manually fix issues ourselves. Solutions that we could not execute on our Jupyter environment, such as solutions that had to modify the installed Python version were discarded. We provide cost details and guidelines in Appendix D.

### 3.1.2 Evaluation

**Accuracy Evaluation.** As described in §3.1.1, experts provide us a deterministic solution for each task, which we then execute in our environment to get the gold answer, allowing us to evaluate agents based on their outcome. Answers consist of several values (e.g., numbers for metrics, string for model predictions). We define the accuracy metric as the portion of correctly answered values: where the predicted answer precisely matches the gold one (up to a  $10^{-2}$  error). Unlike reference based evaluation used in cloze tests and various prior coding benchmarks (Liu et al., 2023a,b), outcome-based evaluation allows for alternate valid solutions.

**Landmark-Based Evaluation.** Sometimes an indication of whether the model was precisely correct may be too strict, “punishing” models that make progress but don’t reach the end. E.g., an agent that loads the data but doesn’t train would have the same accuracy as an agent that fails at the start.

To measure progress towards the final goal, we use the gold task notebooks to identify *landmark outputs*; outputs from the environments that act as “evidence” that a particular step was run successfully. E.g., the explicit output string “`*** training completed ***`” in Figure 1 or the string “Loading data... 100%” implying successful data loading.

Importantly, a perfect landmark score does not entail a perfect accuracy score, as landmarks only indicate that some action was performed, but it was not necessarily correct (e.g., a training script

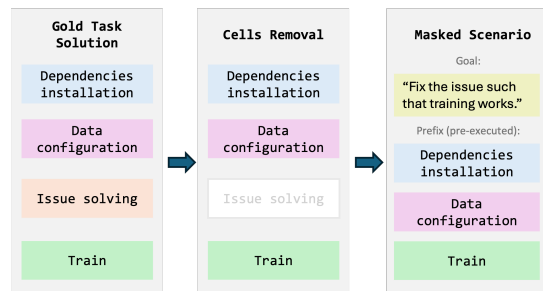


Figure 3: An abstract illustration of sub-problem extraction: starting from the gold end-to-end solution (left), we remove cells focused on specific aspects (middle), then create a masked problem by defining a goal and providing prefix cells (right). The prefix cells are executed, and the agent’s task is to complete the sub-problem.

run successfully but with wrong hyper-parameters could be counted as success). Similarly, albeit unlikely by design, a model could correctly solve the task but not hit all of the landmarks (e.g., if it uses an alternate approach or guesses a solution) and have a lower landmark score. For each gold solution we manually extract 2-6 landmark outputs patterns. The landmarks metric evaluates the percentage of these patterns that appear in the outputs of any of the cells executed by the agent.

### 3.2 Masked Coding sub-problems Extraction

Solving end-to-end execution tasks is often long and complex, involving multiple non-trivial steps. As such, evaluating agents on the entire task provides a sparse signal of success, where agents have to complete numerous steps correctly to succeed, making it harder to “hill-climb” results. Instead, we want to evaluate models in a more fine-grained way that will allow us to get success signals for any incremental progress towards the task. To this end, we propose to focus on a specific sub-problem from the task solution at a time, leveraging the expert solutions from the expert set.

We turn to an approach loosely inspired by cloze tests (Taylor, 1953) and masked language models (MLM; Devlin et al., 2019): given a gold solution for a task we remove (*mask*) some part of it (e.g., code that solves a dependency installation issue), and manually define the sub-problem such that an agent only needs to solve this narrower aspect of the overall task, as captured by the removed cells.

While the ideal goal is for the agents to complete tasks in an end-to-end manner, extracting masked sub-problems allows us to evaluate agents on a broad range of technical challenges often encountered when working with research repositories,

while performing a finer-grained analysis of their performance. In addition, this setup aligns well with the usage of interactive code assistants (e.g., CoPilot and Colab’s AI-powered coding), where agents assist users that have already written or even executed some code, and can specify what issue or problem remains to be solved.

**Masked Coding Sub-Problems.** Each masked sub-problem includes (1) a textual description of the remaining tasks or issues to resolve (e.g., "fix the runtime error to complete training" when cells fixing the error were removed), and (2) a code prefix (e.g., unmasked cells from the original notebook). The code prefix is pre-executed in the environment, and agents must then write and execute their own code to solve the sub-problem.

**Extraction Procedure.** We manually extract masked coding sub-problems by first identifying specific cells (not necessarily consecutive) in the gold solution that focus on a particular aspect. For example, cells related to loading and modifying the dataset can be grouped into a *data configuration* block (shown in pink in Fig. 3). To create sub-problems, we first mask a block (e.g., the orange *Issue solving* block). Next, we identify cells that don’t depend on the masked block and define a remaining goal (such as making the training run given the issue). These cells are pre-executed, and the agent’s task is to complete the remaining goal.

We use the masked block and goal cell to define the sub-problem, e.g., if the code to handle execution on CPU has been masked, the sub-problem definition would be “*I have already executed some of the required steps. Now, you should make the necessary changes to make sure the code runs on a CPU. Your goal is to successfully run ‘train.py’.*”. We choose cells with clearly identifiable success indicators as goals, e.g., successfully running ‘train.py’ would produce metrics on completion.

**Evaluation.** Since each sub-problem has a clearly defined goal, extracted from the original task, we can use the same outputs and landmarks as in the expert set, and similarly evaluate accuracy and landmarks (§3.1.2). We evaluate sub-problems with the same metrics defined in §3.1.2.

### 3.3 Automatically Generated Tasks

The Expert and Masked sets provide validated problems and reproducible solutions, allowing for more accurate evaluation of agents. However, creating

expert tasks is both time-consuming and costly, and the limited number of tasks hinders their use for agent improvements, such as fine-tuning models based on trajectories and environment feedback (e.g., Chen et al., 2023; Song et al., 2024; Yin et al., 2024). To address this, we automatically generate tasks using an LLM (namely, GPT-4o).

#### 3.3.1 Construction

Generation involves two steps: (1) filtering suitable repositories, and (2) generating tasks based on the readmes of these repositories.

The first step filters repositories from "paper-with-code" (§3.1.1), aiming to retain those more likely suitable for task generation. Many randomly sampled repositories are unsuitable due to issues like not supporting the running of any experiments, requiring GPU hardware, or dependencies on external LLM APIs (e.g., OpenAI or Anthropic). To avoid such cases, we apply a combination of heuristic and LLM-based filtering. Details on the filtering process and LLM prompts are provided in Appendix B.

For the filtered repositories, we prompt the LLM with each repository’s README and instruct it to generate an experiment-running task. This includes specifying the goal, dataset, model, and script (e.g., “Run probability-based prompt selection on the SST-2 dataset using opt-125m with run\_prompt\_selection.py”). Further details on this process are provided in Appendix B. To assess quality, we sampled 100 tasks, finding 81% to be feasible. The main issues among the remainder were missing resources (e.g., dead links, missing code) or conceptually flawed tasks, such as a task asking to use a discriminative model for a generative task.

**Difference from Expert Set.** The Auto tasks differ from those in the Expert set in several key ways: Expert tasks may require training or inference on datasets not explicitly supported, sometimes with different formats and columns, whereas Auto tasks prioritize installing dependencies and starting experiments. Additionally, the Expert set includes only tasks that were successfully run in our Jupyter environment, while Auto tasks may involve more complex setups, such as requiring changes to the Python version. Finally, Auto tasks sometimes require navigating web pages mentioned in the repository’s README to download datasets.

Category (%)	Portion	Gold LOC	Example(s) description of a gold solution
Dependencies	19.7%	4.1	Downgrade ‘transformers’ version to allow execution of an older repository
CPU	7.2%	5.1	Remove ‘.cuda()’ from different locations in code
Configuration	12.5%	8.2	Edit Python or shell scripts to set hyper-parameters and experiment details
Data	23.7%	22.7	Download custom dataset, update data loader, limit to loading first 10 samples
Issue	9.2%	5.8	Pytorch incompatible tensor shapes; incorrectly loaded Python package
Goal	25.0%	6.5	Run the evaluation script then load generated file to report metrics
Other	2.6%	3.8	Save the model after training to allow evaluation loading

Table 3: Distribution of sub problems categories and description of representative solutions from experts. LOC stands for lines of code, counting the number of lines (excluding comments) in the gold solution.

