EXPLORING REASONING AND INTERACTIVE
BENCHMARKING OF LANGUAGE MODELS

Akshara Prabhakar

A Master’s Thesis
Presented To The Faculty
of Princeton University
in Candidacy For The Degree
Master of Science in Engineering

Recommended For Acceptance By
The Department Of
Computer Science

Adviser: Karthik Narasimhan
Reader: Thomas L. Griffiths

May 2024
I hereby declare that I am the sole author of this thesis.

I authorize Princeton University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Princeton University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Akshara Prabhakar
Abstract

This thesis aims to deepen our understanding of the reasoning process undertaken in Large Language Models (LLMs) and introduces a challenging benchmark to evaluate language agents’ interactive code-generation capabilities.

Part 1 delves into untangling the factors affecting the Chain-of-Thought (CoT) reasoning process in LLMs. Despite CoT prompting having demonstrated significant efficacy in enhancing multi-step reasoning capabilities, it has been found to yield biased answers and produce unfaithful explanations, raising interpretability concerns. Additionally, debates persist regarding whether LLMs truly generalize or rely on heuristics. Focusing on the symbolic reasoning task of decoding shift ciphers, we develop a simple probabilistic approach to identify these factors – the probability of the task’s expected output (probabilistic effect), what the model has implicitly learned during pre-training (memorization influenced), and the model’s attempt to adopt shorter reasoning steps (noisy reasoning) and show the consequent drastic variability in task accuracy. Through a series of experiments, we conclude that LLM behavior exhibits clear hallmarks of both memorization and true reasoning suggesting that CoT resembles a probabilistic, memorization-influenced form of noisy reasoning.

Part 2 transitions from LLMs to the emerging domain of language agents, where the LLM is now grounded in a digital environment to aid in decision-making. We introduce InterCode – an interactive RL environment to benchmark the interactive code generation abilities of language agents with execution-driven feedback. By converting traditional seq2seq datasets into three interactive code environments (Bash, SQL, and Python), we demonstrate InterCode’s viability as a testbed by evaluating multiple state-of-the-art LLMs configured with different prompting strategies such as ReAct [108] and Plan & Solve [92]. Our results underscore the benefits of interactive code generation and establish InterCode as a robust, scalable benchmark for advancing code understanding and generation capabilities. Furthermore, we use
this framework in the cybersecurity realm; by developing a Capture the Flag task in the InterCode environment, we find that while language agents possess rudimentary cybersecurity knowledge, they are not able to perform multi-step cybersecurity tasks out of the box.

Overall, our results highlight the evolving landscape of LLMs, emphasizing the need for comprehensive evaluation and understanding of their capabilities across different domains.
Acknowledgements

Firstly, I would like to thank Princeton University and the Computer Science Department for fostering an environment of abundant opportunities and growth through my graduate studies. I am grateful to my adviser, Professor Karthik Narasimhan for giving me infinite freedom in my research pursuits and offering invaluable guidance. It is through his support and direction that I got introduced to the field of cognitive science, and could collaborate with various researchers including Professor Tom Griffiths. I convey my sincere gratitude to Shunyu Yao, Samy Jelassi, and Professor Tom McCoy for mentoring me at various junctures; the insightful discussions we had greatly impacted my growth as a researcher. I am thankful to Ameet, Vishvak, and Austin for helping me get started with a research direction and guidance on choosing coursework. I consider myself fortunate to have been part of the Princeton NLP and PLI communities, as they not only facilitated brainstorming sessions but also taught me much more than research. Finally, I am deeply indebted to my parents and grandmother for always standing by me and motivating me.

This thesis is dedicated to my family.
Contents

Abstract .................................................................................. iii
Acknowledgements .................................................................. v
List of Tables .......................................................................... ix
List of Figures .......................................................................... xii

1 Introduction ........................................................................... 1
  1.1 Reasoning in Language Models ............................................ 1
  1.2 Current Landscape of Code Benchmarks ............................... 3

2 Related Work .......................................................................... 7
  2.1 In-Context Learning in Language Models ............................. 7
  2.2 Understanding CoT prompting ............................................. 7
  2.3 Interactive environments for coding ..................................... 8
  2.4 Execution-based evaluation for coding ................................. 9
  2.5 Methods for interactive/execution-based coding .................... 9
  2.6 LLMs for Cybersecurity .................................................... 10

3 Formulation for CoT Reasoning ............................................. 11
  3.1 Choice of task .................................................................. 11
  3.2 Probabilistic memorization influenced noisy reasoning .......... 12
  3.3 Prompting styles ............................................................. 13
    3.3.1 Text-based chain of thought. ...................................... 13
3.3.2 Math-based chain of thought. ........................................... 14
3.3.3 Standard. ................................................................. 14
3.4 Data preparation ............................................................... 14

4 InterCode Benchmark .............................................................. 16
4.1 Formulation ................................................................. 16
4.2 Construction pipeline ......................................................... 17
  4.2.1 Docker-based environments. ........................................... 17
  4.2.2 Data collection. ......................................................... 17
  4.2.3 Reward design. ......................................................... 17
4.3 Implementations .............................................................. 18
  4.3.1 InterCode-Bash ......................................................... 18
  4.3.2 InterCode-SQL ......................................................... 20
  4.3.3 InterCode-Python ...................................................... 21
  4.3.4 Validations. ............................................................ 21
4.4 Methodology ................................................................. 21

5 Experiments ................................................................. 24
5.1 CoT Reasoning ............................................................... 24
  5.1.1 Disentangling the factors involved in decoding shift ciphers .. 24
  5.1.2 Analyzing the probabilistic effect ................................... 28
  5.1.3 Analyzing the noise effect ........................................... 29
  5.1.4 Analyzing the memorization effect ................................... 29
  5.1.5 Producing explicit reasoning steps and impact of their validity 31
5.2 InterCode ................................................................. 33
  5.2.1 Base models comparison ............................................. 33
  5.2.2 Prompting strategy comparison .................................... 34
  5.2.3 New tasks & datasets opportunities ............................... 36
6 Conclusion

6.1 Summary & Discussion ........................................... 38
6.2 Limitations and future directions. ............................... 39

A CoT Reasoning ......................................................... 40
A.1 Math only prompt ................................................. 40
A.2 Silent thinking & unnaturalness. ................................. 40

B InterCode ............................................................... 44
B.1 InterCode Environment Details ................................. 44
  B.1.1 InterCode Interface .......................................... 44
  B.1.2 Bash Environment .......................................... 47
  B.1.3 SQL Environment .......................................... 51
  B.1.4 Python Environment ...................................... 54
B.2 InterCode Experiment Details ................................. 56
  B.2.1 Model Details .............................................. 56
  B.2.2 Additional Experiments & Analysis ..................... 56
  B.2.3 Additional Prompting Strategy .......................... 59
  B.2.4 Task Trajectory Analyses ................................ 60
  B.2.5 Capture the Flag Analysis ............................... 62
  B.2.6 Human Performance Baseline ......................... 65
  B.2.7 Prompt Templates ................................. 66

viii
List of Tables

4.1 Rundown of the two environments with Bash and SQL as action spaces developed using the InterCode framework. The numbers in parentheses refer to the number of task instances adopted from each dataset. Each environment is defined in under 200 lines of code total. Specific discussion of the environment construction and reward function can be found in § B.1.2 and § B.1.3  

5.1 Confusion matrices (100 examples; 3 probability bins {high, medium, low} for ease) for rot-4 and rot-13. Memorization effect: incorrect step outputs leading to correct final answers 2x times on average in rot-13. Prior effect: positive influence for high probability bins, negative influence for low probability bins.  

5.2 Accuracy % comparison between Text-based and Math-based CoT prompt styles. Implicit need for memorization in text-based CoT causes an overall reduced performance. Model performs best at shift levels 13, 1, 3, and 2 which are the most frequently occurring shifts in real-world corpora highlighting the memorization effects.  

5.3 Success Rate for single vs. multi turn evaluation on InterCode-SQL (refer §B.1.3). Query difficulty is adopted from Spider [114]. Best metrics are in bold.
5.4 Success Rate across file systems for single vs. multi-turn evaluation on InterCode-Bash (refer §B.1.2). To evaluate models’ ability to interact with different task settings, we evaluate disjoint sets of Bash instructions across four different file systems. Best metrics are in **bold**.

5.5 Comparison of different prompting strategies across the entire InterCode-SQL and InterCode-Bash datasets using **gpt-3.5-turbo** as the base model. *Turns* refers to the average number of turns taken for a single task episode. For Try Again and ReAct, the max number of turns $n = 10$. The highest Success Rate, fewest Turns, and lowest Error % are highlighted per dataset since they reflect more accuracy and efficient task solving. Best metrics are in **bold**.

B.1 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**.

B.2 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.

B.3 (Robustness Results) Success Rate with standard errors for single vs. multi turn evaluation on InterCode-Bash (refer §B.1.2). 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.4 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).

B.5 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.

B.6 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.

B.7 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.

B.8 Initial, Query, and Retry Prompts for Capture the Flag Evaluation.

