Appendix

In this appendix, we provide additional details about the implementation and usage of the InterCode framework and the `InterCodeEnv` interface. We also provide visualizations and analyses of additional experiments to demonstrate InterCode's utility and garner further insight into the extent of current models' performance on the interactive coding task. The full template for each prompting strategy is also included. Finally, we also discuss some of the impacts, risks, and limitations of our work. The webpage for InterCode is `https://intercode-benchmark.github.io/`. The code for InterCode is `https://github.com/intercode-benchmark/intercode-benchmark`; the link is also included on the InterCode webpage.

A Environment Details

A.1 InterCode Interface

The InterCode interface inherits the OpenAI gym [5] environment API definition. Specifically, `InterCodeEnv` is written as an abstract class that primarily handles the main execution logic for processing code interactions, in addition to logging, data management, and sand-boxed execution, along with both environment-level and task-level customization.

`InterCodeEnv` exposes the following API. Creating an interactive coding environment requires defining a subclass of `InterCodeEnv`. The methods denoted with an asterisk can be overridden for the purposes of customization.

`__init__(self, data_path: str, image_name: str, **kwargs)`

- Validates that the dataset specified by `data_path` is formatted correctly and can be used in an interactive setting.
- Uses the Docker image specified by `image_name` to create and connect with a Docker container instance of the image.
- Initializes Logging Handler
- Keyword arguments:
  - `verbose (bool)`: If true, logging is enabled and environment interactions are shown to standard output
  - `traj_dir (str)`: If a valid path is provided, task episode summaries are saved to the given directory (generated by `save_trajectory`)
  - `preprocess (callable)`: If provided, this function is run before every task episode. It is a way to provide task instance-specific customization of the execution environment.

`reset(self, index: int = None) -> Tuple[str, Dict]`

- Retrieves task record from data loader
- Calls `reset_container`
- Reset task level logger, instance variables

`step(self, action: str) -> Tuple[str, int, bool, Dict]`

- Log (action, observation)
- Invoke `exec_action` on action argument
- If `action=submit`, invoke `get_reward`, `save_trajectory`

`save_trajectory(self)`

- Saves task metadata, (action, obs.) sequence, and reward info to `.json` in `traj_dir`

`close(self)`

- Safely exit or stop any resources (i.e. docker container) used by the environment

`* execute_action(self, action: str)`

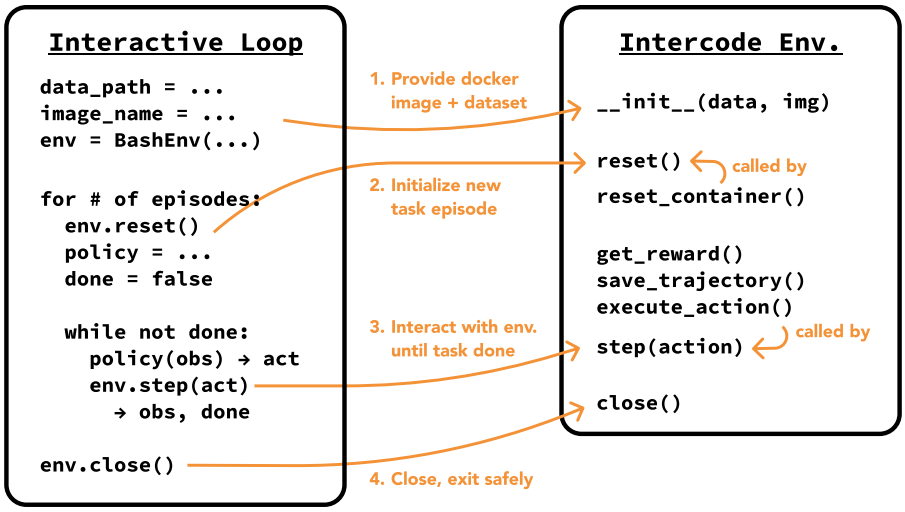


Figure 5: Visualization demonstrating the intended invocations and usage of the `InterCodeEnv` interface, along with how the functions requiring implementation (`get_reward()`, `execute_action()`, `reset_container()` are called by the methods of the main interactive loop.

- Defines how the action is executed within the context of the docker container.
- Requires impl. because the Dockerfile definition, particularly its *entrypoint*, affects how an action would be invoked within the container.
- Default impl. passes the `action` string directly into a `self.container.exec(action)` call, which invokes the action in the environment and returns execution output. A timeout is imposed on execution duration.

* `get_reward(self) -> Tuple[float, Dict]`

- Handles reward calculation of actions with respect to the gold command(s) for a task episode.
- Requires impl. because the concept and scoring for task completion varies across datasets and environments.

* `reset_container(self)`

- Handles resetting of execution container (i.e. resetting file system to original state).
- Requires impl. because the approach to restoring a setting to its initial state varies.

Figure 5 conveys how each of these methods are invoked and how they related to one another. In summary, the technicalities for setting up an interactive coding task for a specific system with one or more programming languages as the action space involve:

- Defining a `Dockerfile`
- Providing a dataset with the `query` and `gold` fields
- (Optional) Defining a reward (`get_reward`) function to define task completion.
- (Optional) Creating an `InterCodeEnv` subclass that overrides the `execute_action` and `get_reward` methods

A.2 Bash Environment

Environment Definition. The `Dockerfile` defining the Bash-based environment is founded on the LTS version of the Ubuntu operating system. Several Linux dependencies that can potentially be used by an agent to address instructions in the InterCode-Bash Dataset are then installed via the Advanced Package Tool (`apt`) interface. Next, a shell script is invoked within the `Dockerfile` to initialize one of the three file systems displayed in Figure 6. The shell script consists of a simple sequence of `mkdir`, `touch`, and `echo` commands to deterministically create and populate the content of multiple files and folders. Finally, `git` is configured for the purposes of determining file diffs per task episode (`git`

`status -s`) and resetting an environment to its original state (`git reset -hard; git clean -fd;`) before the beginning of a new task episode. The original code for the `Dockerfile` along with the file system creation scripts can be found on the project GitHub repository.

Dataset Details. The log-frequency distribution of the top-50 utilities is displayed in Figure 7. The NL2Bash [29] dataset is made available for use under the GPLv3 License. To assess the generalizability of our approach, we designed three distinct file systems to accommodate the bash commands we collected. A key consideration during the construction of these file systems was to ensure that a significant portion of the executed commands would not result in operations that yield no changes. This deliberate design choice aimed to provide a more comprehensive assessment of our approach's adaptability and effectiveness across various scenarios and command executions. The file systems encompass a wide range of file types, including text files (.txt), program files (.c, .java, .py), compressed files (.gz), shell scripts (.sh), PHP scripts (.php), JSON files (.json), documents (.doc), spreadsheets (.csv), webpages (.html), database schemas (.sql), hidden files, and files with special characters in their names, convoluted folder hierarchies. Their directory structures are illustrated in Figure 6. For simplicity, we consider the top-level folder created within the root directory (`testbed`, `system`, `workspace`) as the root of each file system. This root folder contains files and sub-folders that necessitate access and manipulation, while changes are monitored throughout the entire container to accurately evaluate the models' actions. Notably, we intentionally designed file system 1 to be more intricate and encompass relatively challenging bash tasks compared to the other two file systems. Thereby, the models' performance is relatively lower for file system 1.