### 3.3.2 Evaluation

Without expert solutions, we cannot evaluate based on outcomes or landmarks. Instead, we use a simple heuristic metric, termed *Script-Executed*, to ensure the model sets up and executes the experiment without unresolved issues: we check if the script the agent was asked to run executes without exceptions for a minimum duration (see Appendix B.2 for details). The minimum duration ensures that the script was successful and didn’t just fail silently. While this method does not guarantee perfect evaluation, we find it surprisingly effective, as we show in our analysis in §4.3.

### 3.4 SUPER Benchmark

The Expert set consists of 45 collected tasks, where each problem is paired with a gold output for outcome-based evaluation and a small set of landmarks for our softer evaluation metric (an average of 3 landmarks per problem).

To estimate the lines of code (LOC) required for a task, we count the non-comment lines in the gold solution and for editing cells, the number of changed lines.<sup>3</sup> On average, each solution involved 44.3 LOC and 14.4 cells, highlighting the difficulty of these tasks due to potentially long agent trajectories. Consequently, the ability to solve these tasks provides an important signal on the performance of agents in handling long and complex tasks.

The Masked set contains 152 masked coding sub-problems derived from the 45 expert tasks. Like the Expert set, each sub-problem is paired with a gold output and landmarks. Table 3 shows the distribution of the extracted sub-problems across various categories, along with a representative solution for each category and the average lines of code (LOC) that were executed in the gold solution. Finally,

<sup>3</sup>The gold LOC may be significantly lower than what agents require, as experts can use their IDE or browser to read files, while agents must browse files via explicit commands. Additionally, experts may omit cells with failed attempts.

our automated set includes 604 problems, all from unique repositories.

To verify that the repositories used in our benchmarks are indeed ‘low-profile’, we count the number of GitHub stars in the source repositories as a proxy for popularity as shown in Table 1. Intuitively, popularity loosely correlates with the quality of documentation and readiness, which affects the difficulty of experiments execution. We see that the median number of stars for our repositories (14) is considerably lower than other comparable datasets (see Appendix A.1 for details).

## 4 Experiments

**Experimental Setup.** We limit execution time for each problem at 30 minutes (excluding API reply time) and run all tasks on sandboxed Modal instances (<https://modal.com>), enabling safe, concurrent evaluations. The total number of input tokens is limited to 400k per sub-problem and 600k for expert and auto-generated tasks combined. The compute cost per problem in Modal (excluding API costs) is 2-3 cents, which is negligible compared to API expenses. If the time or token limit is exceeded before submission, the task ends without submission (resulting in 0 accuracy, though agents are still scored on landmark evaluations). For sub-problems, we execute the provided ‘pre-execute’ cells (§3.2) and pass them as an existing history of actions to each agent. With end-to-end tasks (Expert and Auto), we simply prompt the agent with the task description. We confirmed that gold trajectories yield perfect scores on all metrics across three consecutive attempts.

**Underlying LLMs.** We experiment with agents based on commercial LLMs GPT-4o (gpt-4o-2024-08-06) and GPT-4o mini (gpt-4o-mini-2024-07-18) (OpenAI, 2023), as well as the open-source models Mixtral-8x22B-Instruct (Jiang et al., 2024)

and Llama 3.1 70B (Dubey et al., 2024) (Meta-Llama-3.1-70B-Instruct-Turbo), both served by <https://www.together.ai/>.

#### 4.1 Baselines

In this section, we describe the three baseline agents that we evaluate on SUPER.

**ReAct** (Yao et al., 2023) is a baseline agent that iteratively prompts the LLM to generate both an action and a natural language "thought," using the interaction history as context. At each step, the action (a bash or Python command) is executed in the environment, and a `<thought, action, observation>` tuple is added to the history. This process continues until the agent submits an answer or reaches token or compute limits.

A challenge in running experiments is that output observations can be extremely long (e.g., training scripts or dependency installations producing 10k-40k tokens). Since ReAct agents accumulate history at each step, token usage increases rapidly. Given that agents operate within a fixed token or cost budget, this can lead to task failures. We thus apply truncation strategies for all our agents to mitigate this issue. See Appendix C for details.

**ReAct-SUPER.** The ability to execute Python and bash commands, in theory, allows agents to perform any necessary task. However, these actions are still limited compared to humans who can use IDEs to browse and edit files. In our early experiments, we indeed found that agents struggle to edit files (e.g., change configs) using just bash.

To address this challenge, we introduce an additional **edit** action, similar in spirit to the *Agent-Computer Interfaces* from SWE-Agent (Yang et al., 2024a). The edit command takes three parameters: the file name, the content to be replaced, and the new content. If the exact content isn't found (e.g., due to missing whitespaces), the tool provides fixing suggestions. See App. E for more details.

**SWE-Agent** (Yang et al., 2024a) is a ReAct-based agent, originally designed to solve GitHub *issues*. Like ReAct-SUPER, it provides agents with tools that allow editing of files, but also tools for reading files, scrolling through their contents and more (see original paper for details). We implement SWE-Agent in our environment and adapt its prompts to handle the execution of research tasks.

**Reflecting Agents.** To explore whether agents can improve by reflecting on failures, we evaluate

Agent	Model	Acc.	Landm.
SWE-Agent	GPT-4o	16.3 ± 2.1	36.8 ± 2.3
React	GPT-4o	12.2 ± 1.0	33.6 ± 0.9
React-Super	GPT-4o	14.4 ± 2.2	42.6 ± 2.9
SWE-Agent	GPT-4o mini	3.3	16.1
React-Super	GPT-4o mini	5.6	20.6
SWE-Agent	Llama 3.1 70B	5.6	4.8
React-Super	Llama 3.1 70B	6.1	9.6
SWE-Agent	Mixtral 8x22B	1.1	0.0
React-Super	Mixtral 8x22B	3.3	3.7

Table 4: Results on Expert, with GPT-4o numbers averaged across 3 seeds.

agents with a reflection mechanism (Shinn et al., 2023). When an agent fails to submit an answer on its first attempt, the LLM is prompted to reflect on the trajectory and devise a plan to avoid similar mistakes. This reflection is incorporated into the agent's prompt, and the agent retries the task. The agent is given  $k$  attempts, with each attempt allocated  $1/k^{th}$  of the token budget.

#### 4.2 Results

**Expert Set.** Table 4 presents results for the Expert set, averaged across three seeds for the most performant LLM (decoding temperature is 0.2). The low accuracies (12.2-16.3) suggest that current agents cannot yet perform this task well. However, the higher landmarks metric shows that agents often make partial progress, suggesting they could still assist in setting up repositories.

**Masked Set.** Table 6 presents results on the Masked set, demonstrating that SWE-agent correctly solves a significant portion (46.1%) of the challenges, but that most sub-problems are still unsolved. The higher landmarks evaluation score (74.9%) suggests that agents often make progress towards solving the sub-problems, even if some of the steps might not necessarily be correct.

We find that SWE-agent performs better than ReAct-SUPER with GPT-4o as the LLM, but slightly worse with all weaker models, suggesting that weaker models were less effectively in leveraging SWE-Agent tools. The open-source Mixtral and Llama reach significantly lower scores on both the Masked and Expert sets.

**Auto Set.** We show in Table 5 results for the Auto tasks, where ranking of models and agents are mostly consistent with the ranking of the models on the Masked set, suggesting potential usefulness of this set for future development.



Agent	Model	Script-Executed
SWE-Agent	GPT-4o	18.0
React	GPT-4o	14.0
React-Super	GPT-4o	18.8
SWE-Agent	GPT-4o mini	5.2
React	GPT-4o mini	16.0
React-Super	GPT-4o mini	14.8

Table 5: Results on 250 of the Auto tasks.

Agent	Model	Acc.	Landm.
SWE-Agent	GPT-4o	46.1	74.9
React	GPT-4o	37.0	65.7
React-Super	GPT-4o	41.6	72.5
SWE-Agent	GPT-4o mini	27.0	51.8
React-Super	GPT-4o mini	31.5	58.3
SWE-Agent	Llama 3.1 70B	17.4	35.0
React-Super	Llama 3.1 70B	22.8	38.3
SWE-Agent	Mixtral 8x22B	9.5	26.6
React-Super	Mixtral 8x22B	7.0	13.2

Table 6: Results of our baselines on SUPER (Masked) with different underlying LLMs.

**Ablations.** Comparing React with ReAct-SUPER shows that the editing function enables the agent to hit more landmarks (72.5% vs 65.7%) and produce more accurate answers (41.6% vs 37.0%). We find that without the editing command, the agent usually resorts to editing files with the sed command, which is designed for simple single-line edits.