B.9 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.2.3. It is not used for the original Plan & Solve strategy.
List of Figures

1.1 Overview of the probabilistic, memorization-influenced noisy reasoning in LLMs. ................................................................. 2

1.2 Overview of InterCode. Setting up an interactive code environment with InterCode requires a Dockerfile, dataset, reward function definition, and a small amount of subclass implementation. The interactive loop between agent and environment closely mirrors real world software development processes. While InterCode task performance is generally quantified as a binary 0/1 completion score, InterCode allows for the design of more complex evaluation criteria that can incorporate execution output and the effects of interaction on the state space. ............... 4

3.1 Shift cipher, a symbolic reasoning task. ......................................................... 12

3.2 Hypothetical accuracy vs. shift-level for various types of reasoning. Noisy one-way considers the model knows only going forward a-z; noisy two-way considers the model adopts the shorter path between going forward and backward. Memorization is based on shift levels observed in internet corpora. Probabilistic would imply high probability (bin 1) is much better than low probability (bin 5). ....................... 12
3.3 **Actual overall decoding accuracy** vs. **step accuracy** across shift levels showing the hallmarks of “noisy reasoning”, “memorization”, and “probabilistic reasoning”. The probabilistic effect is exhibited by a drop in accuracy from step outputs curves to the overall answer curves. The effect is amplified for low probability outputs as seen in the large spread in overall accuracies between bin 1 and bin 5 lines.

3.4 Text-based CoT prompt consisting of a **description** and the **demonstration** that includes several **reasoning steps**.

3.5 Text-based CoT prompt consisting of a **description** and the **demonstration** that includes several **reasoning steps**.

4.1 Overview of Prompting Strategies adjusted for evaluation on InterCode. The “Try Again” termination constraint is conditioned on reward = 1, while ReAct [108] and Plan & Solve [92] are determined by the agent itself. This is because the purpose of the "Try Again" method is to explore how capable agents are at error correction from feedback, while the other two are more concerned with the overall success of general problem-solving strategies.

5.1 Logistic Regression to model GPT-4’s performance on the shift cipher task. The logistic regression curve captures the overall trend exhibited by GPT-4. The other curves are ablations of using logistic regressions with only one feature among `output_logprob`, `shift_level`, `shift_freq` to show individual effects.
5.2 Normalized frequency distribution vs. predicted \textit{shift\_level} of step answers for \texttt{rot-20} to \texttt{rot-23}. The appearance of peaks at 26 − \textit{shift\_level} in Math-CoT and Text-CoT prompts showcasing the model’s noisy attempt in taking the shorter reasoning path i.e. moving 26 − \(x\) shifts forward instead of \(x\) backward. ............................................. 30

5.3 Growth in Success Rate with increase in number of interaction turns across models configured with Try Again prompting strategy for InterCode-Bash and SQL tasks. .................................................. 34

5.4 GPT-4’s interaction trajectory for a binary exploitation CTF task. This requires proficiency in Bash and Python, among additional knowledge and reasoning. Orange text and arrows highlight the feedback that the model attends to in generating the next action. In last step, agent submits flag. ......................................................... 37

A.1 Math-based CoT prompt consisting of a \textit{description} and the \textit{demonstration} that includes several \textit{reasoning steps}. Here both the encoded input and decoded output are number sequences abstracting out the alphabet and its influences. ............................................. 41

A.2 Accuracy vs. shift-level for math prompt with the encoded input and decoded output as number sequences. ............................................. 41

A.3 Text-based CoT prompt with consisting of a \textit{description} and the \textit{demonstration} and hidden (above) and partially hidden (below) \textit{reasoning steps}. 42

A.4 Prompt snippets with the types of mismatches/corruptions introduced highlighted in yellow. ............................................. 43
B.1 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. 46

B.2 File System structures designed for InterCode-Bash. 49

B.3 Top 30 most frequently occurring bash utilities out of the 66 in InterCode-Bash with their frequencies in log scale. 51

B.4 Distribution of gold command difficult for InterCode-SQL task data adapted from the Spider SQL dataset. 51

B.5 Example of interactions between an agent and the InterCode SQL Environment. 52

B.6 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. 52

B.7 Example of interactions between an agent and the InterCode Python Environment. 55

B.8 In this setting, an agent interacts with a Python Interpreter to 1. implement the requested method and 2. write test cases to determine function correctness. Upon submission, the reward function then evaluates the agent’s implementation with a set of unit tests. 55

B.9 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 is 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. 60
Chapter 1

Introduction

1.1 Reasoning in Language Models

Reasoning as an ability has been a weakness in natural language processing and rarely been observed in models like GPT-2 \cite{74} and smaller; requiring explicit injection of capabilities in the form of unstructured sources \cite{122, 46, 70}; commonsense \cite{89}; graphs \cite{83}; structured data \cite{71, 44} into deep-learning/transformer architectures. Ever since the emergence property was documented in billion parameter scale models \cite{102}, the tremendous growth of Language Models (LLMs) has spurred extensive research into assessing their reasoning capabilities. The Chain-of-Thought (CoT) \cite{103} prompting strategy has taken a significant step towards narrowing the gap between human and machine intelligence. CoT entails prompting an LM to generate a sequence of intermediate reasoning steps before producing the final answer, given a few in-context exemplar(s). Due to its widespread adoption, several variants \cite{47, 125, 95} have been proposed to further boost performance. Besides its performance enhancements, CoT is also purported to offer “an interpretable window into the behavior of the model” \cite{103}.

Prior works have attempted to understand what the concept of reasoning entails
Rot-x is a cipher in which each letter is shifted x positions forward in the alphabet. For example, here is a message written in rot-3:

### Rot-x text:

- f -> <>
- g -> <>
- n -> <>
- l -> <>

Therefore, the original text is: <>

Here is another message in rot-x. Decode this message one letter at a time. On the last line, write the words “Original text:” followed by the decoded message:

### Rot-x text: <encoded_test_input>

---

Figure 1.1: Overview of the probabilistic, memorization-influenced noisy reasoning in LLMs.

with regards to LMs [39, 73] and probe the emergence of reasoning, but it is not clear if model predictions are based on “true reasoning” or other “heuristics”. On one hand there are indications that LLMs are able to reason [81, 25, 78], while others show that they still struggle on tasks that humans can solve through reasoning [35, 88], even in associating the information they have memorized [40] and are strongly affected by the frequency of pre-training terms [76, 10].

In this work, we do a deep dive into a task that allows us to disentangle reasoning from memorization: Shift ciphers. We choose this task because it allows us to independently manipulate several factors that could be relevant for characterizing how LLMs solve reasoning tasks: difficulty, frequency, and answer probability. Through a range of experiments, we conclude that LLM processing can be characterized as probabilistic and memorization-influenced noisy reasoning. Figure 1.1:

- **probabilistic** - affected by the probability of occurrence of the task output. Evidence includes higher accuracy on high-probability bins and vice-versa.
- **memorization** - model’s ability to memorize patterns and frequently encountered shift levels during pre-training.
- **noisy** - involves systematic reasoning with added noise, leading to errors in every step. Evidence includes decreased accuracy as the number of reasoning steps involved increases.
- **reasoning** - representing model’s true reasoning ability.
1.2 Current Landscape of Code Benchmarks

One domain where reasoning may be practically important is in solving challenging problems such as computer programming. In recent years, the task of code generation has become an increasingly popular way of assessing the capabilities of language models. The art of computer programming is naturally an interactive process. When a human programmer writes code, she relies on several iterations of a ‘write-execute-test’ loop in order to iteratively refine solutions, plan changes, test sub-modules, and solve ambiguities by checking execution behavior. While this is reminiscent of other human endeavors like writing, code compilation and execution produce exact results that provide a deterministic form of feedback to make the refinement process more straightforward. Depending on the observed results, programmers perform various levels of debugging and rewriting, and continue the process until their code satisfies the requirements.

There has been increasing interest in recent years around the development of models that can automatically generate code given a specification in natural language [32, 97, 23, 54, 51]. Powered by large-scale pre-training over thousands of codebases [2, 42, 34], these models have shown solid performance on static benchmarks like HumanEval [16], APPS [38], MBPP [5], CodeXGLUE [58]. However, generating code in a static, sequence-to-sequence or auto-regressive fashion has several drawbacks: 1) simple errors (even typos) can propagate and there is no chance for recovery or revision, 2) there is a disconnect between the code generation process and its downstream execution on the desired software and hardware environment, and 3) there is little room for human intervention or collaboration in the code generation process.

Recently, some works have proposed the use of execution feedback or interaction [98] to benefit code generation models [50, 41, 99, 38]. However, these papers consider their own individual setup and are difficult to compare with one other due to the use of different compilers, execution environments, feedback signals, and assumptions on the
Figure 1.2: Overview of InterCode. Setting up an interactive code environment with InterCode requires a Dockerfile, dataset, reward function definition, and a small amount of subclass implementation. The interactive loop between agent and environment closely mirrors real world software development processes. While InterCode task performance is generally quantified as a binary 0/1 completion score, InterCode allows for the design of more complex evaluation criteria that can incorporate execution output and the effects of interaction on the state space.
interactive process such as human participation to create task descriptions or provide natural language feedback. This makes it difficult to compare existing methods for code generation and to clearly understand the benefits of interactive generation.

To address these issues, we propose InterCode, the first standard coding benchmark designed natively with an interactive execution environment. Closely mimicking the human decision-making process, InterCode allows a coding agent to interactively receive feedback from compilers/interpreters that execute its code, and to submit further refinements. We design InterCode to be like a standard reinforcement learning (RL) environment that requires minimal human intervention and one in which generated code is treated as actions, which are executed to reveal observations. Our framework is (1) language and platform agnostic and can easily be used for new coding problems, (2) uses self-contained Docker environments to provide safe execution, and (3) compatible out-of-the-box with traditional seq2seq generation methods, while also enabling and empowering the development of new interactive techniques.