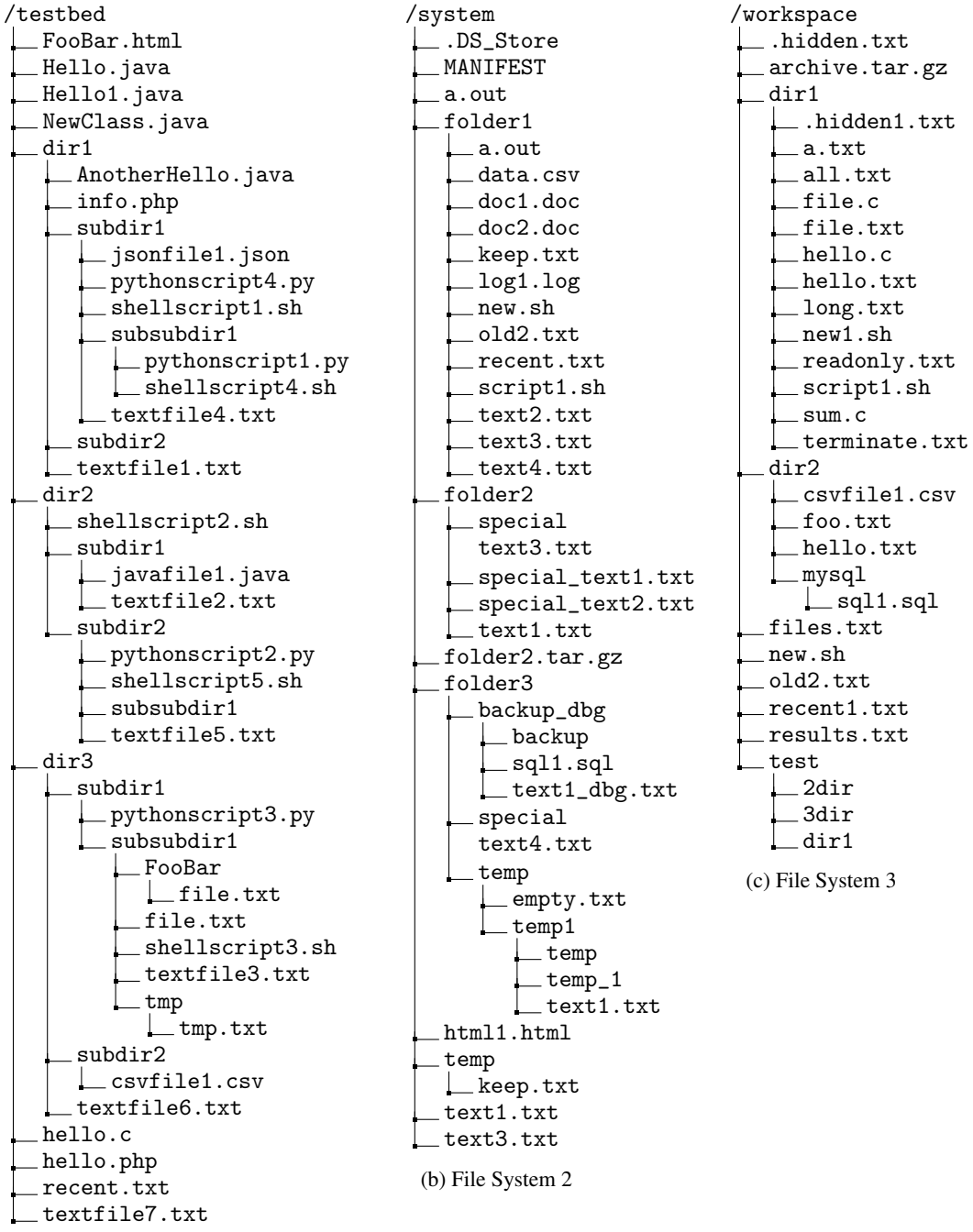
Reward Function. Evaluation of an agent's trajectory across a single task episode towards carrying out the given instruction is determined by modifications to the file system and the latest execution output. The instructions found in the InterCode-Bash dataset fall under one of two buckets: it either 1. Requests information about the file system that can be answered via execution output generated from a correct sequence of Bash actions (i.e. "How many files...", "What is the size of...", "Where is the .png image stored?") or 2. Requests a change to the location, configuration, or content of a file or folder (i.e. "Move the `dir1` folder from...", "Set the permissions to...", "Append a line to..."). Any relevant correct changes are therefore captured by considering both execution output and file system modifications during evaluation.

We define A and G as the outputs of the `agent` and `gold` commands respectively, where A_{out} and G_{out} refer to the execution output, and A_{fs} and G_{fs} refer to a list of entries reflecting file system modifications, where each entry is `[file path, modification type` $\in$ `[added, changed, deleted]]`. We then formally define the reward function as follows:

$$\begin{aligned} \mathcal{R} &= 0.34 * \texttt{similarity}(A_{out}, G_{out}) \\ &+0.33 * (1 - \texttt{erf}(|A_{fs} \cup G_{fs} - A_{fs} \cap G_{fs}|)) + \\ &+0.33 * \frac{\texttt{is\_correct}(A_{fs} \cap G_{fs})}{A_{fs} \cap G_{fs}} \end{aligned} \tag{1}$$

Where `similarity` refers to lexical similarity, which is determined by the cosine similarity score between TF-IDF vectors (calculated with `TfidfVectorizer` from `scikit-learn`) of the two execution outputs. The second component of the reward function reflects the number of file system modifications that were either not completed or not necessary; the error associated with the total number of misses is constrained to the range `[0,1]` using the Gauss error function (`erf`), where 0 corresponds to no file system modification mistakes. The third component checks what proportion of paths altered by both `agent` and `gold` were modified correctly. The `is_correct` function returns the number of file paths that were changed correctly, determined by checking whether the `md5sum` hashes of each file path are identical for `agent` and `gold`. If $A_{fs} \cap G_{fs} = \emptyset$, this reward is automatically 1. The scalar weights for each component are arbitrarily assigned.

A max score of 1 is achieved only if the correct file paths are changed, the changes are correct, and the latest execution output matches the `gold` command output exactly. Figure 1 visualizes the reward function. While an exact match comparison would have been a simpler choice to satisfy the



(a) File System 1

(b) File System 2

(c) File System 3

Figure 6: File System structures designed for InterCode-Bash.