**Can agents that reflect do better?** We next evaluate if retrying after reflecting on failures can improve the performance of our baseline agents, with  $k = 3$  retries. As shown in Table 7, the additional retries with reflections have a positive but minimal impact on the score. If models lack the inherent ability to resolve some of these issues, retrieval with reflections are not likely to help.

### 4.3 Error Analysis

The Masked set categorizes each problem, allowing us to break down performance of agents (Table 3). We find that the hardest categories for the agent are data (27%), configuration (38%) and goal (43% accuracy), whereas CPU, issues and dependencies are easier (73%, 61% and 54% respectively). These findings suggest that agents are better at solving sub-problems where there is a specific error message to be solved (such as CPU support errors, incompatible dependencies, or exceptions) than more open-ended problems such as configuring data loading for a custom dataset.

Specifically, in the latter case, we find that agents

Agent	Acc.	Landm.
ReAct-SUPER	41.6	72.5
Reflexion (Shinn et al., 2023)	45.4	76.6

Table 7: Results of the ReAct-SUPER agent (using GPT-4o) with and without Reflexion on the Masked set. While retrying with reflection does help improve the submission rate and accuracy, SUPER benchmark still remains challenging.

often skip reviewing repository code, tending to hallucinate script or function arguments (e.g., adding `n_examples=10` when no such argument exists) or missing available script parameters and attempting to modify them unsuccessfully in the code. Moreover, once agents commit to an approach, they rarely reconsider it until failure.

These issues suggest that agents should be better designed to analyze repositories and consider multiple solution approaches. We provide all trajectories of ReAct-SUPER and SWE-Agent in our code repository.

**Effectiveness of Proxy Metric.** While the Expert and Masked sets are evaluated using expert solutions, the Auto set lacks solutions, so we rely on the weaker Script-Executed proxy metric (§3.3.2). To validate its reliability, we compared Script-Executed with the accuracy and landmark metrics using ReAct-SUPER trajectories on the Masked set. Script-Executed agreed with landmark (assuming a score of 1 when landmark > 0.5) in 90% of cases and with accuracy in 69%. We found two key cases of disagreement with the landmark metric: (1) when the script ran correctly to get the answer but encountered an exception afterward (e.g., while creating a figure), and (2) when the script ran just long enough to appear successful by the proxy metric but later failed due to misconfiguration or exceptions without reaching the answer.

## 5 Conclusion

We introduce SUPER, a benchmark for evaluating LLM-based agents on tasks from low-profile research repositories. Our empirical results demonstrate the difficulty of the benchmark, even for leading LLMs like GPT-4o. SUPER highlights key challenges in building autonomous execution agents, such as repository reasoning and code editing, and we hope it will drive measurable progress in tackling this important problem.

## Limitations

**Dataset Size.** The dataset size of our benchmark, comprising 45 and 152 sub-problems, is smaller compared to some other benchmarks available for agent evaluation, which could potentially affect the statistical significance of performance evaluations. However, the use of smaller, high-quality benchmarks is not uncommon. For instance, benchmarks such as HUMAN-EVAL (Chen et al., 2021), CLASSEVAL (Du et al., 2023), and BAMBOOGLE (Press et al., 2023) contain 164, 100, and 125 examples respectively, and are widely used for assessing model performance. In addition, recent work has suggested that reducing large datasets to as few as 100 examples does not diminish their effectiveness (Maia Polo et al., 2024). Moreover, smaller-sized datasets offer the advantage of being less expensive to operate, thus providing better accessibility for researchers with limited resources, particularly when running interactive agents in environments that generate long outputs. Finally, our provided Auto set with 604 problems offers problems purposed for development, which alleviates the risk of overfitting to the evaluation sets.

**Programming Languages and Domains.** We have only collected solutions written in Python, and our environment only supports that programming language. We focus mostly on text-based repositories. While the challenges associated with running these repositories likely overlap with other domains, increasing the diversity of the repository domains could be beneficial.

**Evaluation Based on External Resources.** Running benchmarks in realistic environments often depend on external resources. In our case, agents rely on availability of resources such as GitHub, pip and datasets, which we cannot control across runs. While completely sand-boxed setups could have allowed for a more controlled evaluation, we opt for fidelity, similarly to e.g. benchmarks for web agents that rely on access to real websites (Mialon et al., 2024; He et al., 2024, inter alia).<sup>4</sup>

## Ethical Considerations

While autonomous research execution agents could significantly enhance research advancements, there is a risk of over-reliance on these agents, which

---

<sup>4</sup>Note that all our evaluations are run using the same base Docker image with sand-boxed code execution and should be reproducible; barring any external changes.

could lead to conclusions drawn based on incorrect implementations of agents, and careless actors not checking the agent’s reproduction work carefully.

## Acknowledgments

We thank Ori Yoran for his valuable comments and suggestions, and the anonymous reviewers for their helpful feedback. We also thank the Upworker expert coders for their work on the solutions to the Expert set problems.

## References

- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. [Program synthesis with large language models](#). *arXiv preprint arXiv:2108.07732*.
- Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. [MultiPL-E: A scalable and extensible approach to benchmarking neural code generation](#). *arXiv preprint arXiv:2208.08227*.
- Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Anton Lozhkov, Carolyn Anderson, and Arjun Guha. 2023. [Can it edit? evaluating the ability of large language models to follow code editing instructions](#). *arXiv preprint arXiv:2312.12450*.
- Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. 2023. [FireAct: Toward language agent fine-tuning](#). *ArXiv, abs/2310.05915*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. [Evaluating large language models trained on code](#). *arXiv preprint arXiv:2107.03374*.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. [BERT: Pre-training of deep bidirectional transformers for language understanding](#). In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.
- Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2024. [CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion](#). *Advances in Neural Information Processing Systems*, 36.

- Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. [ClassEval: A manually-crafted benchmark for evaluating llms on class-level code generation](#). *arXiv preprint arXiv:2308.01861*.
- Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, et al. 2024. [The llama 3 herd of models](#). *Preprint*, arXiv:2407.21783.
- Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. 2024. [DS-Agent: Automated data science by empowering large language models with case-based reasoning](#). *ArXiv*, abs/2402.17453.
- Md Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2023. [FixEval: Execution-based evaluation of program fixes for programming problems](#). In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 11–18. IEEE.
- Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. 2024. [WebVoyager: Building an end-to-end web agent with large multimodal models](#). In *Annual Meeting of the Association for Computational Linguistics*.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. [Measuring coding challenge competence with apps](#). *NeurIPS*.
- Sirui Hong, Yizhang Lin, Bangbang Liu, Binhao Wu, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Lingyao Zhang, Mingchen Zhuge, et al. 2024. [Data interpreter: An llm agent for data science](#). *arXiv preprint arXiv:2402.18679*.
- Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2024. [MLAgentBench: Evaluating language agents on machine learning experimentation](#). *Preprint*, arXiv:2310.03302.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. [LiveCodeBench: Holistic and contamination free evaluation of large language models for code](#). *arXiv preprint arXiv:2403.07974*.
- Peter Jansen, Marc-Alexandre Cot'e, Tushar Khot, Erin Bransom, Bhavana Dalvi, Bodhisattwa Prasad Majumder, Oyvind Tafjord, and Peter Clark. 2024. [DISCOVERYWORLD: A virtual environment for developing and evaluating automated scientific discovery agents](#). *arXiv preprint arXiv:2406.06769*.
- Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, L lio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Th ophile Gervet, Thibaut Lavril, Thomas Wang, Timoth e Lacroix, and William El Sayed. 2024. [Mixtral of experts](#). *Preprint*, arXiv:2401.04088.
- 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*.
- Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, S. W. Yih, D. Fried, S. Wang, and T. Yu. 2023. [DS-1000: A natural and reliable benchmark for data science code generation](#). In *International Conference on Machine Learning (ICML)*.
- Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. [API-bank: A comprehensive benchmark for tool-augmented llms](#). In *The 2023 Conference on Empirical Methods in Natural Language Processing*.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, R mi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. [Competition-level code generation with alphacode](#). *Science*, 378(6624):1092–1097.
- Siyi Liu, Chen Gao, and Yong Li. 2024. [Large language model agent for hyper-parameter optimization](#). *arXiv preprint arXiv:2402.01881*.
- Tianyang Liu, Canwen Xu, and Julian McAuley. 2023a. [RepoBench: Benchmarking repository-level code auto-completion systems](#). *arXiv preprint arXiv:2306.03091*.
- Yuliang Liu, Xiangru Tang, Zefan Cai, Junjie Lu, Yichi Zhang, Yanjun Shao, Zexuan Deng, Helan Hu, Zengxian Yang, Kaikai An, Ruijun Huang, Shuzheng Si, Sheng Chen, Haozhe Zhao, Zhengliang Li, Liang Chen, Yiming Zong, Yan Wang, Tianyu Liu, Zhiwei Jiang, Baobao Chang, Yujia Qin, Wangchunshu Zhou, Yilun Zhao, Arman Cohan, and Mark Gerstein. 2023b. [ML-Bench: Evaluating large language models for code generation in repository-level machine learning tasks](#). *arXiv preprint arXiv:2311.09835*.
- Felipe Maia Polo, Lucas Weber, Leshem Choshen, Yuekai Sun, Gongjun Xu, and Mikhail Yurochkin. 2024. [tinyBenchmarks: evaluating llms with fewer examples](#). *arXiv preprint arXiv:2402.14992*.
- Gr goire Mialon, Cl mentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2024. [GAIA: a benchmark for general AI assistants](#). In *The Twelfth International Conference on Learning Representations*.