We demonstrate the power of the framework by implementing Bash, SQL, and Python tasks within InterCode, building on pre-existing static datasets [123, 57, 5]. We perform experiments across diverse models and prompting methods, including ReAct [108] and Plan & Solve [92]. Our findings concretely showcase the benefits of interaction towards solving coding tasks, discuss the distribution of distinct code understanding challenges across different task settings, and explore the ease with which new tasks and datasets can be defined using InterCode.

To summarize:

- Amidst the ongoing debate of reasoning vs. memorization, our results indicate a middle-ground perspective – LLM behavior displays traits of both memorization and generalization.
- We develop InterCode, a new, universal framework for interactive code gen-
eration, which provides ease of use, extensibility, and safety, and perform a comprehensive evaluation of state-of-the-art models, identifying several avenues for improvements.
Chapter 2

Related Work

2.1 In-Context Learning in Language Models

The ability of LLMs to “learn” a task purely from example demonstrations given in their context without any additional training [8], has led to extensive investigation into how it operates. Theoretical frameworks [105, 121] model the pretraining data as a mixture of Hidden Markov Models and show In-Context Learning (ICL) is the result of implicit Bayesian inference; for which empirical evidence has been provided through synthetic data and tasks [21]. The emergence of ICL has been attributed to factors like data distributional properties [11], pertaining term frequencies [76], and creation of task vectors [37]. However, the extent of true “learning” is unclear [66, 62], and a recent work [48] compares ICL to conventional learning showing it relies on in-context label information but cannot overcome pre-training preference.

2.2 Understanding CoT prompting

Since CoT demonstrated superior gains on reasoning tasks by presenting intermediate steps before reaching the final answer, several studies have endeavored to uncover its underlying mechanisms. Theoretical evidence has been formulated regarding the
reasoning capability of Transformers [31], and the significant reduction in sample complexity of ICL by CoT [55]. Recent works, including [109, 91, 59] study the effects of demonstration perturbations while [91, 95] showed that the relevance and ordering of reasoning fragments were more important than their accuracy. Alongside, input perturbations [84] could systematically bias models toward specific answers, and the effect of increasing the length of reasoning chains [45] to enhance performance on certain tasks has been studied. [96] perceive demonstrations as just guides for models on how to solve a task, with models primarily mimicking the format of these prompts. Another work [104] attempted to probe CoT based on the influence the input tokens have on model output.

A common thread running throughout prior works is the debate about memorization vs. learning/abstraction. Our contribution is to propose a middle ground between these perspectives.

### 2.3 Interactive environments for coding

Most coding benchmarks (e.g. SQL - Spider [114], KaggleDBQA [52]; Bash - NLC2CMD [1], NL2Bash [57]; Python - HumanEval [16], APPS [38], MBPP [5], CodeXGLUE [58], CodeNet [72]) frame the coding problem as a sequence transduction problem (from instruction to code), rather than an interactive decision making problem with an execution environment. Attempts have been made to simulate interaction by developing conversational, dialogue-style [115, 113], multi-step problem solving [64] datasets, which involve pre-annotated human-designed queries. The work closest to InterCode has been recent explorations of Python Jupyter Notebooks as a natural choice for interactive coding [41, 50, 111]. However, task data and settings often constrain allowed actions to a closed domain of code and libraries [50, 111], use evaluation procedures or metrics that may not generalize [41], require human-in-the-
loop participation (i.e. create task contexts, write problems, evaluate execution per task instance) [50], or are Python-exclusive [41, 50, 111, 99]. InterCode provides a more general purpose foundation defining interactive coding tasks that enables easy construction of diverse task settings, can have any programming language(s) as the action space, and has automatic, execution-based evaluation.

2.4 Execution-based evaluation for coding

Evaluation for NL-to-code generation models has recently shifted away from surface form similarity metrics (BLEU [67, 2], ROUGE [56], Exact Match) towards execution oriented ratings (unit tests [5, 16, 41, 50, 38], output matching [27, 41, 123]). The rigidity of surface form analysis overlooks code syntax features, ignores execution effect, or over-penalizes alternative solutions [124]. On the contrary, execution-based assessment is a more thorough and comprehensive score of code functionality [38] and is a more natural fit for open-domain program usage that does not constrain code generation to a subset of the language space [99]. However, for newer benchmarks and datasets that put forth task definitions incorporating execution-based evaluation (APPS [38], ExeDS [41], ODEX [99]), the fundamental code generation task (Context + Code → Execution → Score) is still devoid of interaction. InterCode combines execution-based evaluation with flexible task construction, enabling more diverse problem-solving paradigms within a unified coding task formulation. InterCode’s use of virtual containers as execution sandboxes protect against harmful actions and allow for advanced evaluation criteria beyond the aforementioned ones.

2.5 Methods for interactive/execution-based coding

The value of generative code models and interactive problem solving has motivated a recent proliferation of work to augment reasoning capabilities’ of existing language
models [108, 80, 92, 107, 119, 17] or propose new modeling techniques to tackle coding as a sequential decision making and reasoning tasks [9, 18, 28, 54, 15, 51], of which evaluation is unit test based. Approaches that leverage execution typically use re-ranking [120, 63, 112, 116] or majority vote [18, 54, 79] to decide on a final prediction. Additional work also explores incorporating human-in-the-loop [14, 49], compilers [94], and text [93, 118] feedback. A common thread among these contributions is that 1) the task setting can only provide the investigated form of feedback and 2) sought-after capabilities are exemplified by strong performance on favorably curated tasks and datasets, rendering comparisons across benchmarks tedious. InterCode has the potential to standardize the evaluation of these methods because 1) the interactive coding task is a conglomeration of many interesting interaction, reasoning, and decision-making challenges and 2) InterCode’s task construction makes it possible to incorporate a wide variety of sources of feedback.

2.6 LLMs for Cybersecurity

Moving from specialized deep learning methods for cybersecurity applications such as cyber attacks [4], malware detection [69], intrusion detection [90], LLMs have been recently used to weaponize code and generate attacks [6, 26]. Specifically for CTF, works since the competition’s inception has primarily focused on its use as an educational tool [13, 87, 12]. Automation and neural methods were first introduced as augmentations to the game’s proceedings [19, 36], before later works in recent years have then attempted to tackle the game itself formulated as a reinforcement learning task [68, 20, 117]. These approaches have typically focused on a subfield of CTF (e.g. web hacking [29], SQL injection [20], threat detection [33]). [82] carries out an qualitative case study of three LLMs on 7 CTF questions.
Chapter 3

Formulation for CoT Reasoning

3.1 Choice of task

Shift cipher is a kind of substitution cipher used to encode text messages. It involves replacing every letter in the message with another letter that is some fixed number of positions (called shift level) forward in the alphabet as shown in Figure 3.1. It falls under the category of symbolic reasoning as it involves the manipulation of abstract symbols according to a formal rule. The goal of the LLM is to decode and recover the original text message encoded using shift cipher. This task is chosen as it is easy to systematically generate data of varying difficulty for the model (explained later), probably not explicitly encountered by the LM during its pre-training, and easy to verify correctness yet is sufficiently difficult even for the most advanced models - GPT-4 [65] attains avg. 32% accuracy. More importantly, every step in the reasoning chain for decoding is independent of other steps making it easier to isolate and study properties. We chose this task because it enables us to investigate several hypotheses about possible types of reasoning that GPT-4 could be performing. In the next section, we list these possible types of reasoning and then discuss how this task enables us to test whether GPT-4 displays these types of reasoning.
Figure 3.1: Shift cipher, a symbolic reasoning task.

Figure 3.2: **Hypothetical** accuracy vs. shift-level for various types of reasoning. **Noisy one-way** considers the model knows only going forward a-z; **noisy two-way** considers the model adopts the shorter path between going forward and backward. **Memorization** is based on shift levels observed in internet corpora. **Probabilistic** would imply high probability (bin 1) is much better than low probability (bin 5).

Figure 3.3: **Actual overall decoding accuracy** vs. step accuracy across shift levels showing the hallmarks of “noisy reasoning”, “memorization”, and “probabilistic reasoning”. The probabilistic effect is exhibited by a drop in accuracy from step outputs curves to the overall answer curves. The effect is amplified for low probability outputs as seen in the large spread in overall accuracies between bin 1 and bin 5 lines.

### 3.2 Probabilistic memorization influenced noisy reasoning

Reasoning, one of the key aspects of human intelligence, is the process of thinking about something logically and systematically using evidence and past experiences to make a decision [100, 101, 30]. To motivate our formulation, we begin by explaining a few types of reasoning processes.

(a) Symbolic reasoning: shift ciphers can be solved with a simple algorithm, so a system using fully systematic reasoning, should attain 100% accuracy on all shift
cipher examples.

(b) Noisy reasoning: like symbolic reasoning, but with some noise added in - where the noise introduces some possibility of each step being wrong. Thus, if the system is doing noisy reasoning, we should see accuracy decrease as we increase the number of steps that need to be performed. Shift ciphers let us test this possibility: by varying $shift_{\text{level}}$, we can modulate the number of steps that need to be performed, and observe if accuracy decreases as the number of steps increases.

