

# Evaluating the Generalization Capabilities of Large Language Models on Code Reasoning

Rem Yang  
MIT CSAIL  
remyang@csail.mit.edu

Julian Dai  
Brown University  
julian\_dai@brown.edu

Nikos Vasilakis  
Brown University  
nikos@vasilak.is

Martin Rinard  
MIT CSAIL  
rinard@csail.mit.edu

## Abstract

We assess how the code reasoning abilities of large language models (LLMs) generalize to different kinds of programs. We present techniques for obtaining in- and out-of-distribution programs with different characteristics: code sampled from a domain-specific language, code automatically generated by an LLM, code collected from competitive programming contests, and mutated versions of these programs. We also present an experimental methodology for evaluating LLM generalization by comparing their performance on these programs. We perform an extensive evaluation across 10 state-of-the-art models from the past year, obtaining insights into their generalization capabilities over time and across different classes of programs. Our results highlight that while earlier models exhibit behavior consistent with pattern matching, the latest models exhibit strong generalization abilities on code reasoning.

## 1 Introduction

Large language models (LLMs) are increasingly used in software engineering and have demonstrated high performance on a variety of code-related benchmarks (Austin et al., 2021; Chen et al., 2021; Gong et al., 2024; Jain et al., 2025; Jimenez et al., 2024; Liu et al., 2023). However, the extent to which these models can effectively reason about program behavior remains an open question. Investigating LLM performance on code reasoning tasks (Chen et al., 2025; Gu et al., 2024; Jain et al., 2025; Liu et al., 2024) is therefore of critical importance, especially since the ability to reason about programs has been shown to enhance the performance of LLMs on code generation (Ni et al., 2023), program repair (Ni et al., 2024), and more general tasks that involve logical, mathematical, scientific, and commonsense reasoning (Li et al., 2025).

A key challenge in evaluating LLMs on coding tasks is distinguishing generalization from memorization and/or pattern matching across different classes of programs. Previous works (Dong et al., 2024; Riddell et al., 2024) have shown that HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), two of the most commonly used code generation benchmarks, contain a large overlap with LLM training data. On competitive programming benchmarks, LLMs exhibit a drop in performance on problems after their knowledge cutoff date, likely due to data contamination (Huang et al., 2024; Jain et al., 2025; Roberts et al., 2024). However, understanding the behavior of LLMs on different distributions of programs and how to effectively measure their generalization abilities has not yet been extensively studied for code reasoning.

To address this gap, we systematically evaluate LLM code reasoning performance on in- vs. out-of-distribution programs. We propose two general methods for obtaining new programs that are likely out-of-distribution: (1) sampling programs from a domain-specific language (DSL) with a combinatorially large space of programs and (2) applying mutation operators (Jia and Harman, 2010; Papadakis et al., 2019).

These mutation operators apply small syntactic changes that (approximately) preserve the complexity of a given program while changing its semantics.

We create three datasets containing different classes of programs: (1) **LLM-List**: common list-processing functions (e.g., sort, search) generated by an LLM (which we expect to be in-distribution); (2) **DSL-List**: list-processing functions sampled from a DSL (which we expect to be out-of-distribution); and (3) **LeetCode**: human-submitted LeetCode contest solutions, which we separate into those before and after the knowledge cutoffs for the LLMs we evaluate. For each dataset, we further obtain mutated versions of its programs.

We perform two sets of experiments on these datasets:

1. **Execution Prediction.** These experiments present an LLM with a program and a test input and instruct it to predict the output. We measure whether a model correctly predicts the outputs of the original and mutated programs. We also introduce the novel metric of *reversion*, which measures whether a model predicts the output of the other version of its given program; e.g., reversion on a mutated program means that a model predicts the output of its corresponding original program.

Substantially higher correctness on original programs than on mutated programs and high reversion on mutated programs are consistent with (1) original programs being in-distribution, (2) mutated programs being out-of-distribution, and (3) a model using pattern matching to predict program outputs. Comparable performance on original and mutated programs with little reversion is consistent with either both sets of programs coming from similar distributions or a model generalizing to out-of-distribution programs. To distinguish these cases, we design a second set of experiments.

2. **Execution Choice.** These experiments present an LLM with both the original and mutated versions of a program and instruct it to (1) identify the version it is more confident in reasoning about and (2) predict the output for that program. We emphasize that we present the two programs without mentioning the existence of mutations. We measure LLM preference toward choosing the original or mutated program and its correctness and reversion when reasoning about the chosen program.

The key insight is that a consistent preference for selecting the original programs is consistent with them being in-distribution and mutation moving them out-of-distribution; selecting the original and mutated programs at a similar rate indicates that the two versions’ distributions are similar.