- OpenAI. 2023. [GPT-4 technical report](#). *Preprint*, arXiv:2303.08774.
- OpenDevin Team. 2024. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents. <https://github.com/OpenDevin/OpenDevin>. Accessed: 06/11/2024.
- Joon Sung Park, Joseph O’Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. [Generative agents: Interactive simulacra of human behavior](#). In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–22.
- Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive APIs. *arXiv preprint arXiv:2305.15334*.
- Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah Smith, and Mike Lewis. 2023. [Measuring and narrowing the compositionality gap in language models](#). In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5687–5711, Singapore. Association for Computational Linguistics.
- Sheeba Samuel and Daniel Mietchen. 2022. [Computational reproducibility of jupyter notebooks from biomedical publications](#). *GigaScience*, 13.
- Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. 2023. [TaskBench: Benchmarking large language models for task automation](#). *arXiv preprint arXiv:2311.18760*.
- Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. [Reflexion: Language agents with verbal reinforcement learning](#). In *NeurIPS*.
- Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. 2024. [Trial and error: Exploration-based trajectory optimization of LLM agents](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7584–7600, Bangkok, Thailand. Association for Computational Linguistics.
- Shane Storcks, Keunwoo Yu, Ziqiao Ma, and Joyce Chai. 2023. [NLP reproducibility for all: Understanding experiences of beginners](#). In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 10199–10219, Toronto, Canada. Association for Computational Linguistics.
- Wilson L Taylor. 1953. [“cloze procedure”: A new tool for measuring readability](#). *Journalism quarterly*, 30(4):415–433.
- Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2024. [DebugBench: Evaluating debugging capability of large language models](#). *arXiv preprint arXiv:2401.04621*.
- Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023a. [Voyager: An open-ended embodied agent with large language models](#). *arXiv preprint arXiv:2305.16291*.
- Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. [Executable code actions elicit better llm agents](#). *arXiv preprint arXiv:2402.01030*.
- Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2023b. [Mint: Evaluating llms in multi-turn interaction with tools and language feedback](#). *arXiv preprint arXiv:2309.10691*.
- Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. 2024. [OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments](#). *arXiv preprint arXiv:2404.07972*.
- John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024a. [SWE-agent: Agent-computer interfaces enable automated software engineering](#). *Preprint*, arXiv:2405.15793.
- Zhiyu Yang, Zihan Zhou, Shuo Wang, Xin Cong, Xu Han, Yukun Yan, Zhenghao Liu, Zhixing Tan, Pengyuan Liu, Dong Yu, et al. 2024b. [MatPlotAgent: Method and evaluation for llm-based agentic scientific data visualization](#). *arXiv preprint arXiv:2402.11453*.
- Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. [Webshop: Towards scalable real-world web interaction with grounded language agents](#). *Advances in Neural Information Processing Systems*, 35:20744–20757.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. [React: Synergizing reasoning and acting in language models](#). In *The Eleventh International Conference on Learning Representations*.
- Da Yin, Faeze Brahman, Abhilasha Ravichander, Khyathi Chandu, Kai-Wei Chang, Yejin Choi, and Bill Yuchen Lin. 2024. [Agent lumos: Unified and modular training for open-source language agents](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12380–12403, Bangkok, Thailand. Association for Computational Linguistics.
- Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. [Repocoder: Repository-level code completion through iterative retrieval and generation](#). *arXiv preprint arXiv:2303.12570*.
- Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan

Bisk, Daniel Fried, Uri Alon, et al. 2023. [WebArena: A realistic web environment for building autonomous agents](#). *arXiv preprint arXiv:2307.13854*.

## A Additional Benchmark Details

### A.1 Repositories

Table 8 shows information about the 45 source repositories used to create the Expert and Masked sets, including their name, original GitHub link and the number of stars on GitHub.

Task	GitHub	Stars
colbert	<a href="https://github.com/stanford-futuredata/ColBERT">https://github.com/stanford-futuredata/ColBERT</a>	2826
textbox	<a href="https://github.com/RUCAIBox/TextBox">https://github.com/RUCAIBox/TextBox</a>	1069
amrbart	<a href="https://github.com/goodbai-nlp/AMRBART">https://github.com/goodbai-nlp/AMRBART</a>	94
g-transformer	<a href="https://github.com/baoguangsheng/g-transformer">https://github.com/baoguangsheng/g-transformer</a>	43
pie-perf	<a href="https://github.com/madaan/pie-perf">https://github.com/madaan/pie-perf</a>	80
safetybench	<a href="https://github.com/thu-coai/SafetyBench">https://github.com/thu-coai/SafetyBench</a>	138
discoDisCo	<a href="https://github.com/gucorpling/DisCoDisCo">https://github.com/gucorpling/DisCoDisCo</a>	6
acqsurvey	<a href="https://github.com/rahmanidashiti/acqsurvey">https://github.com/rahmanidashiti/acqsurvey</a>	11
curriculum_learning	<a href="https://github.com/adymaharana/curriculum_learning">https://github.com/adymaharana/curriculum_learning</a>	9
spa	<a href="https://github.com/OceannTwT/SPA">https://github.com/OceannTwT/SPA</a>	5
mezo	<a href="https://github.com/princeton-nlp/MeZO">https://github.com/princeton-nlp/MeZO</a>	1016
mode-connectivity-plm	<a href="https://github.com/thunlp/mode-connectivity-plm">https://github.com/thunlp/mode-connectivity-plm</a>	7
mbib	<a href="https://github.com/Media-Bias-Group/MBIB">https://github.com/Media-Bias-Group/MBIB</a>	22
quantifying-stereotypes-in-...	<a href="https://github.com/nlply/quantifying-stereotypes-in-language">https://github.com/nlply/quantifying-stereotypes-in-language</a>	1
rah-kbqa	<a href="https://github.com/yanmenxue/rah-kbqa">https://github.com/yanmenxue/rah-kbqa</a>	6
dir-gnn	<a href="https://github.com/wuyxin/dir-gnn">https://github.com/wuyxin/dir-gnn</a>	115
unsupervisedhierarchicalsymbolic..	<a href="https://github.com/SiyuLou/UnsupervisedHierarchicalSymbolicRegression">https://github.com/SiyuLou/UnsupervisedHierarchicalSymbolicRegression</a>	0
conv_graph	<a href="https://github.com/huawei-noah/noah-research/tree/master/conv_graph">https://github.com/huawei-noah/noah-research/tree/master/conv_graph</a>	0
mera	<a href="https://github.com/ai-forever/MERA">https://github.com/ai-forever/MERA</a>	55
pira	<a href="https://github.com/C4AI/Pira">https://github.com/C4AI/Pira</a>	5
pet	<a href="https://github.com/timoschick/pet">https://github.com/timoschick/pet</a>	1618
transnormerllm	<a href="https://github.com/opennlp/abstractive-summarization">https://github.com/opennlp/abstractive-summarization</a>	221
bert-lnl	<a href="https://github.com/uds-lsv/BERT-LNL">https://github.com/uds-lsv/BERT-LNL</a>	9
blockskim	<a href="https://github.com/chandlerguan/blockskim">https://github.com/chandlerguan/blockskim</a>	6
data_label_alignment	<a href="https://github.com/gyaaney/data-label-alignment">https://github.com/gyaaney/data-label-alignment</a>	3
hype	<a href="https://github.com/yuanhy1997/HyPe">https://github.com/yuanhy1997/HyPe</a>	13
paraphrase-nli	<a href="https://github.com/matejklemen/paraphrase-nli">https://github.com/matejklemen/paraphrase-nli</a>	3
powerfulpromptft	<a href="https://github.com/zhengxiangshi/powerfulpromptft">https://github.com/zhengxiangshi/powerfulpromptft</a>	71
robust_prompt_classifier	<a href="https://github.com/adianliusie/robust-prompt-classifier">https://github.com/adianliusie/robust-prompt-classifier</a>	5
align-to-distill	<a href="https://github.com/ncsoft/Align-to-Distill">https://github.com/ncsoft/Align-to-Distill</a>	4
inbedder	<a href="https://github.com/zhang-yu-wei/InBedder">https://github.com/zhang-yu-wei/InBedder</a>	20
transpolymer	<a href="https://github.com/ChangwenXu98/TransPolymer">https://github.com/ChangwenXu98/TransPolymer</a>	51
memorizing-transformers-...	<a href="https://github.com/lucidrains/memorizing-transformers-pytorch">https://github.com/lucidrains/memorizing-transformers-pytorch</a>	622
multi3woz	<a href="https://github.com/cambridgeltl/multi3woz">https://github.com/cambridgeltl/multi3woz</a>	14
galore	<a href="https://github.com/jiaweizhao/galore">https://github.com/jiaweizhao/galore</a>	1332
amos	<a href="https://github.com/microsoft/amos">https://github.com/microsoft/amos</a>	24
glee	<a href="https://github.com/genezc/Glee">https://github.com/genezc/Glee</a>	9
parallel-context-windows	<a href="https://github.com/AI21Labs/Parallel-Context-Windows">https://github.com/AI21Labs/Parallel-Context-Windows</a>	98
logme-nlp	<a href="https://github.com/mainlp/logme-nlp">https://github.com/mainlp/logme-nlp</a>	5
mixup-amp	<a href="https://github.com/pai-smallsallyourneed/mixup-amp">https://github.com/pai-smallsallyourneed/mixup-amp</a>	4
upet	<a href="https://github.com/wjn1996/UPET">https://github.com/wjn1996/UPET</a>	2
dpt	<a href="https://github.com/xyaoooo/dpt">https://github.com/xyaoooo/dpt</a>	6
team	<a href="https://github.com/declare-lab/team">https://github.com/declare-lab/team</a>	22
cet	<a href="https://github.com/zzz47zzz/CET">https://github.com/zzz47zzz/CET</a>	18
linkbert	<a href="https://github.com/michiyasunaga/LinkBERT">https://github.com/michiyasunaga/LinkBERT</a>	411