(c) Memorization: If memorization is all that’s happening, we should see higher performance in the cases that are frequently encountered during pre-training than the ones that are not. This is one of the strongest reasons for using shift ciphers because they allow us to disentangle “difficulty” from “frequency in pre-training”.

(d) Probabilistic-influenced reasoning: As the probability of the correct output decreases, accuracy should also decrease. Shift ciphers are also ideal for testing this because the output can be any string we want - allowing us to modulate the probability easily.

Figure 3.2 spotlights the hypothetical performances of a system adopting these reasoning approaches.

3.3 Prompting styles

The description of the task at hand is common across all styles.

3.3.1 Text-based chain of thought.

Figure 3.4 demonstrates the prompt used for decoding a shift cipher message. To get a reasoning step correct, the model must have learned the alphabet during pre-training.
Rot-13 is a cipher in which each letter is shifted 13 positions forward in the alphabet. For example, here is a message written in rot-13:

Rot-13 text: "fgnl"

To decode this message, we shift each letter 13 positions backward:
1. f -> s
2. g -> t
3. n -> a
4. l -> y

Therefore, the original text is: "stay"

Here is another message in rot-13. Decode this message one letter at a time. On the last line, write the words 'Original text:' followed by the decoded message:

Rot-13 text: <test_input>

Figure 3.4: Text-based CoT prompt consisting of a description and the demonstration that includes several reasoning steps.

3.3.2 Math-based chain of thought.

Figure 3.5 demonstrates a prompt that utilizes arithmetic skills in the reasoning steps eliminating the need for the model to know the alphabet ordering as the letter-position mapping is specified.

3.3.3 Standard.

This is a prompt with just the description provided and no demonstration.

3.4 Data preparation

We construct a dataset comprising 7-letter words having exactly 2 tokens (measured using the tokenizer used by GPT-4) to control for confounding factors. Specifically, we find all 3-letter and 4-letter tokens from the lowercase English alphabet and form words by considering all possible combinations of a 3-letter and 4-letter token. The words are scored by their log probability and arranged in descending order. Subsequently, five bins are formed by selecting equidistant log probability values as centers. Following this, 150 words (100 used to train, 50 to test) with log probability values nearest to each respective center are allocated into 5 distinct bins \{bin1, bin2, bin3, bin4, bin5\}, i.e. bin1 meant high probability and bin5 meant low probability. The log
Rot-13 is a cipher in which each letter is shifted 13 positions forward in the alphabet. For example, here is a message written in rot-13:

Rot-13 text: “fgnl”

To decode this message, we need to shift each letter 13 positions backward. Let’s start by writing the position-letter mapping for the alphabet:

a -> 0
b -> 1
c -> 2
...

Next, we find the encoded letter as follows:

Position of original letter = (Position of given letter - 13) mod 26

Then map the found position to the corresponding letter using the letter-position mapping.

Using this,
1. f -> (5 - 13) mod 26 = 18 -> s
2. g -> (6 - 13) mod 26 = 19 -> t
3. n -> (13 - 13) mod 26 = 0 -> a
4. l -> (11 - 13) mod 26 = 24 -> y

Therefore, the original text is: “stay”

Here is another message in rot-13. Decode this message one letter at a time. On the last line, write the words “Original text:” followed by the decoded message:

Rot-13 text: <encoded_test_input>

Figure 3.5: Text-based CoT prompt consisting of a description and the demonstration that includes several reasoning steps.

...
Chapter 4

InterCode Benchmark

4.1 Formulation

The InterCode benchmark formalizes interactive coding with execution feedback as a partially observable Markov decision process (POMDP) $(\mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{R})$ with instruction space $\mathcal{U}$, state space $\mathcal{S}$, action space $\mathcal{A}$, observation space $\mathcal{O}$, transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$, and reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow [0,1]$. Given a coding instruction $u \in \mathcal{U}$ in natural language, an agent issues code or a special submit keyword as an action $a_t \in \mathcal{A}$. An action is admissible [106] if it can be parsed and executed in the compiler/interpreter environment, and an admissible action incurs a change in the latent state space $s_{t+1} \in \mathcal{S}$, and an execution feedback as observation $o_{t+1} \in \mathcal{O}$. The interaction loop repeats until the submit action is issued, wherein the task episode ends and a reward $r = \mathcal{R}(s_T, \text{submit}) \in [0,1]$ is computed, with 1 representing task completion. We use the Success Rate (SR) metric, defined as the proportion of task episodes where $r = 1$. We also define the Error % metric, which is the percentage of non admissible actions across task episodes.
4.2 Construction pipeline

At a high level, InterCode decomposes the construction of an interactive coding task into three modular parts: (1) environment construction, (2) data collection, and (3) reward design. This workflow allows for the safe execution of transition functions, flexible reward design, and convenient adaptation of existing instructions to an interactive setting.

4.2.1 Docker-based environments.

InterCode uses Docker [61] virtual containers as a general-purpose execution sandbox. Given a Dockerfile that defines a system and execution entrypoint, InterCode creates a corresponding, stateful virtual container that hosts the desired state space and transition function. We choose Docker as the basis of InterCode’s environment construction for its safe execution in virtual containers, reproducibility of a Dockerfile across any Docker-equipped machine, and excellent coverage of application code, libraries, and dependencies offered by the Dockerfile DSL.

4.2.2 Data collection.

InterCode requires that a dataset has at minimum two fields: query, a natural language instruction $u \in \mathcal{U}$, and gold, an answer or code block that is a procedure for generating the correct answer. We define these conditions to make it easy to adapt existing text-to-code datasets to an interactive setting while also leaving plenty of bandwidth for constructing new tasks and datasets.

4.2.3 Reward design.

Across a single task episode, the action, observation, and state modification (if any) per interaction loop are implicitly logged by InterCode. InterCode’s default reward
<table>
<thead>
<tr>
<th>Action Space</th>
<th>Environment</th>
<th>Dataset</th>
<th>Reward Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bash</td>
<td>Ubuntu Terminal</td>
<td>NL2Bash [57] (200)</td>
<td>Latest Std. Output + File System Δ</td>
</tr>
<tr>
<td>SQL</td>
<td>MySQL Database</td>
<td>Spider 1.0 [114] (1034)</td>
<td>Latest Std. Output</td>
</tr>
<tr>
<td>Python</td>
<td>Python Interpreter</td>
<td>MBPP [5] (117)</td>
<td>Submitted Function</td>
</tr>
</tbody>
</table>

Table 4.1: Rundown of the two environments with Bash and SQL as action spaces developed using the InterCode framework. The numbers in parentheses refer to the number of task instances adopted from each dataset. Each environment is defined in under 200 lines of code total. Specific discussion of the environment construction and reward function can be found in § B.1.2 and § B.1.3.

Function determines task completion via an exact match of the agent’s execution output (observation and state modifications) against the gold command, where 1 is awarded only if all components match. Since Exact Match is usually too stringent of an evaluation criteria, InterCode exposes a reward function endpoint that has access to both the interaction history and the execution container, allowing for custom reward function definitions that can incorporate multiple signals.

### 4.3 Implementations

Following the procedure discussed in Section 4.2, we create two separate InterCode based environments where Bash and SQL are the action spaces respectively. Table 4.1 summarizes them.

#### 4.3.1 InterCode-Bash.

We define a `bash` shell within an Ubuntu Operating System as the task setting. To evaluate an agent’s ability to adapt generations to different situations, we architect four distinct file systems that can be swapped into the Bash environment by changing a single line in the `Dockerfile`.

We bootstrap the NL2Bash [57] dataset (which lacks specificity in queries and grounding to any underlying file system, preventing it from being used directly for
interactive evaluations) to create an interactive coding task where an agent completes an instruction via bash actions. Transferring NL2Bash to the interactive task setting requires simple transformations to ground instructions and gold code blocks in the file system. First, we consider a subset of 1000 commands with each having \( \geq 4 \) utilities. We then filter out commands that are non-UNIX, non-Linux, or use utilities we currently do not support (e.g. ssh, sudo, time, and GUI-dependent utilities). Finally, we enhance under-specified commands with specific file names/directory names/paths and update deprecated utilities/flags. The resulting 200 commands are grouped into 4 disjoint sets, 3 of which were grounded to custom-designed file systems, while one set is file-system agnostic. This categorization allows for a comprehensive evaluation of different command-grounding scenarios.

The InterCode-Bash dataset instructions typically make one or both of the following two types of requests. It either 1. Requests information that can be answered via execution output (i.e. "How many files...", "What is the size of...", "Where is <file> stored?") or 2. Requests a change to the location/configuration/content of a file or folder (i.e. "Move dir1 folder...", "Set permissions of...", "Append a line to..."). Therefore, we define a custom reward function that evaluates an agent’s performance against file system modifications and the latest execution output. Execution output is graded with a simple lexical similarity function. File system assessment is done in two parts. First, a comparison of the agent’s and gold command’s list of file system changes (list of \([\text{path}, \text{modification type} \in \{\text{added}, \text{changed}, \text{deleted}\}]\) entries) reveals any extraneous or missing changes. Second, md5sum hashes of each commonly edited file path are compared to determine if an added or changed file was altered correctly. 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. Additional Bash statistics and design details are discussed in § B.1.2.
4.3.2 InterCode-SQL.