**Empirical Findings.** We perform an extensive evaluation on a suite of state-of-the-art open- and closed-access models released between November 2023 and March 2025, spanning the latest reasoning models — QwQ (Qwen Team, 2025), DeepSeek-R1 (Guo et al., 2025), and o3-mini (OpenAI, 2025) — and more traditional LLMs — DeepSeek-Coder (Guo et al., 2024), Qwen2.5-Coder (Yang et al., 2024a), GPT-4o-mini (OpenAI, 2024), and GPT-4o (Hurst et al., 2024). We summarize our main empirical findings below.

1. **How has LLM generalization capability on code reasoning changed over time?** Our results indicate a substantial increase in LLM generalization on code reasoning over the past year. Earlier LLMs from a little over a year ago exhibit poor correctness on all problems except original LLM-List programs; in stark contrast, the most recently released reasoning LLMs exhibit near-perfect correctness on all problems.
2. **How does LLM performance vary across different classes of programs?** On LLM-List, original correctness is high for all models; mutated correctness is low and reversion is high for earlier models, while the latest models obtain near-perfect mutated correctness and near-zero reversion. On DSL-List, original and mutated correctness is comparable and reversion is low for each model; earlier models achieve poor performance while the latest models are near perfect. On LeetCode, earlier LLMs exhibit a noticeable gap in correctness between problems before and after cutoff, while there is essentially no gap for the latest reasoning LLMs; correctness on mutated problems compared to original problems is much lower (and reversion is high) for earlier models, while later models exhibit only a small drop in correctness with low reversion. We also further examine these results as a function of program complexity in our evaluation.

These results are consistent with LLM-List programs being in-distribution, DSL-List programs being out-of-distribution, LeetCode programs being somewhat in-distribution, and mutation moving in-distribution programs out-of-distribution. They are also consistent with earlier models relying on pattern matching and the latest models generalizing to reason successfully for all of our benchmark programs. The results also underline that LLM code reasoning performance on substantially in-distribution problems is not representative of their generalization abilities, and that DSL-sampling and mutation are effective ways to obtain programs that better measure generalization capabilities.

Furthermore, the diminishing performance gap between before- and after-cutoff LeetCode problems as LLMs become more capable highlights that the technique of evaluating LLMs’ code reasoning abilities on problems after cutoff may not be sufficient for creating out-of-distribution problems; our hypothesis is that human-written code, whether produced before or after knowledge cutoff, contains common programming patterns.

- 3. Can LLMs distinguish between original and mutated programs, and how does choosing to reason about the original or mutated programs affect their performance?** On LLM-List, all models prefer original programs to mutated programs and achieve high correctness and minimal reversion when reasoning about the original programs. For the (relatively few) cases where the models choose the mutated version, correctness is poor and reversion is substantial (with the exception of GPT-4o and the latest reasoning LLMs). On DSL-List, all models exhibit little preference for original versus mutated versions and achieve comparable correctness for both versions when selected. On LeetCode, all models prefer reasoning about the original programs. Correctness (on both selected original and mutated versions) generally improves as LLM release date increases; reversion when reasoning about the mutated version is high for most traditional models, while being nonexistent for reasoning LLMs.

These results are consistent with LLM-List programs being in-distribution for all models (including the latest ones) and mutation moving in-distribution programs out-of-distribution (while not affecting the distribution of already out-of-distribution programs) for all models. They also further highlight the reliance of earlier models on pattern matching, while accentuating the generalization abilities of the latest reasoning LLMs on code reasoning.

## 2 Related Work

### 2.1 Data Contamination and Memorization

HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) are two of the most widely used evaluation benchmarks for code LLMs, designed to test their ability to synthesize relatively simple Python functions given a natural language description. However, these benchmarks have been found to suffer from severe data contamination issues, with a large portion of their solutions (or similar code) seen during training for many LLMs (Dong et al., 2024; Riddell et al., 2024). Furthermore, Huang et al. (2024) and Roberts et al. (2024) show that while competitive programming questions are effective at evaluating LLMs, there is a large disparity between the performance of models before and after their knowledge cutoffs. These concerns have motivated the development of live benchmarks (Jain et al., 2025; Jimenez et al., 2024; White et al., 2025). These benchmarks aim to bypass data contamination by continually updating their set of problems from a given data source, and by evaluating LLMs on problems released after their knowledge cutoffs. For instance, Jain et al. (2025) continually draws competitive programming questions from sources such as LeetCode and Codeforces contests. Another line of work reveals that LLMs memorize large amounts of code and are prone to pattern matching (Carlini et al., 2021; Rabin et al., 2023; Yang et al., 2024c). These findings further highlight the importance of evaluating LLM performance on out-of-distribution scenarios that test their genuine reasoning abilities. Our work addresses data contamination issues by using DSL sampling and program mutation as mechanisms to create out-of-distribution problems with which to evaluate LLM generalization. Furthermore, contrary to prior work, our evaluation highlights that on the task of code reasoning, (1) evaluating LLMs on problems after their cutoff date may not be sufficient for testing their

generalization abilities and (2) recent reasoning models possess surprisingly strong generalization abilities, providing evidence that LLMs are capable of going beyond memorizing or performing surface-level pattern matching on their training data.