Table 8: Details of the 45 repositories used in SUPER along with GitHub link and star information as of September 3rd, 2024.

Adding to the information in Table 1, we show below the average and median star ratings for other comparable benchmarks (all star ratings are collected as of September 3rd, 2024). This was computed automatically from the GitHub API based on the repositories listed in Jimenez et al. (2024), Lai et al. (2023) and Liu et al. (2023b) (we group together both the train and test repositories mentioned in this table).

dataset	# repos	stars (mean (median))
SWE-Bench	12	27,844 (12,557)
DS1000	8	55,227 (35,309)
MLBench	18	13,099 (9,632)
<b>SUPER (Expert)</b>	45	224 (14)
<b>SUPER (Auto)</b>	604	96 (23)

### A.2 Expert Tasks

In Table 9 we present six examples for problems from the Expert set.

<p>Train a teacher model on the iwslt14-de-en dataset (available from the download script in the repo), then train a teacher model and evaluate it. Report the BLEU score on the test set for the student model as a json structured as follows: "bleu": 0.0 (replace 0.0 with the actual value).</p> <p>Additional instructions:</p> <ol style="list-style-type: none"> <li>1. Once loaded/tokenized, keep only the first 10 rows of each set in the dataset.</li> <li>2. Train for only one epoch and with batch_size=1.</li> <li>3. To evaluate the teacher model, generate predictions on the test set using 'generate.py' with beam_size=5. Then take the evaluation score with the 'score.py' script, with '-ignore-case -sacrebleu'.</li> </ol> <p>Git repository: <a href="https://github.com/ncsoft/Align-to-Distill">https://github.com/ncsoft/Align-to-Distill</a></p>
<p>Evaluate the generations of my code improving model which are provided in <a href="https://drive.google.com/file/d/1izsliF5cd_NAZsOaZvrrQF3NAsoP8lHf/view?usp=sharing">https://drive.google.com/file/d/1izsliF5cd_NAZsOaZvrrQF3NAsoP8lHf/view?usp=sharing</a> (v1 vs v0). Once evaluated, report the result problem_id and input_acc for each problem of the dataset, as a json list of dictionaries structured as follows: [{"problem_id": "", "input_acc": 0.0}] (replace "" and 0.0 with the actual values).</p> <p>Additional instructions:</p> <ol style="list-style-type: none"> <li>1. Set "num_trials": 2 in the evaluation configuration file to reduce computation time.</li> <li>2. Load only the first 10 rows of the dataset.</li> </ol> <p>Git repository: <a href="https://github.com/madaan/pie-perf">https://github.com/madaan/pie-perf</a></p>
<p>Evaluate the performance of gpt2-small (openai-community/gpt2) on our own sentence classification dataset, available at <a href="https://drive.google.com/file/d/1WhDfjunl1VRxK6Vhakhm05C8IABSa1Qc/view?usp=drive_link">https://drive.google.com/file/d/1WhDfjunl1VRxK6Vhakhm05C8IABSa1Qc/view?usp=drive_link</a>, using the PCW method with 1 window. Report the accuracy as a json structured as follows: "accuracy": 0.0 (replace 0.0 with the actual value).</p> <p>Additional instructions:</p> <ol style="list-style-type: none"> <li>1. Run for only 1 epoch.</li> <li>2. Perform only 1 run of the experiment (n-runs).</li> <li>3. label 0 stands for 'negative' and label 1 stands for 'positive'.</li> </ol> <p>Git repository: <a href="https://github.com/AI21Labs/Parallel-Context-Windows">https://github.com/AI21Labs/Parallel-Context-Windows</a></p>
<p>Train and evaluate the bert-base-uncased model on the SST-2 dataset using the AMP method. Report the maximum test accuracy as a json structured as follows: "max_test_acc": 0.0 (replace 0.0 with the actual value).</p> <p>Additional instructions:</p> <ol style="list-style-type: none"> <li>1. Only load the first 10 examples of each (training/validation/test) set in the dataset.</li> <li>2. Train only one batch.</li> </ol> <p>Git repository: <a href="https://github.com/pai-smallisallyourneed/mixup-amp">https://github.com/pai-smallisallyourneed/mixup-amp</a></p>
<p>Use the lm-evaluation-harness to evaluate ai-forever/rugpt3small_based_on_gpt2 on the chegeka test set. Report "metric" and "metric_stderr" as a json structured as follows: {"metric": 0.0, "metric_stderr": 0.0} (replace 0.0 with the actual values).</p> <p>Additional instructions:</p> <ol style="list-style-type: none"> <li>1. Load only the first 10 rows of the dataset.</li> <li>2. Use the following hyperparameters: batch_size=1, num_fewshot=4</li> </ol> <p>Git repository: <a href="https://github.com/ai-forever/MERA">https://github.com/ai-forever/MERA</a></p>
<p>Fine-tune the roberta-base model on the MRPC dataset using the GaLore optimizer with the 'run_glue.py' script. Report the accuracy and f1 score as a json structured as follows: {"accuracy": 0.0, "f1": 0.0} (replace 0.0 with the actual values).</p> <p>Additional instructions:</p> <ol style="list-style-type: none"> <li>1. Only load the first 10 examples of each (training/validation/test) set in the dataset.</li> <li>2. Run for only 1 epoch.</li> <li>3. Use the following hyperparameters: '-enable_galore -lora_all_modules -max_length 512 -seed=1234 -lora_r 4 -galore_scale 4 -per_device_train_batch_size 16 -update_proj_gap 500 -learning_rate 3e-5</li> </ol> <p>Git repository: <a href="https://github.com/jiaweizhao/galore">https://github.com/jiaweizhao/galore</a></p>

Table 9: Sampled problems from the Expert set.

## B Automatic Generation of Tasks

### B.1 Tasks Generation

The automatic tasks generation involves two high-level steps: filtering repositories, and creating tasks for repositories.