We write a **Dockerfile** that defines a SQL interpreter within a MySQL database as the task setting. To create the databases and tables necessary for the task dataset, we write type resolution scripts and perform database conversions using the `sqlite3mysql` [85] Python library to adapt the Spider [114] database and table schema to a MySQL format. We then consolidate all setup code into a single, unified MySQL `.sql` dump that contains the complete set of schemas for all tables across 20 different databases. On container start-up, this file is invoked automatically, creating and populating databases with tables and tables with records.

The Spider [114] dataset is a large-scale cross-domain dataset originally meant for evaluating SQL query generations from natural language questions. We adapt the development set, which contains 1034 task instances, and remove all extraneous columns aside from the natural language questions and gold SQL command. The **instruction** and **gold** values do not require any additional pre-processing to be compatible with the MySQL task environment.

Finally, 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, where both outputs are a list of records. A non-tabular execution output receives a reward of 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. To quantify how sorted the agent output is relative to the gold output, we lean on Kendall’s $\tau$ 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 receives a score of 1. Visualizations like Figure 1.2 for SQL along with a more extensive implementation discussion for this environment are in § B.1.3.
4.3.3 InterCode-Python.

In this setting, we define a Python interpreter running within an Ubuntu operating System as the task setting. The Dockerfile can be configured to run any Python version. The interpreter is not initialized with any dependencies, but PyPI packages can be installed and used by the agent.

We use the MBPP [5] dataset which presents the code completion task of synthesizing Python code from a method header and docstring. Evaluation of correctness is performed with an associated set of unit tests given by MBPP. The MBPP dataset is straightforward to adapt to the interactive setting, requiring no modifications to the query or evaluation components. Finally, we directly inherit MBPP’s evaluation procedure of proportion of unit tests passed. With InterCode, it is easy to use existing datasets to evaluate how well models can use different programming languages as actions.

4.3.4 Validations.

To verify the functionality of action execution in the task environment and the correctness of custom reward functions, we write testing scripts for both Bash and SQL that pass the gold command in as a dummy agent’s action to ensure that the command is admissible and executes without error, and to verify that the reward received by the command is 1. To confirm that InterCode’s dataset specification is enforced across multiple accepted file formats, we define a custom InterCode data loader class which is then rigorously unit tested.

4.4 Methodology

We perform preliminary experiments to gauge the proficiency and behavior of current large language models on interactive coding tasks with Bash and SQL. To observe and
elicit relevant reasoning skills, we draw on several existing prompting strategies that have been put forth to augment language models’ reasoning and problem-solving skills. We apply these prompting strategies to models across the following three families: OpenAI (text-davinci-003, gpt-3.5-turbo, gpt-4), PaLM-2 (text-bison-001, chat-bison-001) [3], and Open Source (Vicuna-13B [22], StarChat-16B [53]).

Figure 4.1 visualizes the four adjusted prompting strategies we evaluate on InterCode.

Single Turn is a zero-shot attempt. A model is given a simple description of the task setting and asked to generate code in a specific programming language that would address the query. The first generation in response to the user’s question is then evaluated in the InterCode environment.

“Try Again” is an iterative feedback set up. In the initial message, the agent is informed of the task setting and its interactive nature; an agent has multiple turns to
interact with the system, wherein each turn, upon generating an action, the execution output of the action is fed back as an observation. This continues until a reward of 1 (task completion) is achieved or the number of turns ($n$) is exhausted. The agent’s position in this approach is meant to mirror human software development as closely as possible. The goal of this method is to probe language models’ raw interactive coding abilities in addition to illustrating the benefits and different challenges that arise in interactive coding tasks.

**ReAct and Plan & Solve.** We write prompts and design workflows that follow the text and task configurations described in ReAct [108] and Plan & Solve [92] as faithfully as possible. For these two approaches, the termination of a task episode is conditioned upon the agent’s own judgment, as our goal with these methods is to gauge the transferability to and efficacy of existing reasoning frameworks with respect to the interactive coding task. Full prompt templates are included in §B.2.7.
Chapter 5

Experiments

5.1 CoT Reasoning

We run experiments using OpenAI’s GPT-4 (gpt-4-0613 version) that is state-of-the-art in many areas, setting temperature to 0 and max_new_tokens to 200. Using GPT-4 helps us to control several sources of noise making it easier to focus on the task itself and isolate the factors affecting CoT. It ensures that the format of the demonstration is closely followed, and copy errors are avoided (correctly copying the letter from encoded text input in the reasoning steps and the prompt metadata such as the letter-position mapping). The main prompt style we use throughout is Text-based CoT as it effectively helps us observe and analyze all the factors involved in the reasoning factors. Math-based CoT served to aid in some scenarios as mentioned.

5.1.1 Disentangling the factors involved in decoding shift ciphers

A direct comparison between Figures 3.2 and 3.3 highlights the noisy two-way reasoning, memorization, and probabilistic reasoning types detailed in § 3.2, culminating in the observed overall decoding accuracy curve in blue in Figure 3.3. Despite rot-13
requiring most reasoning steps and being the most noisy, the memorization effect notably enhances the model’s performance. While the step-level probabilistic effects may not be conspicuous due to the output being a single letter, their impact becomes evident at the overall level, as evidenced by the substantial variance between bin 1 and bin 5. This illustrates the hallmarks of noisy reasoning, memorization, and probabilistic reasoning. Now, to make this result more rigorous, we undertake a logistic regression analysis to ascertain the statistical significance of each of these factors on performance.

**A simple probabilistic approach to model the reasoning process.**

We begin with all measurable features that can aid in explaining the model’s observed performance across 25 shift levels on varying probabilities of expected output.

- **input_logprob** - the log probability of the encoded input text as measured by GPT-2 [74]. These inputs are created by taking our prepared word dataset (described in § 3.4) and encoding them using all shift levels 1 to 25. This corpus of words is mostly of low probability.
- **output_logprob** - the log probability of the expected ground-truth output text (words in our prepared dataset) as measured by GPT-2.
- **shift_freq** - we obtain the frequency of occurrence of all shift levels by manually analyzing the C4 corpus [75]. The assumption is that the distribution of shifts in C4 is similar to the distribution of shifts in the training data for GPT-4.
- **output_num_chars** - number of characters in the output word. By construction this is 7.
- **output_num_tokens** - number of tokens in the output word. By construction this is 2.

Decoding shift ciphers with CoT involves:
1. copying the input letters from the encoded input in corresponding reasoning steps,
2. given the input letter for a reasoning step, producing the output letter for that step,
3. generating the final answer by concatenating outputs from all the reasoning steps.

Step 1 is done accurately by advanced models like GPT-4. The accuracy of steps 2 and 3 are influenced by
- the probability of occurrence of the expected output (§ 5.1.2),
- its ease to decode smaller shift levels and attempts to adopt shorter alternate reasoning paths making the reasoning noisy (§ 5.1.3),
- what shift levels the model has encountered more during its pre-training (§ 5.1.4),
- and whether the model explicitly generates helpful step outputs (§ 5.1.5).

Accordingly, we set up a logistic regression to model the probability of the test model accurately solving the shift cipher task (decoding the encoded input text message) when prompted with CoT,

\[
\text{correct} \sim \min(\text{shift}\_\text{level}, 26 - \text{shift}\_\text{level}) + \text{input}\_\text{logprob} + \text{output}\_\text{logprob} + \text{shift}\_\text{freq} \tag{5.1}
\]

The intuition for using \(\min(\text{shift}\_\text{level}, 26 - \text{shift}\_\text{level})\) arises from the observation that LMs try to adopt shorter reasoning paths (Figure 3.3, 5.2).

**Observations from the probabilistic model.**

The logistic regression model described in § 5.1.1 found the following features had a statistically significant effect on model performance: \text{output}\_\text{logprob}, \text{shift}\_\text{level}, and \text{shift}\_\text{frequency}. We associate \text{output}\_\text{logprob} to model the probabilistic effect, \text{shift}\_\text{frequency} to model memorization, and \text{shift}\_\text{level} to model the noise. Fig-
Figure 5.1: Logistic Regression to model GPT-4’s performance on the shift cipher task. The logistic regression curve captures the overall trend exhibited by GPT-4. The other curves are ablations of using logistic regressions with only one feature among output_logprob, shift_level, shift_freq to show individual effects. Figure 5.1 shows the prediction of logistic regression compared to the model’s performance. Given the limited training set we collect and the simplistic nature of the model, it is able to roughly capture the trend.

Isolating the reasoning process.

To assess the model’s performance on only the decoding task without being influenced by probabilistic and memorization effects, we create a version of the problem using the math-based CoT prompt by replacing the encoded test word with its equivalent number sequence in which the elements are the positions of the corresponding letters of the input word in the alphabet (refer § A.1 Figure A.1). The input and output are now number sequences abstracting out the alphabet. The model’s near-perfect accuracy across shift levels and related error analysis shown in § A.1 Figure A.2 implies that the model is in fact very strong in the core reasoning needed for decoding shift ciphers. However, when the same task is shifted to the alphabet domain, it is impacted by effects discussed in the following sections.
Table 5.1: Confusion matrices (100 examples; 3 probability bins {high, medium, low} for ease) for rot-4 and rot-13. Memorization effect: incorrect step outputs leading to correct final answers 2x times on average in rot-13. Prior effect: positive influence for high probability bins, negative influence for low probability bins.