## 2.2 LLM Reasoning Capabilities in Other Domains

With the rapid rise in LLM performance across many domains, recent research has studied how well the apparent reasoning abilities of LLMs generalize. [Mirzadeh et al. \(2025\)](#) study LLM reasoning on math problems when names and numerical values in the problems are altered and when irrelevant facts are introduced; they find that LLMs are brittle, with performance varying significantly in response to these perturbations. Similarly, [Jiang et al. \(2024\)](#) show that LLMs’ reasoning processes exhibit token bias: their performance on logical puzzles is affected by entities (e.g., names and numbers) not relevant to the underlying logic required to solve the problem. [Qi et al. \(2025\)](#) investigate how an LLM’s ability to solve algorithmic tasks differs between in-distribution and out-of-distribution input values; however, this work does not evaluate LLMs on code, and their tasks are instead phrased in natural language. These works start with a fixed underlying structure that solves a given problem, then evaluate the ability of LLMs to ignore irrelevant perturbations or augmentations to this structure. In contrast, the mutations that we apply are directly relevant to the reasoning task because they change the correct result. Here, we evaluate the ability of LLMs to compose their reasoning process over parts of the task (program statements in this case) in new ways, as opposed to identifying and ignoring irrelevant information.

## 2.3 Robustness of Code LLMs

Testing the robustness of code LLMs to semantics-preserving transformations is a widely studied problem in the literature ([Gao et al., 2023](#); [Hooda et al., 2024](#); [Sarker et al., 2024](#); [Wang et al., 2023](#); [Yang et al., 2024b](#)). These works utilize techniques such as renaming variables, permuting statements without dependencies, branch rewriting, and adding dead code to evaluate how an LLM’s performance is affected by these changes, which are designed specifically to preserve the semantics of the code. Therefore, this line of research measures how LLMs generalize to new representations of the same functionality. In contrast, our mutation mechanism modifies programs with the explicit goal of changing semantics while preserving overall code complexity, enabling research that measures generalization to new functionality expressed with similar complexity. Most previous robustness works also focus on code generation, while we focus on the task of code reasoning.

## 2.4 Evaluating Different Aspects of Code Reasoning

Researchers have proposed multiple benchmarks to measure the ability of LLMs to reason about different aspects of program behavior. [Gu et al. \(2024\)](#) evaluate the code execution capabilities of LLMs on a dataset comprising simple Python functions generated by Code Llama ([Roziere et al., 2023](#)). CodeMind ([Liu et al., 2024](#)) augments the previous work’s framework to obtain a benchmark with additional programming languages and tasks such as dependent execution and specification reasoning. REval ([Chen et al., 2025](#)) proposes to not only measure LLMs’ abilities to predict program output, but also intermediate states of the program’s execution. Our work is orthogonal and complementary to this line of research. We benchmark code execution as it is the most fundamental code reasoning task, but our approach of obtaining new programs (i.e., DSL sampling, mutation) and comparing performance across different distributions of code is directly applicable to other settings. [Li and Shin \(2024\)](#) use mutation testing to explore whether LLMs can detect inconsistencies between a program’s natural language description and mutated code. This approach tests if LLMs can match natural language with a program, but the goal is not to determine whether LLMs can actually reason about code behavior.

## 3 Obtaining New Programs

### 3.1 DSL Sampling

Our first approach to acquiring programs for benchmarking LLM generalization in code reasoning is to (1) define a functional DSL and a set of syntactic constraints and (2) sample programs from the context-free grammar (CFG) compiled from the DSL and constraints. We describe this procedure in detail below. A DSL is defined by a set of primitives, together with the type and semantics of each primitive. Syntactic constraints include a diverse set of properties that one may specify to limit the space of possible programs, e.g., maximum depth (of the abstract syntax tree) and the type of the program. Together, the DSL and the set of syntactic constraints are compiled to a CFG. For details on this procedure, we refer the reader to Fijalkow et al. (2022). To sample a program, we first assign a probability to each production in the CFG. Starting from the initial non-terminal node, we sample a production according to the probability distribution at that node; then, we recursively sample productions until reaching a terminal node. Finally, a sampled program can be converted into any common programming language with a transpiler.

We note two properties about this technique. First, since the resulting CFG defines a combinatorially large space of programs, it is unlikely for any sampled program to have been in the training data. Second, the difficulty of the sampled programs can be controlled by varying the syntactic constraints, e.g., program depth and number of function parameters.