**Step 1: Filtering Repositories.** We start from 5915 repositories listed by “paperswithcode” to be added on year 2021 or later and having modality ‘Text’. We then automatically clone each of these repositories, filtering out those where: (1) cloning failed, (2) use LLM APIs (based on the occurrence of importing Python packages of LLM provider such as OpenAI, Anthropic, etc.) or (3) no readme file was found.

On the remaining repositories, we then use the following prompt on GPT-4o (gpt-4o-2024-08-06) to filter repositories.

Repositories Filtering Prompt
<p>Your task is to analyze a GitHub repository and answer the following questions:</p> <ol style="list-style-type: none"> <li>1. Can this repository run on one of these datasets (they should be explicitly mentioned in readme or grepped code)? SST-2, QNLI, QQP, MRPC, RTE, MMLU, wikitext, yelp, ai2_arc, hellaswag, winogrande, piqa, humaneval, truthful_qa, cnn_dailymail, mnist, xsquad, squad, xnli, mnli, multi_nli, ag_news, WikiText, gsm8k, cola, triviaqa, hotpotqa, humaneval, fever, boolq, openbookqa, drop, coqa, GLUE.</li> </ol>

2. Can the repository be run on a CPU? It is acceptable if it runs slowly, as we will use small models and datasets. The repository should be rejected only if it relies on specific GPU acceleration methods such as LoRA adapters, if it aims to improve GPU utilization or otherwise relies on specific GPU features.
3. Can the repository be used with any of the following model families: BERT, T5, RoBERTa, GPT-2, GPT-Neo, DeBERTa, DistilBERT, BART, Pythia, OPT?
4. Does the README provide an example or instructions on how to run an executable Python or Bash file to start an experiment? If so, provide the name of the executable file. For example:
  - ↪ `python run_glue.py --model_name_or_path roberta-base --task_name mrpc --output_dir results/ft/roberta_base/mrpc``

Return a json file in this format:

```
{
  "q_supported_dataset_name": "hellaswag",
  "q_supported_dataset_reason": "The README mentions the use of the hellaswag dataset, which is
  ↪ one of the supported datasets.",
  "q_cpu": true,
  "q_cpu_reason": "The repository does not rely on specific GPU acceleration methods such as LORA
  ↪ adapters or repositories that improve GPU utilization.",
  "q_model_families": true,
  "q_model_families_reason": "The repository supports the BERT model family, as indicated by the
  ↪ presence of 'bert-large' in the model configuration.",
  "q_execute_example": true,
  "q_execute_example_reason": "The readme provides an example for running a training pipeline on
  ↪ the hellaswag dataset."
}
```

We then keep repositories where `q_supported_dataset_name`, `q_cpu`, `q_model_families` and `q_execute_example` are all predicted by the LLM to be true, resulting in 1006 repositories. Note that this filtering process is rather conservative; it is likely possible to get more high-quality tasks from repositories that were filtered out.

**Step 2: Generating Tasks.** We iterate the filtered repositories and prompt GPT-4o (same version) with the following text to generate the tasks.

#### Tasks Generation Prompt

Create an experiment-running task based on the provided README file of a research code repository.  
 ↪ Output a json dictionary with the following structure:  
 {"thought": "...", "output": {"task": "...", "entrypoint": "...}}

Instructions:

1. Choose a specific script (either Bash or Python) for which the README provides an example. If no example is found, skip the repository (return an empty dictionary). The "entrypoint" field should specify the script file to be run without any arguments. Use a specific file path, even if the script is typically executed using a module (e.g., "train/run.py", not "python -m train.run").

Ensure that the script selected is for running experiments (e.g., evaluation or fine-tuning) and not a utility script (such as a server or data processing script).

2. The task description should include the following:
  1. A statement that reflects the goal of the repository. For instance, use "Fine-tune a model with the question answering infused pre-training method" rather than just "fine-tune a model," or "Pre-train a reduced-scale model" rather than just "pre-train a model." If the repository lacks sufficient detail, you may keep this generic.
  2. The specific dataset to be used. If the repository doesn't specify supported datasets, skip the repository. For repositories mentioning a group of datasets (e.g., GLUE), choose a specific dataset (e.g., MRPC) instead of just mentioning the group. If the README indicates that the data is unavailable (e.g., "data will be uploaded soon"), skip the repository.
  3. The Python script/Bash file/entry point to be run, which should match the "entrypoint" field.
  4. The model to be used. To ensure the task is feasible with minimal compute, select a small or base model from one of the following families: bert, roberta, t5, gpt, opt, deberta, distilbert, bart, pythia.  
 Then, select the smallest model in the family, based on this mapping: bert: bert-base, t5: google-t5/t5-small, gpt: openai-community/gpt2, deberta: deberta-base, bart: bart-base, pythia: EleutherAI/pythia-70m, OPT: facebook/opt-125m  
 If the README or repository content does not explicitly mention the model family or model size, skip the repository.



Here are a few examples.  
[...]

We filter out repositories/tasks when (1) the model decides to skip (e.g. if no indication of model or dataset), or (2) the provided script selected by the model cannot be found in the repository, or is not a Python or bash file.

## B.2 Tasks Evaluation (Script-Executed metric)

We use a simple heuristic to determine if a script was run successfully: we check if the script was executed without any exceptions being raised (according to printed output), and if it was executed for at least  $s$  seconds. We use time limit to make sure we avoid any quick failures that did not raise an exception, such as messages about missing arguments. Based on the gold expert solutions, we find that  $s = 10$  is a good trade-off to distinguish unsuccessful short runs from successful ones. Importantly, this evaluation metric is an approximation, and can surely be incorrect or even manipulated by agents that are aware of it. Yet as we show in §4.3, we found it to match the landmarks evaluation in 90% of the cases, hopefully making it useful for development, and as a basis for potentially creating even larger sets of tasks for development and training purposes.

## C Prompts and Interaction History

### C.1 ReAct-SUPER

We use the following prompt for ReAct-SUPER.

#### ReAct-SUPER Prompt

```
Interact in a jupyter notebook to solve the request.
Output a thought and an action.
Thought can reason about the current situation, and action allows you to interact with the Jupyter
↳ notebook. There are three types of actions:
(1) execute, which executes a notebook cell with the command under the "content" field. This could
↳ be a bash command (beginning with !), a python command, or magic command (beginning with %).
(2) edit, which allows you to replace existing lines in a file with new lines. To replace lines,
↳ you will need to provide the lines as they appear before the replacement, and the lines to
↳ replace them with.
In the "content" field, format should be
...
{filename}
<<<BEFORE_EDIT>>>
lines before edit
lines before edit
<<<AFTER_EDIT>>>
lines after edit
lines after edit
...
- Include the entire lines WITH indentation. After editing, make sure indentation is correct.
(3) submit, which submits the final answer (or null if there's no answer to submit), formatted as
↳ JSON (make sure to use double quotes for keys and values).

Always output exactly one "thought" and one "action", and encapsulate the content with
↳ <content>...</content> tags.
I will provide the cell output (observation) for you.

Tips:
- When a github repository is given, you should start by cloning it. You can read the readme file
↳ and browse the repo for installation and usage instructions.
- Avoid using conda, instead use pip for package management.
- Each execution has a 5-minute timeout. After this period, you can decide whether to interrupt the
↳ kernel or continue waiting.
  You should interrupt if: (1) you are prompted with an interactive message (e.g., "Do you want to
  ↳ continue? [Y/n]"). In that case, you can bypass it by executing echo "y" | command, or (2)
  ↳ the execution is taking longer than expected because the wrong task is being executed, such
  ↳ as a different model or incorrect number of epochs.
  You should continue to wait if the current execution is correct and proceeding as expected.

Here are some examples.
```

```

Git repository: EleutherAI/lm-evaluation-harness
Query: Evaluate EleutherAI/gpt-j-6B on hellaswag. What's the score?
Thought: First, I need to clone the GitHub repository 'EleutherAI/lm-evaluation-harness'.
Action: execute
<content>
!git clone https://github.com/EleutherAI/lm-evaluation-harness.git
</content>
Observation:
...
Cloning into 'lm-evaluation-harness'...
...
Thought: The repository has been cloned. Next, I should check the README file for installation and
↳ usage instructions to proceed with evaluating EleutherAI/gpt-j-6B on hellaswag.
Action: execute
<content>
!cat lm-evaluation-harness/README.md
</content>
Observation:
...
# Language Model Evaluation Harness ... truncated ...
...
Thought: The README provides detailed instructions for installing the `lm-eval` package and
↳ evaluating models like GPT-J-6B on tasks like hellaswag. I need to install the package and its
↳ dependencies first.
Action: execute
<content>
!pip install -e lm-evaluation-harness/.
</content>
Observation:
...
Obtaining file:///home/nora/lm-evaluation-harness
Installing build dependencies ... truncated ...
...
Thought: The `lm-eval` package and its dependencies have been installed successfully. Now, I can
↳ proceed with evaluating the EleutherAI/gpt-j-6B model on the hellaswag task as per the
↳ instructions in the README.
Action: execute
<content>
!lm_eval --model hf --model_args pretrained=EleutherAI/gpt-j-6B --tasks hellaswag --device cuda:0
↳ --batch_size 8
</content>
Observation:
...
{
  "hellaswag": {
    "acc": 0.24603658536585363,
    "acc_norm": 0.34436059706446426,
  }, ... truncated ...
}
...
Thought: The `lm-eval` was executed successfully, providing the metrics I needed. The `acc` metric
↳ is 0.2460. The `acc_norm` metric is 0.3444
Action: submit
<content>
{"acc": 0.2460, "acc_norm": 0.3444}
</content>

Query: use pandas to read the file 'data.csv' and display the first 2 rows.
Thought: First, I need to read the file 'data.csv' using the pandas library.
Action: execute
<content>
import pandas as pd
pd.read_csv('data.csv').head()
</content>
Observation:
...
   name
0  Alice
1    Bob
2  Charlie
...