5.1.2 Analyzing the probabilistic effect

Despite the reasoning process to be followed by the model for decoding any input being the same, we note from Figure 3.3 that the overall decoding accuracy is much lower for output texts having a low probability of occurrence (the gap between bin 5 vs. bin 1 lines in blue). Mathematically, we are visualizing the difference between $P(\text{correct\_step\_output}|\text{shift\_level})$ and $P(\text{correct\_final\_output}|\text{shift\_level})$. The consequent unfaithfulness aspect is also captured by an average 72% relative drop in accuracy compared to producing the step-level outputs accurately and concatenating them to form the final answer when averaged across all shift levels and probability bins.

Table 5.1 more concretely analyzes the unfaithfulness in CoT due to the flip effects of the output’s probability. On average, incorrect intermediate chains yield correct answers 44% times in high probability but just 10% times in low probability. On the other hand, correct intermediate chains result in incorrect answers 4% times in high probability outputs but 12% times in low probability outputs.

This establishes that models over-rely on their prior knowledge [43] thereby introducing biases in their outputs. If the answer has a high probability of occurrence, the
model’s priors favor generating it even if its intermediate reasoning steps suggest an alternative output. Conversely, if the answer is of lower likelihood, then even if the chain of reasoning is correct, the priors exert a detrimental influence causing wrong answer generations.

5.1.3 Analyzing the noise effect

The inverse impact of the statistically significant shift_level can be characterized as noise in the reasoning process. Smaller shifts are done easily, presumably due to fewer reasoning steps, as seen in the falling accuracy values from rot-1 to rot-12 and it is two-way as higher shift levels recover in Table 5.2.

Tendency to adopt shorter reasoning paths. There is an interesting emerging phenomenon as seen in Figure 5.2; for large shift_level, the emergence of additional peaks at 26 − shift_level in both text and math cases. It looks like the model has understood the equivalence between x shifts backward and 26 − x shifts forward; but in an attempt to take a shorter reasoning path, it is still affected by the demonstration showing backward shifting, possibly giving rise to these peaks. This noise is very prevalent until rot-23 after which the model performs similarly to rot-1 and rot-2.

5.1.4 Analyzing the memorization effect

Shifts 13, 3, 1, and 2 (in descending order) were the most frequently occurring shift levels in the analyzed C4 corpus, and we observe in Table 5.2 how the model uses this to its advantage to perform best at these shift levels in both text-based and math-based CoT. Even with just standard prompting, GPT-3.5 and GPT-4 have been found to perform significantly better for such shifts [60] and they attribute it to task probability. The aforementioned property of unfaithfulness also partly demonstrates this effect. A comparison between Table 5.1b and Table 5.1a shows how in the more
Figure 5.2: Normalized frequency distribution vs. predicted shift level of step answers for rot-20 to rot-23. The appearance of peaks at $26 - \text{shift level}$ in Math-CoT and Text-CoT prompts showcasing the model’s noisy attempt in taking the shorter reasoning path i.e. moving $26 - x$ shifts forward instead of $x$ backward.

Table 5.2: Accuracy % comparison between Text-based and Math-based CoT prompt styles. Implicit need for memorization in text-based CoT causes an overall reduced performance. Model performs best at shift levels 13, 1, 3, and 2 which are the most frequently occurring shifts in real-world corpora highlighting the memorization effects.

frequently encountered rot-13, despite incorrect chain step outputs, the final answer is accurate on average twice as often for rot-13 compared to rot-4.

**Another effect of the model’s implicit memorization.** It is more likely that the model has seen the alphabet in the forward direction due to which it finds it harder to go backward in the alphabet and circle backward from a to z. Using our two prompting styles helps us verify this as when we remove the model’s dependence on its memory of the alphabet in the Math-CoT case by providing a definite formula, the model consistently does better across most shift levels (Table 5.2).
5.1.5 Producing explicit reasoning steps and impact of their validity

The prompts used for the experiments mentioned below are shown in § A.2 Figure A.3.

Strong reliance on the surface strings produced in reasoning steps.

1. Hidden chain. In this case, we use the Text-CoT prompt but instruct the model to not reveal step answers and instead output a *. The step answers in the demonstration are also replaced by '*' - thus we leave the format of reasoning intact but the expected generation token has changed. When tested, the performance is similar to that of standard prompting.

2. Partially hidden chain. Next, we ask the model to explicitly think about the correct letter that should go in the place of the star but just not write it down. In the demonstration, we first provide an example with all step answers and then replace it with '*' to guide the model. But even this does not rescue the model.

This supports the view that CoT is more about “self-conditioning” - it needs to explicitly produce these relevant step outputs, that later predictions can be conditioned on to generate the final answer. Merely instructing a model to think silently is not helping, proving that the reasoning is not an intrinsic process.

Unnaturalness phenomenon: Description vs. demonstration. All prompts variants are displayed in § A.2 Figure A.4.

1. Random corruptions. Here the step answers in the demonstration are replaced by a random letter - both format and expected generation token (a letter) are left unchanged.

2. Systematic mismatches. The description for rot-13 is provided while the demonstration is that of rot-14. This can be viewed as a systematic "helpful" corruption as it still retains the overall format and expected generation token (a letter)
In none of these cases (i.e. random or systematic corruption) is the model severely distracted from the description, the converse was observed with GPT-3 [77, 43]. This reinforces the fact that there is no strong correlation between the validity of the reasoning shown in the demonstration and the model performance [91, 59, 110]. The demonstration seems to merely guide the model to solve the task by providing a format to generate accurate reasoning steps.
5.2 InterCode

5.2.1 Base models comparison

Task performances. We first compare the success rate of models in the Single Turn and Try Again settings for both the InterCode-Bash and SQL datasets. From Table 5.3 and Table 5.4, we observe that performance across different levels of task difficulty (SQL) and different file systems (Bash) is superior in the interactive setting for all models, with a notable multi-fold increase for GPT-4 (9.1% → 73.7%) on the InterCode-SQL task.

Analysis of interactions. Manual inspection of trajectory logs indicates that models actively exercise later turns for discovering relevant context, correcting errors via execution feedback as observations, and solving problems via iteratively constructing and editing actions as affirmed by Figure 5.3. In addition, models also demonstrate a level of planning and modular problem solving; for instructions with gold commands that chain multiple commands together (i.e. with |, >, or ; in bash) or consist of multiple sub-problems (i.e. subqueries in SQL), models will use observations from solving smaller sub-problems in earlier turns to compose the higher-order action. Trajectories that exhibit these phenomena are in § B.2.4

Failure cases. With that said, both Figure 5.3 exhibits a plateauing in Success Rate and and Error %. This suggests that as the amount of context and feedback builds up, models are less capable of discerning relevant past history toward future actions. In late-turn scenarios, task episode trajectories often reveal repetition of earlier actions, a failure to effectively use recent observations towards deciding an appropriate next action, or an inability to recognize that a current problem-solving chain of thought is inconclusive or futile. This is particularly evident for hard and extra level InterCode-SQL task instructions that require context spanning across several tables and actions that incorporate multiple clauses. We note that even when
Figure 5.3: Growth in Success Rate with increase in number of interaction turns across models configured with Try Again prompting strategy for InterCode-Bash and SQL tasks.

The full schema of all tables and their descriptions are offered in addition to the original instructions, models still benefit greatly from using interaction to experiment with different JOIN and filtering operators across multiple turns, as demonstrated in § B.2.2. A larger context window size, retrieval of useful memory, and more adaptive reasoning paradigms are just a handful of potential solutions to overcoming such challenges.

5.2.2 Prompting strategy comparison

Initiating language agents with prompting strategies that encourage different forms of reasoning toward problem-solving improves performance on the interactive coding task to varying degrees. Table 5.5 presents side-by-side comparisons of the success rate, number of turns, and error rate per strategy. Compared to Try Again, which lacks specific guidance on leveraging multiple turns, more explicit reasoning frameworks such as ReAct and Plan & Solve policies generally achieve higher success rates (SQL: 47.3% → 58.7%) with fewer turns and a higher rate of admissible commands.

Different tasks present different learning challenges. An important skill to solving the InterCode-SQL task is the ability to discover context and construct
<table>
<thead>
<tr>
<th></th>
<th>Try Again ((n = 10))</th>
<th>ReAct ((n = 10))</th>
<th>Plan &amp; Solve</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SR</td>
<td>Turns</td>
<td>Error %</td>
</tr>
<tr>
<td>SQL</td>
<td>47.3</td>
<td>7.25</td>
<td>46.4</td>
</tr>
<tr>
<td>Bash</td>
<td><strong>46.5</strong></td>
<td><strong>6.15</strong></td>
<td><strong>24.9</strong></td>
</tr>
</tbody>
</table>

Table 5.5: Comparison of different prompting strategies across the entire InterCode-SQL and InterCode-Bash datasets using **gpt-3.5-turbo** as the base model. **Turns** refers to the average number of turns taken for a single task episode. For Try Again and ReAct, the max number of turns \( n = 10 \). The highest Success Rate, fewest Turns, and lowest Error % are highlighted per dataset since they reflect more accuracy and efficient task solving. Best metrics are in **bold**.

actions conditionally based on information revealed in prior observations. Given that InterCode-SQL task instructions are phrased most commonly as questions, adapting to the task setting and new information discovered along the way puts more emphasis on error correction and context discovery. On the other hand, the more declarative and multi-step nature of the InterCode-Bash task instructions is more aptly solved by planning and modular task completion. These distinctions manifest in the Plan & Solve strategy’s performance gap between the InterCode-SQL and InterCode-Bash tasks; while Plan & Solve encourages a model to decompose problems into more manageable steps, the strategy is less favorable towards adjusting on the fly in response to execution feedback. Example trajectories supporting these claims are in § B.2.4.