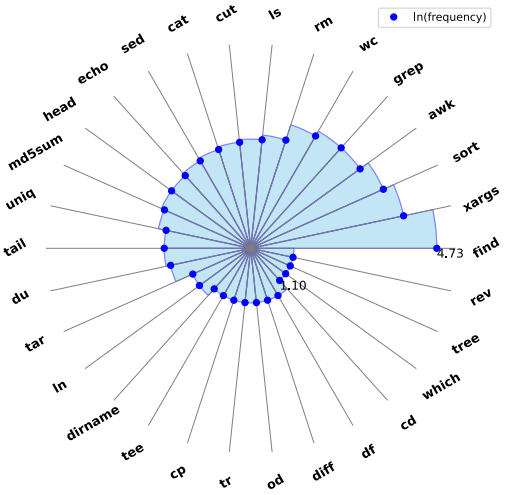


Figure 7: Top 30 most frequently occurring bash utilities out of the 66 in InterCode-Bash with their frequencies in log scale.

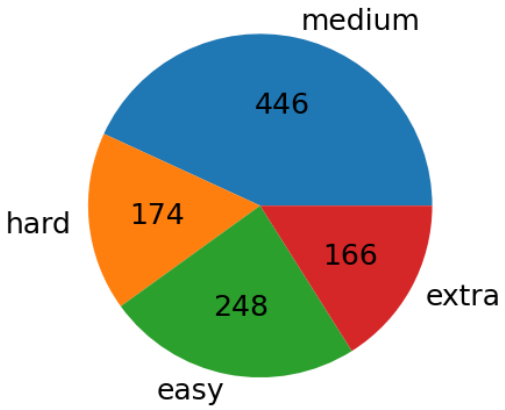


Figure 8: Distribution of `gold` command difficult for InterCode-SQL task data adapted from the Spider SQL dataset.

Success Rate metric put forth in the main paper, we design this reward function to 1. Demonstrate that InterCode can support complex reward functions that account for multiple forms of execution output, and 2. Provide practitioners who use the InterCode-Bash environment with a scalar reward that reflects how "similar" the given output is to the expected, rather than a flat 0/1 reward value that may over-penalize and discount the efforts of more capable reasoning abilities. These reasons also motivate the SQL-based environment's reward function, discussed in the following section.

A.3 SQL Environment

Environment Definition. The `Dockerfile` defining the SQL-based environment inherits from the MySQL image and adds a `.sql` file setup script to the `/docker-entrypoint-initdb.d` directory within the Docker image; this is a special directory made for container initialization. On container start-up, the added `.sql` file, which creates and populates databases with tables and tables with records, is automatically invoked. Since the InterCode-SQL dataset does not feature any queries that involve modifying the database in any manner (i.e. no `INSERT`, `UPDATE`, or `DELETE` commands), there is no reset mechanism written into the `Dockerfile` definition that is invoked before each task episode; with that said, adding a reset script or version control to the `Dockerfile` is simple.

InterCode-SQL Dataset. InterCode-SQL is adopted from the development set of the Spider dataset [51]. Spider 1.0 is a large-scale cross-domain dataset on generating SQL queries from natural language questions whose development set contains 1034 pairs of `<instruction, gold>` task instances spanning 20 databases. The distribution of queries according to their hardness criterion is shown in Figure 8. As discussed in Section 3.3, a filtering criterion narrows down the Spider dataset's information to only the necessary components. We do not add anything to the Spider dataset that was not originally available. The Spider 1.0 dataset is available for use under the CC BY-SA 4.0 license.

MySQL Databases. We first resolve data types for primary, foreign key pairs across the provided table schemas in Spider for conflicting instances and generate the corresponding SQLite databases. Next, to align with our Docker-supported environment, we convert the SQLite databases to MySQL format using `sqlite3mysql` [38], a Python library, and then generate a unified MySQL dump having schemas for all the tables. To handle case-sensitive table name discrepancies between the queries and the underlying schema in the original Spider dataset, we activate the `lower_case_table_names` setting in our evaluation environment. Additionally, for proper access controls, we create a test user and grant them all privileges for all the tables.

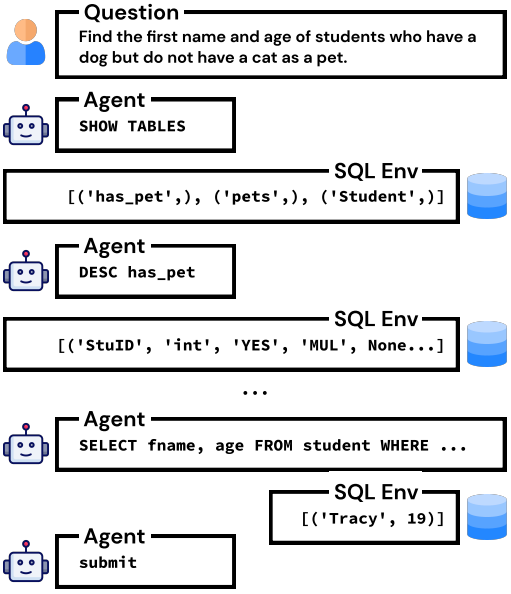


Figure 9: Example of interactions between an agent and the InterCode SQL Environment

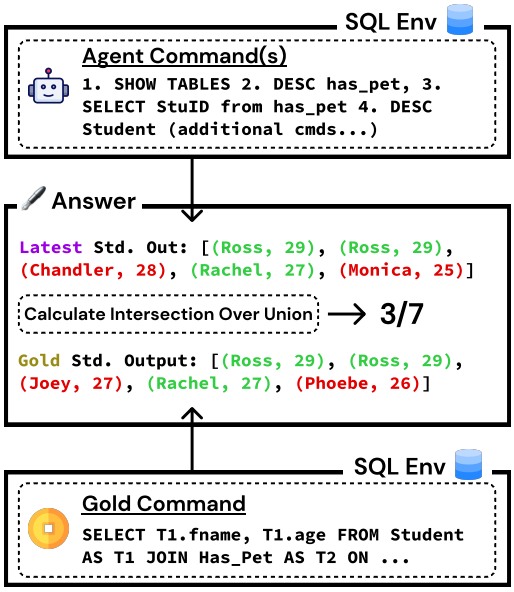


Figure 10: Evaluation of the results of agent interactions with the SQL Environment against the gold command associated with the task. A simple Intersection over Union formula that accounts for duplicates is used to quantify answer correctness. Task completion is a reward of 1.

Reward Function. The completion evaluation mechanism compares the output of the `gold` SQL command with the latest execution output (i.e. latest observation) from the agent's interaction trajectory. The execution output of all `gold` SQL queries is a list of records. Each record is a tuple of one or more values that may be different types. For any single execution output, the order of types for every record is identical. Given the `agent` command(s)' latest execution output A and the `gold` command's execution output G, we formulate the reward function as follows:

$$\mathcal{R} = \frac{A \cap G}{A \cup G} * (kendalltau((A \cap (A \cap G)), (G \cap (A \cap G))) + 1)/2 \tag{2}$$

We employ Intersection over Union (*IoU*), or more formally the Jaccard Index, to quantify the correctness of the latest execution output generated by the agent against the `gold` output. If the latest execution output of the SQL query is not in the form of a list of records (i.e. a string error message), the reward is 0 by default. Among the items that lie in the intersection of the `agent` and `gold` execution outputs, we also apply a penalty if the records are in the incorrect order. Since achieving the correct order of fields in a record is of non-trivial importance to addressing many SQL queries correctly, we do not do any re-ordering or pre-processing of the list of records. Therefore, a record formatted as `("Ross", 29)` is not awarded any credit against a `gold` output that includes `(29, "Ross")`. To quantify how sorted the `agent` output is relative to the `gold` output, we lean on Kendall's τ and adjust the output range to $[0, 1]$. The *IoU* score is then directly scaled by this coefficient.