```

```

Thought: The file 'data.csv' has been read successfully and the first 2 rows have been displayed.
Action: submit
<content>
["Alice", "Bob"]
</content>

Query: Change the model name in the 'config.json' file from 'roberta-base' to 'roberta-large'.
Thought: First, I need to read the 'config.json' file to find the location where I can replace the
↳ model name.
Action: execute
<content>
!cat config.json
</content>
Observation:
```
{
  "model": "roberta-base",
  "batch_size": 8,
  "epochs": 10
}
```
Thought: The file has been read successfully. Now, I can replace the model name in the
↳ 'config.json' file.
Action: edit
<content>
config.json
<<<BEFORE_EDIT>>>
{
  "model": "roberta-base",
  "batch_size": 8,
<<<AFTER_EDIT>>>
{
  "model": "roberta-large",
  "batch_size": 8,
</content>
Observation: "editing file config.json"
Thought: The model name has been replaced in the 'config.json' file.
Action: submit
<content>
null
</content>

```

The demonstrations used in this prompt were not taken from any of the SUPER sets, and were designed specifically to show the basic usage of the actions that are available to the agent.

**History of Interactions.** Following the above prompt, at each step we provide the history of all past interactions by concatenating (thought, action, observation) tuples into a string, which we pass to the LLM as a single message:

```

Thought: {{thought}}
Action: {{action}}
Observation: {{observation}}

```

When executing problems from the Masked set, some steps are pre-executed. We run the pre-execute commands in the environment (without any agent interaction) to collect action and observation pairs. We then use these pairs in the history of succeeding agent steps, using the following fixed thought:

[pre-executed by the user]

Since trajectories can often get long (§4.1), we use the following truncation strategy for all agents: for the last step, we provide the 50k last characters, which is usually enough for the entire observation. For earlier steps, we shorten the observations to show the last 500 characters.

## C.2 Reflection Agent Prompt

We use the following prompt to generate reflections (Reflection agent), without any demonstrations.

### Reflection Agent Prompt

```
You will be given the history of a past experience in which you were placed in an environment and
↪ given a task to complete.
You were unsuccessful in completing the task. Do not summarize your environment, but rather think
↪ about the strategy and path you took to attempt to complete the task.
Devise a concise, new plan of action that accounts for your mistake with reference to specific
↪ actions that you should have taken.
For example, if you tried A and B but forgot C, then devise a plan to achieve C with
↪ environment-specific actions. If you wasted too much time on A, then devise a plan for more
↪ easily and directly achieving A.
You will need this later when you are solving the same task. Give your plan after "Plan" and end it
↪ with [END].
```

## C.3 SWE-Agent Prompt

We take the original SWE-Agent prompt and adjust it to instruct the agent to complete the research task in our environment rather than fixing GitHub issues, while making sure the tips and information are as similar as possible to the ReAct-SUPER prompt for fair comparison. We include the exact same three demonstrations provided in the other agents, adjusted for SWE-Agent tools, in a similar format to the original implementation.

### SWE-Agent System Prompt

```
SETTING: You are an autonomous programmer, and you're working directly in the command line with a
↪ special Jupyter notebook interface.

The special Jupyter notebook interface consists of a file editor that shows you {WINDOW} lines of
↪ a file at a time.
You can execute commands in the notebook using:

1. Bash commands: Commands starting with !.
2. Python commands: Standard Python code.
3. Magic commands: Commands starting with %, e.g., %cd <path>.

Additionally, you can also use the following commands to help you navigate and edit files.

COMMANDS:

{command_docs}

Please note that THE EDIT COMMAND REQUIRES PROPER INDENTATION.
If you'd like to add the line '    print(x)' you must fully write that out, with all those
↪ spaces before the code! Indentation is important and code that is not indented correctly will
↪ fail and require fixing before it can be run.

RESPONSE FORMAT:
Your shell prompt is formatted as follows:
(Open file: <path>)
(Current directory: <cwd>)
In [ ]

You need to format your output using two fields: discussion and command.
Your output should always include _one_ discussion and _one_ command field EXACTLY as in the
↪ following example:
DISCUSSION
First I'll start by using ls to see what files are in the current directory. Then maybe we can
↪ look at some relevant files to see what they look like.
...
!ls -a
...

You should only include a *SINGLE* command in the command section and then wait for a response
↪ from the shell before continuing with more discussion and commands. Everything you include in
↪ the DISCUSSION section will be saved for future reference.
If you'd like to issue two commands at once, PLEASE DO NOT DO THAT! Please instead first submit
↪ just the first command, and then after receiving a response you'll be able to issue the
↪ second command.
```

You're free to use any other bash commands you want (e.g. find, grep, cat, ls, cd) in addition to  
↪ the special commands listed above.

#### SWE-Agent Instance Prompt

We're currently solving the following task in the repository.

TASK:  
{query}

#### INSTRUCTIONS:

Now, you're going to execute this task on your own. You can use any bash commands or the special  
↪ interface commands to help you. Edit all the files you need to and run any checks or tests  
↪ that you want.

Remember, YOU CAN ONLY ENTER ONE COMMAND AT A TIME. You should always wait for feedback after  
↪ every command.

When you obtain the final answer for the requested TASK, you can submit it by running the  
↪ `submit` command.

NOTE ABOUT THE EDIT COMMAND: Indentation really matters! When editing a file, make sure to insert  
↪ appropriate indentation before each line!

#### IMPORTANT TIPS:

1. When a github repository is given, you should start by cloning it. You can read the readme  
↪ file and browse the installation and usage instructions. Then you need to set up the Python  
↪ environment and install necessary packages before you run any scripts in the repo. Avoid  
↪ using conda, instead use pip for package management.
2. If you run a command and it doesn't work, try running a different command. A command that did  
↪ not work once will not work the second time unless you modify it!
3. If you open a file and need to get to an area around a specific line that is not in the first  
↪ 100 lines, say line 583, don't just use the scroll\_down command multiple times. Instead, use  
↪ the goto 583 command. It's much quicker.
4. Always make sure to look at the currently open file and the current working directory (which  
↪ appears right after the currently open file). The currently open file might be in a different  
↪ directory than the working directory! Note that some commands, such as 'create', open files,  
↪ so they might change the current open file.
5. When editing files, it is easy to accidentally specify a wrong line number or to write code  
↪ with incorrect indentation. Always check the code after you issue an edit to make sure that  
↪ it reflects what you wanted to accomplish. If it didn't, issue another command to fix it.
6. Each execution has a 5-minute timeout. After this period, you can decide whether to interrupt  
↪ the kernel or continue waiting.  
You should interrupt if: (1) you are prompted with an interactive message (e.g., "Do you want to  
↪ continue? [Y/n]"). In that case, you can bypass it by executing echo "y" | command, or (2)  
↪ the execution is taking longer than expected because the wrong task is being executed, such  
↪ as a different model or incorrect number of epochs. You should continue to wait if the  
↪ current execution is correct and proceeding as expected.