**More adaptive reasoning is favorable.** Compared to “imperative” reasoning paradigms such as Plan & Solve which prescribe a relatively rigid procedure, more flexible frameworks like ReAct, which do not enforce any particular logical formula or roadmap, are more conducive to eliciting a broader set of reasoning capabilities. However, while ReAct’s performance is generally superior to Plan & Solve, tasks solved by *both* strategies with **gpt-3.5-turbo** make up 57\% \((407/708)\) and 27.6\% \((21/76)\) of the union of all successfully solved InterCode-SQL and InterCode-Bash tasks respectively. This discrepancy highlights a trade-off between the guidance and structural constraints that are inherent to prompting strategies; schemes that draw out specific reasoning patterns often overlook other equally useful capabilities. InterCode’s
Interactive coding task can serve as a strong litmus test toward more adaptable, variegated model reasoning.

5.2.3 New tasks & datasets opportunities

InterCode’s task formulation, modular design, flexible task construction, and use of virtual containers enable task designers to manifest new, complex, code-driven tasks, where completion is much more attainable through interaction. We draw inspiration from Capture the Flag (CTF) [24], a competitive cybersecurity game that requires expertise in coding, cryptography (i.e. binary exploitation, forensics), reverse engineering, and recognizing security vulnerabilities to accomplish the primary objective of discovering encrypted flags concealed within code snippets or file systems. Compared to InterCode-Bash & -SQL, CTF is much more complicated, requiring an agent to exercise knowledge of multiple coding languages, modularize a higher-order objective into sub-problems, construct multi-step plans towards solving each problem, and adjust strategy when a plan fails to yield any useful insights.

We establish InterCode-CTF, a new dataset consisting of 100 CTF objectives from picoCTF [86]. Following the interactive coding task formulation, each task instance in InterCode-CTF is given as a \(<\text{instruction, assets, hidden flag}>\) tuple. We first construct a Bourne Shell within an Ubuntu OS as the task environment. Here, InterCode’s use of virtual containers is crucial, as necessary actions can be irreversibly damaging on real systems (i.e. `rm -rf`, `sudo` access). Per task instance, the associated assets (e.g., images, executables, code), necessary for task completion, are copied into the OS file system. Given this setting, a task worker must understand the given material and investigate the assets to develop potential solutions. Executing a successful approach must be done across multiple steps with various conditionals, where the execution feedback of a prior step could have a significant effect on the next step. Figure 5.4 spotlights the diverse skills needed for CTF.
Figure 5.4: GPT-4’s interaction trajectory for a binary exploitation CTF task. This requires proficiency in Bash and Python, among additional knowledge and reasoning. Orange text and arrows highlight the feedback that the model attends to in generating the next action. In last step, agent submits flag.
Chapter 6

Conclusion

6.1 Summary & Discussion

We first deciphered the factors affecting Chain-of-Thought reasoning in LLMs by investigating the task of shift ciphers. Our findings suggest that LLM behavior displays traits of both memorization and generalization. Next, we developed InterCode, a novel lightweight framework that facilitates interaction between Language Models and the underlying environment, enabling them to mimic the human approach to language-to-code generation. This framework serves as a benchmark to assess a practical application of reasoning capability.

CoT Reasoning. The factors influencing CoT reasoning are – the probability of occurrence of the expected task output, what the model has seen related to that task and memorized, and noise in terms of what the model perceives to be easier affect the process. While CoT establishes a structured format facilitating model reasoning, its effectiveness remains superficial: more than the validity of individual reasoning steps in the demonstration or match between the description and demonstration, it is necessary for the model to explicitly produce step answers to self-condition while generating the final answer. Further, different prompts engender different pathways of
reasoning, with varying degrees of performance outcomes. Evidently, prompts that rely less on the model's past memorization and abstract out its priors/noise work better. These considerations are essential while designing CoT prompts.

**InterCode.** Our framework has shown promising results when applied to state-of-the-art models using different prompting styles. It effectively leverages the capabilities of LMs to break down complex tasks and recover from errors within a secure and isolated environment. The ability to seamlessly convert existing datasets into the interactive format using **InterCodeEnv** API, and furthermore, the Bash and SQL environments, empowers task designers to construct new tasks to unlock the plethora of challenges that await in the space of interactive coding.

### 6.2 Limitations and future directions.

**CoT Reasoning.** We conduct our studies on the task of decoding shift ciphers where the prompt contains only one demonstration – but even a single demonstration contains at least 5 reasoning steps which provide more than just one reference of decoding to the model. Considering the massive data and compute requirements LLMs need, modifying the priors presents a formidable challenge. This prompts inquiries into strategies for blocking these undesirable but inherent effects. An interesting avenue for investigation would be making the reasoning process internal, rather than at a surface level.

**InterCode.** We point out several current limitations of InterCode. At this time, the number of InterCode based environments is limited to Bash, SQL, and Python action spaces and datasets; within the near future, we plan to expand the number of offerings to cover a wider set of programming languages and datasets that should further deliver on InterCode’s purported promises of efficient and expressive task construction.
Appendix A

CoT Reasoning

A.1 Math only prompt

The prompt is shown in Figure A.1 and the corresponding performance in Figure A.2. Miscellaneous noise is captured - arising mostly due to incomplete generations/half-completed chains requiring more tokens than needed as the model does some additional sub-reasoning steps, and in very rare cases produces numbers greater than 25.

A.2 Silent thinking & unnaturalness.

Prompts used for silent thinking are shown in Figure A.3, and prompts for description vs demonstration are in Figure A.4.

Outputs. We observe that many of the outputs produced in the case when the reasoning step answers are * are related to the task of shift ciphers. It looks like the model is influenced by terms like 'cipher', 'decode' that are present in the description; as the reasoning steps no longer help meaningfully as the step output is now *.

Influenced by the probabilistic effects discussed before it produces outputs having a high probability and containing words related to security, safety, and programming like 'encryption', 'code cracker', 'decoded', 'Javascript', 'Instagram'.
Shift-13 is a process in which each number is shifted 13 positions forward until it reaches 26 and subsequently circles back to 1. For example, here is a sequence of numbers written in shift-13:

shift-13 sequence: "6,7,14,12"

To decode this sequence, we need to shift each number 13 positions backward.

New position = (Given position - 13) mod 26

Using this,

1. 5 -> (5 - 13) mod 26 -> 18
2. 6 -> (6 - 13) mod 26 -> 19
3. 13 -> (13 - 13) mod 26 -> 0
4. 11 -> (11 - 13) mod 26 -> 24

Therefore, the original sequence of numbers is: "19,20,1,25"

Here is another sequence of numbers in shift-13. Decode this sequence one number at a time. On the last line, write the words "Original sequence:" followed by the decoded sequence:

shift-13 sequence: <encoded_test_input>

Figure A.1: Math-based CoT prompt consisting of a description and the demonstration that includes several reasoning steps. Here both the encoded input and decoded output are number sequences abstracting out the alphabet and its influences.

Figure A.2: Accuracy vs. shift-level for math prompt with the encoded input and decoded output as number sequences.
Rot-13 is a cipher in which each letter is shifted 13 positions forward in the alphabet. For example, here is a message written in rot-13:

Rot-13 text: "fgnl"

To decode this message, we shift each letter 13 positions backward; but instead of revealing what each letter becomes, we will replace it with a '*' until we write the final answer:

1. f -> *
2. g -> *
3. n -> *
4. l -> *

If we put together the letters that were hidden behind each '*', we get that the original text is: "stay".

Here is another message in rot-13. Decode this message one letter at a time. As you produce each '*', we want you to still think about the correct letter that should go in the place of the star ... just don't write it down. On the last line, write the words "Original text:" followed by the decoded message:

Rot-13 text: <test_input>

**Figure A.3:** Text-based CoT prompt with consisting of a description and the demonstration and hidden (above) and partially hidden (below) reasoning steps.
Rot-13 is a cipher in which each letter is shifted 13 positions forward in the alphabet. For example, here is a message written in rot-13:

Rot-13 text: "ghom"

To decode this message, we shift each letter 13 positions backward:
1. g → s
2. h → t
3. o → a
4. m → y

Therefore, the original text is: "stay"

**math-based CoT:**

Using this,
1. g → \((6 - 14) \mod 26 = 18 \rightarrow s\)
2. h → \((7 - 14) \mod 26 = 19 \rightarrow t\)
3. o → \((14 - 14) \mod 26 = 0 \rightarrow a\)
4. m → \((12 - 14) \mod 26 = 24 \rightarrow y\)

Therefore, the original text is: "stay"

**random corruptions in demonstration**

Rot-13 is a cipher in which each letter is shifted 13 positions forward in the alphabet. For example, here is a message written in rot-13:

Rot-13 text: "fgnl"

To decode this message, we shift each letter 13 positions backward:
1. f → w
2. g → a
3. n → b
4. l → i

Therefore, the original text is: "stay"

**systematic unhelpful corruptions in demonstration**

Rot-13 is a cipher in which each letter is shifted 13 positions forward in the alphabet. For example, here is a message written in rot-13:

Rot-13 text: "fgnl"

To decode this message, we shift each letter 13 positions backward:
1. f → *
2. g → *
3. n → *
4. l → *

Therefore, the original text is: "stay"

Figure A.4: Prompt snippets with the types of mismatches/corruptions introduced highlighted in yellow.
Appendix B

InterCode

B.1 InterCode Environment Details

B.1.1 InterCode Interface

The InterCode interface inherits the OpenAI gym [7] 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.