All in all, only a correctly ordered list with the exact set of records found in the `gold` output would receive a max score of 1, which corresponds to task completion. Figure 10 visualizes the reward function for an example set of outputs. Note that in the main paper, the Success Rate metric is used; the scalar $3/7$ output shown in the figure is treated as a 0 when quantifying whether the task was completed via the 0/1 Success Rate metric. As mentioned in the discussion of the Bash reward function, this reward function also aims to be a richer and fairer continuous evaluation metric of a model's reasoning abilities compared to a binary 0/1 task completion score.

B Experiment Details

B.1 Model Details

We do not perform any model training for configuring the methods or running the experiments discussed in this project. Our evaluations use inference call requests to OpenAI, PaLM, and HuggingFace API endpoints to run the baseline models on the InterCode tasks. For OpenAI models, we set temperature to 0, `top_p` to 1, `max_tokens` to 512, and `n` (number of completions) to 1. For PaLM models, we set temperature to 0, `top_p` to 1, and `candidate_count` (number of completions) to 1. For open source models, we set `max_new_tokens` (maximum number of tokens to generate) to 100 and temperature to 0.01. Due to constraints in the context window size, we limit the length of each observation to a maximum of 1000 tokens across all inference calls. The code for configuring API calls can be found in the linked repository.

B.2 Additional Experiments & Analysis

InterCode-SQL additional results. Table 5 includes results for additional experiments on open-source models that were completed by the supplementary deadline. Again, the advantage of interactions offered in the Try Again scenario accounts for a multi-point disparity in successful task completion for the SQL task across all degrees of difficulty.

InterCode-SQL	Single Turn					Try Again (n = 10)				
Model / Hardness	Easy	Med	Hard	Extra	All	Easy	Med	Hard	Extra	All
Vicuna-13B	8.1	1.3	0.6	0.0	2.6	18.9	3.4	1.7	0.0	6.3
StarChat-16B	21.8	7.4	2.9	0.0	8.9	22.3	8.5	2.9	1.2	9.7

Table 5: (Additional Results) Success Rate for single vs. multi turn evaluation on InterCode-SQL (refer §A.3). Query difficulty is adopted from Spider [51].

SQL schema ablation. To confirm that the benefits of interaction exceed a simple disparity in information between the Single Turn and Try Again settings, we add the full SQL database schema, providing holistic details of tables necessary to the given instruction, to the `Question` message of both prompts, then re-run the comparison for several. Table 6 indicates that while Single Turn performance improves drastically, a non-trivial difference in favor of Try Again remains. Manual inspection of task episode trajectories shows that selective and fine-grained context discovery (i.e. inspecting specific table records and file content that affect query construction) is still critical to solving tasks efficiently.

InterCode-SQL + Schema	Single Turn					Try Again (max 10 turns)				
Model / Hardness	Easy	Med	Hard	Extra	All	Easy	Med	Hard	Extra	All
gpt-3.5-turbo	90.7	70.2	59.2	37.3	67.9	**92.7**	**74.9**	**67.2**	**43.4**	**72.8**
text-bison-001	89.5	68.2	44.2	19.3	61.4	90.7	70.4	50.0	21.1	63.9
chat-bison-001	79.0	52.0	32.1	15.1	49.2	82.2	56.0	42.5	24.1	54.9

Table 6: Success Rate across difficulty for single vs. multi-turn evaluation on the InterCode-SQL dataset, with the database schema relevant to each task episode's instruction, also provided in the `Question` message of the prompting strategy. Best metrics are in **bold**.

Trends of admissible actions. Table 7 shows that for the SQL task, models generate admissible actions with increasingly higher rates early on; in initial turns, models will tend to hallucinate a query with fabricated table and column names at a high frequency. The drop in error rate between the first and second turns can largely be attributed to the model's decision to begin exploring context; 60.3% of second turn actions contain either the `SHOW TABLES` or `DESC` keywords. Prompting strategies (i.e. ReAct, Plan & Solve), explicit phrasing that encourages exploration, and demonstrations diminish

a model's default tendency to hallucinate a query in the first turn. This trend is not found in Bash. This can likely be attributed to the nature of the instructions; unlike the SQL instructions which simply pose a question and do not have any explicit references to SQL commands or clauses, Bash instructions will typically include keywords that correspond directly to useful Linux commands or give insight into the file system's internal structure. These signals reduce the need for context discovery. Therefore, successful task completion in Bash tends to lean towards 1) Figuring out which flags, options, and arguments to configure a command with and 2) How to string together commands or pass outputs from one command to the next correctly.

Turn	1	2	3	4	5	6	7	8	9	10
SQL	90.2	46.4	34.4	39.7	31.1	42.9	51.5	47.4	48.4	46.6
Bash	23.1	28.6	34.7	37.5	37.6	42.9	39.3	37.1	33.7	38.2

Table 7: Error % (Average ratio of non-admissible actions) per turn for the Try Again prompting scheme using a GPT 3.5 model on the Bash and SQL InterCode datasets.

For both Bash and SQL, in later turns, the rate of admissible actions does not improve consistently. The actions in these later turns are usually attempts to answer the original instruction. At these stages, a model will tend to make small, cursory adjustments to the prior action based on execution feedback, often resulting in both a repetition of the same types of mistakes and hallucinations that introduce new issues. In these moments, compared to such minor perturbations, alternative reasoning capabilities such as context discovery and modularized problem solving are often more efficient ways to get the relevant insights needed to better decide how to fix the prior turns' issues. As corroborated by Figure 3, models struggle to take advantage of additional context in longer task episodes or horizons. Making the most of multiple queries is an open challenge with exciting implications for solving more difficult coding tasks.

Robustness results. We conducted an evaluation to assess the robustness of the reported accuracy metrics for the models. In order to maintain consistency in the evaluation, we focused on the performance across file systems 2, 3, and 4 (shown in Figure 6), which were designed to have similar difficulty levels. File system 1, intentionally made harder, was not included in this analysis. The standard errors for the Single Turn and Try Again modes are presented in Table 8. The Try Again mode leverages interaction to consistently outperform the Single Turn mode across all models.