(Open file: {open\_file})

(Current directory: {working\_dir})

In [ ]

## D Instructions to Experts

Workers were hired through Upwork ([upwork.com](https://www.upwork.com)), were paid \$30-\$40/hour, and were limited to four hours per task, although in some cases they were approved for up to an additional 2 hours if the task wasn't completed in time. In a few cases, the experts were unable to run the experiments due to CoLab or dependencies issues; these tasks were discarded. In total, task collection cost \$6,580 for 50 solutions, of which we keep the final set of 45 tasks.

We provide the instructions given to the Upwork experts in Figs. 4 to 6.

## E Edit command

The edit command is an action that agents can select at every iteration, in addition to execute and command. The format of the input to this command is as follows:

## Project goal

The goal of this project is for you to create Jupyter Notebooks that solve ML/NLP-related problems, which we refer to as "tasks." Each assigned task will include a description of the problem to be solved, a GitHub repository to use when solving that problem, and any additional instructions that are needed. An example task and solution (notebook) is included at the end of this document.

Please read this document in full before beginning your first task.

## Materials

You will be provided with:

- **A Google Drive folder** - this is where you should create your Jupyter Notebooks as Google [Colab](#) files. A screenshot of how to create a Google Colab notebook is included at the end of this document.
- **A Google Sheet** - this is where we will enter the descriptions for each of your assigned tasks. There is space in this spreadsheet where you are asked to enter key information for each task:
  - pip dependencies
  - current repository commit
  - your resulting answer
  - a link to your notebook
  - Task reproducibility comments (which should include any details that are necessary to reproduce your results)
  - **Notebook estimated e2e runtime**: the amount of time it takes to run the entire notebook end to end, in minutes. We generally want this to be less than 5-7 minutes. If it is more than that, please see if some hyperparameters can be changed to reduce time (e.g. less epochs, data etc.) and consult with us if necessary.
  - the amount of time you spent on each task. Ensure that you update this spreadsheet as you work through each task.

## Time considerations

Please **closely track your time spent on each task**, and indicate the time spent in your task description spreadsheet. Some tasks will have to be discarded due to taking an excessive amount of time, so please keep the following points in mind as you work through your assigned tasks:

- **Task setup**: It may be difficult to get research repositories to work properly in a Google Colab setting due to dependencies that changed or other environment issues. For this reason, we will cap the amount of time spent on dependencies setup (solving installation and versions conflict) and ask you to skip tasks where the installation time is over 1 hour. **Do not exceed 60 minutes of task setup unless we have approved you to do so.**
- **Reaching a solution**: If you have spent a total of 4 hours on a task and have not reached a solution, please pause your work on that task and let us know. We will review the work you have completed so far to determine whether or not to have you continue working towards a solution on that task. **Do not exceed 4 hours of work on a single task unless we have approved you to do so.**

Figure 4: Guidelines provided to experts.

```
{filename}
<BEFORE_EDIT>
(lines before edit)
<AFTER_EDIT>
(lines after edit)
```

Where (lines before edit) are the exact lines to be replaced, and (lines after edit) are the lines to replace them with.

To provide the exact file contents that should be replaced, agents typically need to view the existing contents of the file, for example by using the `cat` command, and then copy it verbatim as the lines to be replaced.<sup>5</sup> Our environment then looks for the contents to be replaced in the file and replaces it with the new contents.

The edit command requires the provided replaced lines to be (1) precisely copied, including correct whitespaces and indentations and (2) unique in the contents file, so that the edit command is not ambiguous. To help the agent with these requirements, we configure the edit command to provide specific feedback to the agent in case one of these conditions does not apply.

Specifically, if the the lines to be replaced were not found as-is, but these lines do appear in the edited file *without* surrounding whitespaces or tabs, then the environment provides the following feedback:

```
Did you mean to replace the following lines (notice leading/trailing whitespaces
↔ difference)?
```

<sup>5</sup>Solutions that accept line numbers instead of exact content performed worse in our early experiments.

Please note that you will be paid for your total time spent, not for the number of tasks completed, so **ensure you track your time spent**.

## Instructions

- **Your work should be submitted as a Jupyter Notebook in Google Colab.** Please create your notebook in the Google Drive folder that was shared with you, making sure that we can re-run everything to get the same result (e.g. don't delete or edit cells that need to be run for things to work.) Add the link to each task's notebook to your task description spreadsheet.
- **Different people completing the same task should get identical results.** Wherever possible, use default configurations. If you've made any decision where you think others might choose differently, please state so in the task comments. For example, if there is no default batch-size value, and you chose it to be 8, add this to the task comments in your task description spreadsheet.
- **Add pip dependencies and current repository commit to the spreadsheet.** This helps us to ensure we can reproduce your notebook in the future. Specifically:
  - To fill the dependencies column, paste the output of `!pip list` (make sure you run this after you finish the task)
  - To fill the repository commit column, paste the output of `!git rev-parse HEAD`
- **Google Colab has some environmental limitations.** For example, Conda cannot be easily installed, and RAM may be limited. Use simple means to overcome such limitations, specifically:
  - Instead of using Conda, read the Conda environment [yaml](#) file, and install required packages through pip. You can also download the [miniconda](#) installation file and install it. However, please **do not** use solutions that are specific to Colab such as [condacolab](#).
  - Insufficient RAM/resources: this can often be overcome by reducing the size of data you work on, or by reducing the batch size. If you do that, include your change in the task comments in your task description spreadsheet
  - CPU runs: please run all notebooks on CPU, not GPU.
  - Long runs: if it takes more than a few minutes to run a cell, follow the same instructions mentioned for insufficient RAM/resources.

If limitations cannot be overcome without changing the task, please consult with us.
- **Research repositories often have limitations.** Some repositories might have missing code, or unexpected issues. Do your best to overcome these, and consult with us if you do not think that would be possible. You may need to modify the code as we show next.
- **Change repository code within Colab cells.** One easy way to do this is by cloning the repository on your own machine, changing the code as you need, then creating a patch for your changes (e.g. with `git diff > changes.patch` or through your IDE). Once you have the patch, you can apply it in Colab using `%%writefile` and `git apply my_patch.patch`, as seen here:

```
[20] %%writefile my_patch.patch
Index: colbert/parameters.py
IDEA additional info:
Subsystem: com.intellij.openapi.diff.impl.patch.CharsetEP
<+>UTF-8
-----
diff --git a/colbert/parameters.py b/colbert/parameters.py
--- a/colbert/parameters.py (revision 02f3d07588af8aab2b384884fc28fb89c1e)
+++ b/colbert/parameters.py (date 1716267992256)
@@ -1,6 +1,6 @@
import torch

-DEVICE = torch.device("cuda")
+DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

SAVED_CHECKPOINTS = [32*1000, 100*1000, 150*1000, 200*1000, 300*1000, 400
SAVED_CHECKPOINTS += [10*1000, 20*1000, 30*1000, 40*1000, 50*1000, 60*1000

Overwriting my_patch.patch
```

Figure 5: Guidelines provided to experts (continued).

```
[21] !git apply my_patch.patch
```

As an alternative, you can also use bash commands such as `sed` to edit files, if you prefer.

- **Changes to the repository should be minimal.** Where possible, prefer to write code in notebook cells rather than changing the repository code. For example, if the repository code expects a csv file but you have a json file, convert the json file to a csv file in a notebook cell, rather than editing the code to accept json.
- **Follow the readme instructions wherever possible.**
- **You can use other repositories and packages if needed,** but only use well-known repositories. For example, you could use:
  - [HuggingFace datasets](#), numpy, torch, [sklearn](#), and other popular ML-related packages.
  - [gdown](#) to download dataset files from the google drive links we provide.
  - If you are unsure whether a repo/package is appropriate to use, please consult with us.
- **We should be able to run your notebook effortlessly with the "Run All" command.**
  - Please ensure that there will be no user prompts when running the commands, e.g.
 

```
files2rouge uses scripts and tools that will not be stored with the python package
where do you want to save it? [default: /root/.files2rouge/]
```

You can bypass such prompts using the pipe command (e.g. `echo "answer" | python prompt.py`)

Figure 6: Guidelines provided to experts (continued).

followed by an exact copy of these lines, including the spaces.

If more than one instances of these lines were found, the following feedback is provided:

Found multiple ([k]) occurrences of the <BEFORE\_EDIT> lines. Add 1-3 lines before  
↪ or after these lines to replace to disambiguate.

Here are the first two occurrences with additional context, did you mean one of  
↪ these?

Occurrence 1: ...

with the first two occurrences of these lines.