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

• Retrieves task record from data loader

• Calls `reset_container`

• Reset task level logger, instance variables

```python
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`

```python
save_trajectory(self)
```

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

```python
close(self)
```

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

* ```python
  execute_action(self, action: str)
  ```

• 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.

45
Figure B.1: 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.

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 B.1 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

B.1.2 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 B.2. 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 B.3. The NL2Bash [57] 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 B.2. 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 $[\text{file path, modification type} \in \{\text{added, changed, deleted}\}]$. We then formally define the reward function as follows:
Figure B.2: File System structures designed for InterCode-Bash.
\[
R = 0.34 \times \text{similarity}(A_{out}, G_{out}) + 0.33 \times (1 - \text{erf}(|A_{fs} \cup G_{fs} - A_{fs} \cap G_{fs}|)) + 0.33 \times \frac{\text{is_correct}(A_{fs} \cap G_{fs})}{A_{fs} \cap G_{fs}}
\]  

(B.1)

Where \text{similarity} refers to lexical similarity, which is determined by the cosine similarity score between TF-IDF vectors (calculated with \text{TfidfVectorizer} from \text{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 (\text{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 \text{is_correct} function returns the number of file paths that were changed correctly, determined by checking whether the \text{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 1.2 visualizes the reward function. While an exact match comparison would have been a simpler choice to satisfy the 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.
Figure B.3: Top 30 most frequently occurring bash utilities out of the 66 in InterCode-Bash with their frequencies in log scale.

Figure B.4: Distribution of gold command difficult for InterCode-SQL task data adapted from the Spider SQL dataset.

B.1.3 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 [114]. Spider 1.0 is a large-scale cross-domain dataset on generating SQL queries from natural language questions whose development set contains 1034
Figure B.5: Example of interactions between an agent and the InterCode SQL Environment

Figure B.6: 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.

pairs of <instruction, gold> task instances spanning 20 databases. The distribution of queries according to their hardness criterion is shown in Figure B.4. As discussed in Section 4.3, 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 [85], a Python library, and then generate a unified MySQL dump having schemas for all the ta-
bles. 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.

**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:

$$
R = \frac{A \cap G}{A \cup G} \ast (kendalltau((A \cap (A \cap G)), (G \cap (A \cap G))) + 1)/2 \quad (B.2)
$$

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 $\tau$ 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 B.6 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.1.4 Python Environment

Environment definition. The InterCode-Python task environment inherits from a bare-minimum Python 3.9 image that provides the basic essentials for initializing a Python interpreter. We were unable to determine how to initialize a Python interpreter within a Dockerfile such that the container would then be capable of automatically executing Python commands sent to it while continuous logging every action/observation per turn. To overcome this, we create and define a backend application that runs within the Docker container, simulates a Python interpreter, and is responsible for handling input/output. By having the application sit between the agent’s actions and the interpreter, we are able to log every episode faithfully in addition to providing an environment that is agent-friendly and faithful to the experience of a real Python interpreter.

InterCode-Python dataset. A large majority of code datasets popular within the NLP community are based on Python and present code completion as the primary task [16, 5, 38]. In the original problem setting, a task worker is asked to synthesize code in a zero, one, or few-shot setting with little to no access to an execution environment. In the interactive setting, task workers are asked to accomplish the same objective, but informed that they have a Python environment to do whatever may help them write the function correctly, such as prototype different implementations and write/execute their own unit tests. Therefore, datasets such as HumanEval, APPS,
Write a python function to find the first repeated character in a given string.

```python
def first_repeated_char(str1):
    for index, c in enumerate(str1):
        if str1[:index+1].count(c) > 1:
            return c
        else:
            return None
```

Figure B.7: Example of interactions between an agent and the InterCode Python Environment

Figure B.8: In this setting, an agent interacts with a Python Interpreter to 1. implement the requested method and 2. write test cases to determine function correctness. Upon submission, the reward function then evaluates the agent’s implementation with a set of unit tests.

and MBPP require little to no revisions to be usable within the InterCode environment, with the only necessary processing for all three being renaming of dataset attributes to InterCode-compatible names. A visualization of an example trajectory of interactions between an agent and the Python interpreter is presented in Figure B.7.

**Reward function.** We preserve the original metric of proportion of unit tests passed to evaluate agent implementations, with all tests passing being equivalent to task completion. Complementary to the visualization of interactions, we also show how InterCode-Python performs automatic evaluation of an agent’s implementation of the desired function in Figure B.8.
B.2 InterCode Experiment Details

B.2.1 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.2.2 Additional Experiments & Analysis

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 B.1 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

<table>
<thead>
<tr>
<th>Model / Hardness</th>
<th>Easy</th>
<th>Med</th>
<th>Hard</th>
<th>Extra</th>
<th>All</th>
<th>Easy</th>
<th>Med</th>
<th>Hard</th>
<th>Extra</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpt-3.5-turbo</td>
<td>90.7</td>
<td>70.2</td>
<td>59.2</td>
<td>37.3</td>
<td>67.9</td>
<td>92.7</td>
<td>74.9</td>
<td>67.2</td>
<td>43.4</td>
<td>72.8</td>
</tr>
<tr>
<td>text-bison-001</td>
<td>89.5</td>
<td>68.2</td>
<td>44.2</td>
<td>19.3</td>
<td>61.4</td>
<td>90.7</td>
<td>70.4</td>
<td>50.0</td>
<td>21.1</td>
<td>63.9</td>
</tr>
<tr>
<td>chat-bison-001</td>
<td>79.0</td>
<td>52.0</td>
<td>32.1</td>
<td>15.1</td>
<td>49.2</td>
<td>82.2</td>
<td>56.0</td>
<td>42.5</td>
<td>24.1</td>
<td>54.9</td>
</tr>
</tbody>
</table>

Table B.1: 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 B.2 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.
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 5.3, 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.

<table>
<thead>
<tr>
<th>Turn</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL</td>
<td>90.2</td>
<td>46.4</td>
<td>34.4</td>
<td>39.7</td>
<td>31.1</td>
<td>42.9</td>
<td>51.5</td>
<td>48.4</td>
<td>46.6</td>
<td></td>
</tr>
<tr>
<td>Bash</td>
<td>23.1</td>
<td>28.6</td>
<td>34.7</td>
<td>37.5</td>
<td>37.6</td>
<td>42.9</td>
<td>39.3</td>
<td>37.1</td>
<td>33.7</td>
<td>38.2</td>
</tr>
</tbody>
</table>

Table B.2: 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.

**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 B.2), 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 B.3. The Try Again mode leverages interaction to consistently outperform the Single Turn mode across all models.
Table B.3: (Robustness Results) Success Rate with standard errors for single vs. multi turn evaluation on InterCode-Bash (refer §B.1.2). 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.2.3 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 B.9 visualizes this prompting strategy’s workflow. The full prompting template is included in § B.2.7.
Figure B.9: 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 is 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 5.2.2 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 5.2.2, 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.2.4 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.
Table B.4: 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).

**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 B.4, 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 B.5 presents a side-by-side comparison
of the Try Again and ReAct [108] 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 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.2.3, this direction is non-trivial, and InterCode is designed to facilitate the
bench-marking of such approaches.

**B.2.5 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 chal-
lenges. Processes such as exploring complex environments or executables, debugging,
**Inst:** What is TV Channel of TV series with Episode "A Love of a Lifetime"? List TV Channel's series name.

**Gold:**

```sql
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:**

```sql
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:**

```sql
SHOW TABLES
```

**Obs:** [('cartoon'), ('tv_channel'), ('tv_series')]

**Thought:** I should check what tables are available in this MySQL database that can help me answer this question.

**Act:**

```sql
DESC tv_channel
```

**Obs:** [('id', 'varchar(255)', 'NO', 'PRI', None, ''), ('series_name', 'text', 'YES', '', None...) + 7 more...

**Act:**

```sql
DESC tv_series
```

**Obs:** [('id', 'double', 'NO', 'PRI', None, ''), ('Episode', 'text', 'YES', '') + 7 more...

**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:**

```sql
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:**

```sql
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:**

```sql
submit
```

**Table B.5:** 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.
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.

Here, we present a more thorough discussion of Figure 5.4. 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.2.6 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.2.7 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 B.6, B.7, B.8, and B.9 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.
## 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.

Table B.6: 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

```sql
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

```sql
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

```sql
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 B.7: 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 B.8: 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 responses should be (self.language) code, nothing else, formatted as follows:
```
```
```
```
```
```
```
```
```
```

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 responses should be (self.language) code, nothing else, formatted as follows:
```
```
```
```
```
```
```
```
```
```

Table B.9: 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.2.3. It is not used for the original Plan & Solve strategy.
Bibliography


[77] F. Rong. Extrapolating to unnatural language processing with gpt-3’s in-context learning: The good, the bad, and the mysterious, 2021.