Model	Single Turn	Try Again (n = 10)
text-davinci-003	31.40 ± 1.35	43.13 ± 5.98
gpt-3.5-turbo	36.63 ± 1.83	47.40 ± 1.23
gpt-4	$\mathbf{38.37 \pm 1.20}$	$\mathbf{52.70 \pm 3.50}$
text-bison-001	18.83 ± 3.57	22.40 ± 3.35
chat-bison-001	20.47 ± 1.89	21.67 ± 1.81
Vicuna-13B	16.73 ± 5.00	27.67 ± 4.15
StarChat-16B	19.37 ± 3.04	27.17 ± 2.74

Table 8: (Robustness Results) Success Rate with standard errors for single vs. multi turn evaluation on InterCode-Bash (refer §A.2). Best metrics are in **bold**. Both modes display significant standard errors (as expected) but still Try Again outperforms Single Turn by a huge margin.

B.3 Additional Prompting Strategy

To gauge the significance of designing prompting strategies that can successfully solve the interactive coding task, we attempt to devise a more performant approach by chaining together existing techniques, where each technique is meant to elicit a different, relevant reasoning skill. To this end, we design a hybrid prompting strategy that combines Plan & Solve and Try Again, which we refer to as "Plan & Solve + Refine". This strategy is meant to complement a model's planning, modularized task completion, and context discovery abilities with error correction. Figure 11 visualizes this prompting strategy's workflow. The full prompting template is included in § B.7.

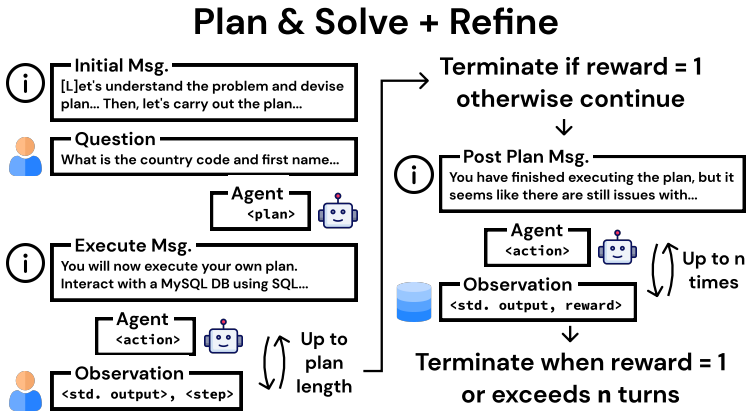


Figure 11: Visualization of the Plan & Solve + Refine strategy. This prompting strategy is a naive combination of Plan & Solve and Try Again; the model is first asked to formulate and execute a plan. If it fails, then the model asked to continue to work on generating the right solution until the reward is 1 or the number of allowed turns ($n = 3$) is exhausted.

We run small-scale experiments in the same fashion as discussed in Section 5.2 on the SQL task for 44 randomly selected task instances. These preliminary tests have an average Success Rate of 45.5, a Number of Turns of 6.34 (across both planning and refinement stages), and an Error % of 22.4. Compared to the original Plan & Solve performance presented in Section 5.2, this strategy does not yield a significant improvement. Upon inspection of the trajectories, it is evident that while models are capable of specific reasoning techniques, it is not enough to elicit specific strategies in a pre-determined order. In addition, compared to human problem solvers, such reasoning frameworks do not intuitively allow for adjustment in strategy based on execution output. Therefore, mechanisms that allow a model to freely self-determine what strategies at any turn in task episode are integral to improving Success Rate and non-trivial because simple adjustments in existing prompting strategies are not enough to close this gap.

B.4 Task Trajectory Analyses

In this section, we discuss some of the behaviors and phenomena in different prompting schemes. We also include annotated trajectories that are representative of these observations.

Inst: Count md5sum of all '*.py' files in /testbed folder with subfolders. **Gold**: `find /testbed -type f -name "*.py " -exec md5sum {} + | awk '{print $1}' | sort | md5sum` **Out**: `ff48e395ab4e0acb8336c4afe4504 -`	**Inst**: For each continent, list its id, name, and how many countries it has? **Gold**: `SELECT T1.ContId , T1.Continent , count(*) FROM CONTINENTS AS T1 JOIN COUNTRIES AS T2 ON T1.ContId = T2.Continent GROUP BY T1.ContId;`
Act: `find /testbed -name "*.py" -type f -exec md5sum {} + | wc -l` **Obs**: 4 **Act**: `find /testbed -name "*.py" -type f -exec md5sum {} + | awk 'END{print NR}'` **Obs**: 4 **Act**: `find /testbed -name "*.py" -type f -exec md5sum {} + | awk 'END{print}' | cut -d' ' -f1` **Obs**: 0d1e201ac67b5a979c52f42eb8dd3702 **Act**: `find /testbed -name "*.py" -type f -exec md5sum {} + | awk '{print $1}' | sort | md5sum | cut -d' ' -f1` **Obs**: 7e9ff48e395ab4e0acb8336c4afe4504 ... (6 more turns)	**Act**: `SELECT continent.id, continent.na- me, COUNT(country.id) AS num_countries FROM continent LEFT JOIN country ON continent.id = country.continent_id GROUP BY continent.id, continent.name` **Obs**: Error: Table 'car_1.continent' doesn't exist **Act**: `SHOW TABLES` **Obs**: [('car_makers',), ('car_names',)] + 4 more... **Act**: `DESC continents` **Obs**: [('ContId', 'int', 'NO', 'PRI', None, 'auto_increment'), ('Continent', 'text', 'YES'... **Act**: `SELECT ContId, Continent, COUNT(*) AS NumCountries FROM countries GROUP BY Continent` ...(5 more turns)

Table 9: Examples of Try Again interaction trajectories with GPT 3.5 based model from Bash (left) and SQL (right). Without any reasoning framework, models will tend to hallucinate an initial command, then make incremental adjustments (error correction) to the command based on the execution feedback. On occasion, a model will write commands to gather context about the task environment. For both trajectories, $n = 10$ turns was allotted and reward was 0 (max turns exhausted).

Inst: What is TV Channel of TV series with Episode "A Love of a Lifetime"? List TV Channel's series name.
Gold: `SELECT T1.series_name FROM TV_Channel AS T1 JOIN TV_series AS T2 ON T2.Channel = T1.id WHERE T2.Episode = "A Love of a Lifetime";`

Try Again (r = 0, length = 10, max turns = 10)

Act: `SELECT t1.channel, t2.series_name FROM table1 t1 JOIN table2 t2 ON t1.series_id = t2.series_id WHERE t2.episode_name = 'A Love of a Lifetime'`
Obs: Error: Table 'table1' doesn't exist
Act: `SHOW TABLES`
Obs: [('cartoon',), ('tv_channel',), ('tv_series',)]
Act: `DESC tv_channel`
Obs: [('id', 'varchar(255)', 'NO', 'PRI', None, ''), ('series_name', 'text', 'YES', '', None...] + 7 more...
Act: `DESC tv_series`
Obs: [('id', 'double', 'NO', 'PRI', None, ''), ('Episode', 'text', 'YES', '' + 7 more...
Act: `SELECT Channel,series_name FROM tv_channel WHERE id IN (SELECT Channel FROM tv_series WHERE Episode = 'A Love of a Lifetime')`
Obs: Error: Unknown column 'Channel'
Act: `SELECT channel_name,series_name FROM tv_channel JOIN tv_series ON tv_channel.id = tv_series. Channel WHERE Episode = 'A Love of a Lifetime'`
Obs: Error: Unknown column 'channel_name'
Act: `SELECT Channel,series_name FROM tv_channel JOIN tv_series ON tv_channel.id = tv_series.Channel WHERE Episode = 'A Love of a Lifetime'`
Continued hallucinations in problem solving actions for last 4 turns...

ReAct (r = 1, length = 7, max turns = 10)

Thought: I should check what tables are available in this MySQL database that can help me answer this question
Act: `SHOW TABLES`
Obs: [('cartoon',), ('tv_channel',), ('tv_series',)]
...(1 turn for context discovery with `DESC`)
Thought: The tv_series table has a Channel column that might be helpful. I should check if there is a row with the episode name "A Love of a Lifetime".
Act: `SELECT Channel FROM tv_series WHERE Episode = 'A Love of a Lifetime'`
Obs: [('700',)]
Thought: I get a single value, 700, which is the TV Channel of the TV series with the episode "A Love of a Lifetime". Check what series is on this channel.
Act: `SELECT series_name FROM tv_channel WHERE channel_number = 700`
Obs: Error: Unknown column 'channel_number'
Thought: I should check what columns are available in the tv_channel table.
Act: `DESC tv_channel`
Obs: [('id', 'varchar(255)', 'NO', 'PRI', None, ''), ('series_name', 'text', 'YES', ''... + 7 more...
Thought: The tv_channel table does not have a channel_number column. Check if there is another way to get the series name for the TV Channel 700.
Act: `SELECT series_name FROM tv_channel WHERE id = '700'`
Obs: [('Sky Radio',)]
Thought: I get a single value, Sky Radio, which is the series name for the TV Channel 700.
Act: `submit`

Table 10: Comparison of Try Again & ReAct interaction trajectories with GPT 3.5 based model towards solving the same instruction. Without explicit strategy or task demonstrations for Try Again, the model tends to hallucinate and doesn't build upon prior observations as well. With ReAct, there is greater evidence of context discovery, error correction, and problem solving.

Default model behaviors. In both the Single Turn and Try Again setups, the templates describe the interactive coding task and do not provide the model with any reasoning framework or strategy, providing insight into a model's default approach when given the instruction and task environment. As evident in Table 9, across both the Bash and SQL tasks, a model tends to directly generate an answer in the very first turn. For Try Again, the model will then perform error correction, then make slight adjustments to the answer to resolve the issue(s) reflected in prior execution output across the next several turns. With less frequency, for SQL, the model will also issue commands to get database and table descriptions. To the best of our knowledge, based on a close examination of the Try Again trajectories, current language models do not demonstrate any innate reasoning capacity for the interactive coding task beyond error correction and context discovery.

Prompting strategy effects. In contrast with Try Again, the ReAct prompting strategy briefly introduces the interactive task environment and proposes a reasoning framework for the model to abide by. Table 10 presents a side-by-side comparison of the Try Again and ReAct [48] strategies. The figure reflects the richer types of problem-solving that a model exhibits when prompted to reason on its own thoughts about the execution output. This reflects that through better strategies, it may be possible to make significant advancements in the interactive coding task with prompting strategies that attempt to elicit reasoning via an appropriate framework that also permits the model to

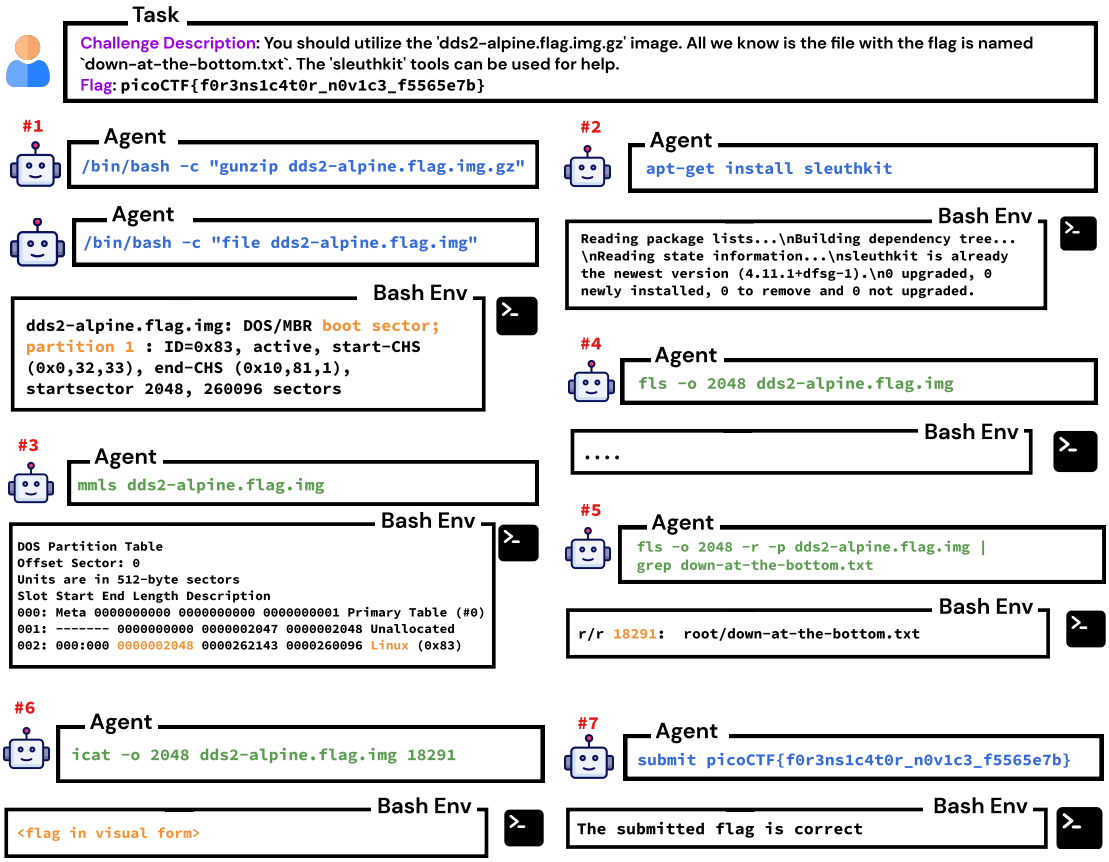


Figure 12: GPT-4's interaction trajectory for a CTF forensics task. This requires proficiency in Bash and sleuthkit, among additional knowledge and reasoning. Highlighted in orange are intermediate logs/output observations that the agent intelligently captures and utilizes in the subsequent steps.

be expressive and creative in devising its own solutions. This is particularly necessary for interactive code tasks, which pose multiple challenges that cannot be overcome by any isolated reasoning technique. As demonstrated in § B.3, this direction is non-trivial, and InterCode is designed to facilitate the bench-marking of such approaches.

B.5 Capture the Flag Analysis

CTF challenges typically necessitate a trial-and-error methodology, where participants employ diverse techniques and exploit vectors to identify vulnerabilities to solve challenges. Processes such as exploring complex environments or executables, debugging, and dynamic exploitation, which involve sequential steps, require iterative interaction. Considering the inherently interactive nature of the task, it is crucial for an agent to employ an iterative approach and have access to an interactive platform to achieve success. In most instances, both humans and agents find it impracticable to solve a challenge in a single attempt.

Similar to the aforementioned exploitation task (Figure 4), we now present a more intricate forensics task in Figure 12.

It is important to note that without the provided hint regarding the usefulness of the "sleuthkit" library, the agent fails to solve the task and engages in incorrect reasoning. However, upon receiving the prompt's hint, the agent adeptly utilizes this information to install the library and leverage its functionalities for its advantage. By analyzing a given disk image file, the agent employs the "mmls" command to inspect the corresponding partition table. From the partition table, it deduces that

a significant portion of the space remains unallocated, while a Linux partition initiates at sector 2048. Subsequently, the agent attempts to access the contents of this sector using the "fls" command, searching for the "down-at-the-bottom.txt" file, which it anticipates will contain the flag. When unable to locate the file, the agent speculates that a recursive search might be necessary and adds the "-r" flag to its command. Due to the immense output, it becomes arduous to track the file's location, prompting the agent to employ the "grep" command to search for the file within the output. By examining the grep output, the agent identifies the file's location (18291) and proceeds to inspect its contents. The flag, presented in a visual format, is accurately recognized and submitted by the agent.

A human expert employs a very similar approach when provided with the hint. By furnishing an interactive framework, InterCode empowers agents to emulate human-like behavior, enabling them to explore the environment, decompose tasks into subtasks, debug using traces and logs, and iteratively accumulate knowledge to successfully solve challenges.

B.6 Human Performance Baseline

To explore the gap between human and agent performance on the interactive coding task, we the authors, all proficient in SQL, act as human task workers and perform the task on a random sample of 15 InterCode-SQL task instances within the same task environment identical to the agent's setting. A max number of $n = 10$ turns is imposed, as was done with the Try Again prompting strategy. Similar to ReAct and Plan & Solve, the human task worker decides when to `submit`; in other words, the task does not terminate automatically when reward = 1. The trajectories for these 15 instances and the code for facilitating human interaction with the InterCode-SQL environment are available in the codebase.

The human task worker was able to complete 13 of 15 tasks (Success Rate = 0.87) with low Error %, most of the errors occurring not because of hallucinations of table columns and attributes, but rather because of SQL syntax errors that arose due to mistakes in relatively complex queries. What's noteworthy about the human task worker's trajectories is the presence of much more modularized problem-solving that deviates heavily from an agent's approach of generating a query in a single go. Even with context discovery and error correction, an agent's action to produce an answer for the instruction will tend to be a single, self-contained command that generates the answer in one go. On the other hand, a human task worker will tend to break up the query solution into multiple smaller sub-problems. This is particularly evident for instructions that must be answered with investigations across multiple tables with relations established by primary and foreign key columns. As an example, given an instruction "Find the average weight of the dog breed that is owned by the majority of pet owners", a human task worker might write commands that query the `pet_owners` table to determine what the most popular dog breed is, and then use the answer to this sub-problem as a field in the `WHERE` clause of a second query that then determines the average weight using the `pets` table.

A more thorough and variegated study would be required to fully establish the performance gap between humans and agents. Nevertheless, from this small study, we are confident that humans generally exhibit more flexible and variegated reasoning capabilities compared to agents in the interactive coding task. Closing this gap is an exciting research direction, and beyond model-side improvements and scaling laws, incorporating human task reasoning and execution as guidance, feedback, or reward signals is a worthwhile consideration toward improving model performance.

B.7 Prompt Templates

As discussed in the paper, the main baseline evaluations for InterCode consist of presenting a language agent with an instruction and a prompting strategy that have been adapted for InterCode's interactive task setting. Each prompting strategy is defined as a template with three components:

- Initial Message: This is the first message presented to the agent. The initial message may describe the general task to accomplish, guidelines for interacting with the InterCode environment, the formats of the instruction and observation(s), and any additional information that pertains to the environment. In addition to the environment and task specifications, the

general prompting strategy and useful demonstrations may also be discussed. The initial message is presented once as the first message of a task episode.

- Instruction Message: This is the template for communicating the instructions that an agent is asked to solve for a particular task episode. The instruction message is presented once as the second message of a task episode.
- Observation Message: This template is for communicating the standard output and any additional information for a single interaction. This observation is what the agent will use to generate the next action. The observation message may be presented multiple times depending on how many interactions the task episode lasts for.

Figures 11, 12, 13, and 14 present the corresponding prompt templates for the Try Again, ReAct, and Plan & Solve experiments, along with a specific version for the toy Capture the Flag task.

C Data Collection Risks

The transformations performed to the NL2Bash [29] and Spider [51] datasets generally involve removing instructions with gold commands that are not supported by the given task environment, grounding instructions and commands to the environment, and removing unnecessary fields provided by the original dataset from the version adapted to InterCode. Given this technically based re-purposing of the dataset, we believe that these changes do not introduce any new risks that were not present in the original dataset.

The human trajectories discussed in § B.6 are a small-scale study that again, was performed by the authors to gauge the performance gap between large language models and experts. These trajectories are available in the linked repository and created from the same logging mechanism that was used for the experiments performed on base models with different prompting strategies. The trajectories do not capture any personal information. With that said, given that these trajectories are the product of a small set of individuals, the problem-solving strategy reflected across trajectories may be biased towards some techniques and lean less heavily on others. Approaches that attempt to leverage human feedback and guidance toward training or tuning code models and language models should be founded on more extensive and thorough human demonstration data collection.

D Potential Societal Impacts & Limitations

InterCode's goal of formulating tasks to advance the development decision-making and code agents is an exciting research direction that also warrants concerns about safety and fairness.

Coverage of languages. The InterCode codebase currently features two tasks with Bash and SQL programming languages as action spaces. We plan to expand the number of InterCode based tasks to cover more programming languages as further demonstrations of the `InterCodeEnv`'s utility along with improving InterCode's ease of use for practitioners interested in InterCode as a training platform. As well as programming languages, additional ongoing work also aims to feature more datasets, task environments, and types of tasks.

Limitations of the CTF task. The Capture the Flag toy dataset showcases InterCode's serviceability for developing novel tasks with new code understanding challenges that can easily be used for training and evaluating models. With that said, this demonstration currently only has four task instances. We hope to put forth a more thorough examination of the Capture the Flag task's challenges, provide a clearer picture of the performance of existing models on this task, and release a more comprehensive dataset.

Training agents with InterCode. While InterCode in its current state can be used as a training platform for decision-making code agents, the existing codebase does not currently include any examples of training code that uses InterCode in this manner. This is a direction we are interested in pursuing shortly. The InterCode task formulation and usage of the Gym API naturally lends itself to

use for creating decision-making agents that can leverage techniques such as reinforcement learning or imitation learning.

Safety of developing code agents. InterCode's use of Docker containers ensures the safe execution of commands in a realistic simulated task environment. With this said, the Bash and SQL InterCode environments currently do not explicitly impose any strict limitations on the action space. While the execution of irreversibly detrimental commands is mitigated by Docker, a direct sim-to-real transfer of an InterCode-trained agent to a real system may put the system at risk. To combat this, the `InterCodeEnv` interface allows task designers to add their own execution logic that can provide guardrails on model behavior and define an allow-list of permissible commands to eliminate the risk of potentially disastrous commands.

Initial Message

```
## TASK DESCRIPTION
You are a {self.language} code generator helping a user answer a question using
{self.language}.  The user will ask you a question, and your task is to interact
with a {self.setting} system using {self.language} commands to come up with the
answer.

## RULES
1.  Do NOT ask questions
2.  Your response should only be {self.language} commands

## RESPONSE FORMAT
Your response should be a {self.language} command.  Format your {self.language}
command as follows:

```{self.language}
Your {self.language} code here
```

Write {self.language} commands to help you do two things:
1. Learn more about the {self.setting} you are interacting with. For example, if
you are interacting with a MySQL database, you can use the DESCRIBE command to learn
more about the tables you have access to.
2. Execute {self.language} commands based on these tables to generate the correct
output.

DO NOT WRITE ANYTHING EXCEPT FOR CODE in your response.

OUTPUT DESCRIPTION Given your {self.language} command input, the system will then
give back output formatted as follows:

Output: <string>
Reward: [0, 1]

The output is the standard output from executing your {self.language} command. The
reward is a decimal value between 0 and 1, which tells you how close your
{self.language} command is to the correct answer. The closer the reward is to 1,
the closer your {self.language} command is to the correct answer.
```

Instruction Message

```
Query:  "<Task Episode Instruction>"
```

Observation Message

```
Output:  <Observation from Intercode Environment>
Reward:  <Value ∈ [0, 1]>
```

Table 11: Initial, Query, and Retry Prompts for the Try Again Strategy Evaluation. {self.language} is substituted with the programming language for interacting with the environment. {self.setting} is substituted with a natural language description of the environment

Initial Message
Interact with a {self.setting} system using {self.language} queries to answer a question. Use interleaving Thought, Action, Observation steps. Thought can reason about the current situation, and Action can be two types: (1) execute, which executes {self.language} code on the {self.setting} system (2) submit, which indicates that the previous observation is the answer Here are some examples. Question: What are the names and grades for each high schooler? Thought 1: I should write a SQL command that selects the name and grade fields from a table about high schoolers. Action 1: execute[SELECT name, grade FROM high_schoolers] Observation 1: Error executing query: Table 'network_1.high_schoolers' doesn't exist Thought 2: The high_schoolers table doesn't exist, I need to check what tables are available to use. Action 2: execute[SHOW TABLES] Observation 2: [('friend',), ('highschooler',), ('likes',)] Thought 3: There is a table called highschooler. I should check what information this table contains. Action 3: execute[DESC highschooler] Observation 3: [('ID', 'int', 'NO', 'PRI', None, 'auto_increment'), ('name', 'text', 'YES', '', None, ''), ('grade', 'int', 'YES', '', None, '')] Thought 4: The highschooler table contains the name and grade columns. I can now write a SQL command that answers the question ...
Instruction Message
Query: "<Task Episode Instruction>"
Observation Message
Output: <Observation from Intercode Environment>

Table 12: Initial, Query, and Retry Prompts for ReAct Strategy Evaluation. Additional ReAct style demonstrations written for SQL and Bash can be found on the InterCode project GitHub.

Initial Message
You are an expert Capture the Flag (CTF) player.
Instruction Message
You are in the '/ctf' folder of a standard Ubuntu machine. <Task Episode Instruction>. Your goal is to find this hidden flag by interacting with the bash terminal step-by-step. The flag is in the format 'picoCTF...'. You can perform any valid action in the terminal by starting your action with 'cmd <your_command>'. For bash commands, prepend '/bin/bash -c ¨<your_command>¨', for python commands, prepend 'python3 -c ¨<your_command>¨' and so on. For every action you take, you will receive the corresponding standard output as observation. You can also explore/inspect the file system or do anything that helps you find the flag. Once you feel you have found the flag, you can choose to submit it by printing 'submit <your_flag>'. Do NOT provide an explanation for your answer, only output the action you want.
Observation Message
Output: <Observation from Intercode Environment>

Table 13: Initial, Query, and Retry Prompts for Capture the Flag Evaluation.

Plan Message

```
For the following user question, let's first understand the problem and devise a
plan to solve the problem.  Then, let's carry out the plan to solve the problem step
by step.

Your plan should describe a sequence of {self.language} queries you can write to
determine the answer.  Here are three examples of coming up with a plan for a
question.

Question:  What are the names and grades for each high schooler?
Plan:
1.  Check what tables are available for use.
2.  Inspect each table to identify which has information about high schoolers.
3.  Use the table to write a query that selects the name and grade fields for each
high schooler.
...
```

Execute Plan Message

````
You will now execute your own plan.  Interact with a {self.setting} system using
{self.language} queries to answer a question.  Per turn, you will be given the
following information:

```
Observation:  Standard output from executing previous instruction
Step:  Current step
```

Your response should be {self.language} code, nothing else, formatted as follows:
```{self.language}
Your {self.language} code here
```
````

Observation Message

```
Output:  <Observation from Intercode Environment>
Step:  <Next step to execute from the plan>
```

Post-Plan Refinement Message

````
You have finished executing the plan, but it seems like there are still issues with
your answer.  Please continue to work on getting the correct answer.  Per turn, you
will be given the following information:

```
Observation:  Standard output from executing previous instruction
```

Your response should be {self.language} code, nothing else, formatted as follows:
```{self.language}
Your {self.language} code here
```
````

Table 14: Initial, Query, and Retry Prompts for Plan & Solve Strategy Evaluation. Additional Plan & Solve style demonstrations written for SQL and Bash can be found on the InterCode project GitHub. Note that the Post-Plan Refinement Message is only used for the Plan & Solve + Refine strategy discussed in § B.3. It is not used for the original Plan & Solve strategy.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] (See §D)
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] (See §D)
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you ran experiments (e.g. for benchmarks)...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] (See beginning of the appendix)
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] (See §B.1
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] (See §B.2)
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] (See §B.1)
3. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes] (See §A.2 and §A.3)
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] (See beginning of the appendix)
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [Yes] (See §B.6)
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [Yes] (See §C)
4. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [Yes] (See §B.6)
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [Yes] (See §B.6)
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [Yes] (See §B.6)