Concretely, we construct a dataset by implementing a list-processing DSL containing common list transformations, conditionals, and maps, then translating sampled programs into imperative Python code. We focus on list-processing programs because they comprise an important class of widely used functions. We present details on this dataset in Section 4.1.

### 3.2 Program Mutation

---

**Algorithm 1** Mutated Dataset Creation

---

**Input:** Original dataset  $D$   
**Output:** Mutated dataset  $D'$

```
1: procedure MUTATEPROGRAMS( $D$ )
2:    $D' \leftarrow \emptyset$ 
3:   for  $(P, x) \in D$  do
4:      $S \leftarrow \text{Mutate}(P)$  ▷ Generate all mutants
5:     for  $P' \in S$  do ▷ Remove invalid mutants
6:       if not  $(\text{IsExecutable}(P', x) \text{ and } \text{HasDifferentOutputs}(P, P', x))$  then
7:          $S \leftarrow S \setminus \{P'\}$ 
8:       end if
9:     end for
10:    if  $S \neq \emptyset$  then ▷ Select a mutant
11:       $P' \leftarrow \text{MostSimilarCoverage}(P, S, x)$ 
12:       $D' \leftarrow D' \cup \{(P', x)\}$ 
13:    end if
14:  end for
15:  return  $D'$ 
16: end procedure
```

---

Mutation testing (Jia and Harman, 2010; Papadakis et al., 2019) applies local *mutation operators* to programs. Originally developed to assess test suite quality, we use mutation to obtain new programs with different behaviors as a means to evaluate LLM generalization in code reasoning. At a high level, for each program-input pair in a given (original) dataset, we create a mutant that executes without error and produces a different output from the original program.

```

def binary_search(lst, value):
    left, right = 0, len(lst) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if lst[mid] == value:
            return mid
        elif lst[mid] < value:
            left = mid + 1
        else:
            right = mid - 1
    return -1

def f1(a1, a2):
    v1 = []
    a2.pop(0)
    v1.append(a1)
    a2.pop(0)
    for i in range(len(v1)):
        v1[i] = v1[i][-1]
    if a1[0] > 1:
        v2 = v1
    else:
        v2 = a2
    return v2

def minimumOperations(nums: List[int]) -> int:
    cnt = defaultdict(int)
    for x in nums:
        cnt[x] += 1
    ans = p = 0
    while any(x > 1 for x in cnt.values()):
        for i in range(3):
            if p < len(nums):
                cnt[nums[p]] -= 1
                p += 1
            ans += 1
    return ans

```

Figure 1: Example programs from the LLM-List (left), DSL-List (center), and LeetCode (right) datasets.

Algorithm 1 presents our mutation procedure. This procedure takes as input a dataset  $D$  consisting of tuples  $(P, x)$ , where  $P$  is a program and  $x$  is an input to the program. The algorithm loops over each program-input pair  $(P, x) \in D$  (Lines 3–14) and performs the following steps in each iteration. First, it modifies  $P$  by applying the mutation operators shown in Table 1. It tries all mutations separately, producing a set  $S$  consisting of all programs  $P$  with one mutation applied (Line 4). Then, it filters  $S$  so that  $S$  only consists of programs that (1) execute without error on the input  $x$  and (2) produce a different output than the original program, i.e.,  $P(x) \neq P'(x)$  (Lines 5–9). If  $S$  is now empty, it skips to the next iteration without adding to  $D'$ . Otherwise, it selects the mutated program  $P'$  that has the most similar line coverage on  $x$  compared to  $P$ ; if there are multiple, it selects one uniformly at random (Line 11). This check ensures that the mutation approximately preserves the complexity of the problem and does not make the program trivial (e.g., returning on the second line after a condition on the first line becomes always true). This program  $P'$  and the input  $x$  are then added to  $D'$  (Line 12). Some problems in  $D$  may not have any mutant that results in valid code that runs without error and produces a different output (i.e., those with empty  $S$  on Line 10). In this case, we remove these problems from the original dataset  $D$  to maintain a one-to-one correspondence between the problems in the original and mutated datasets. We apply the mutation procedure described above to each of the three datasets used in our evaluation (Section 4).

Table 1: Mutation operators and examples.

Mutation Type	Example
Arithmetic Operator	$a + b \rightarrow a - b$
Relational Operator	$a < b \rightarrow a \leq b$
Logical Operator	$a \text{ and } b \rightarrow a \text{ or } b$
Keyword	<code>continue</code> $\rightarrow$ <code>break</code>
Numerical Literal	$1 \rightarrow 0$

## 4 Code Reasoning Benchmark Sets

We describe how we create the three benchmark sets used in our evaluation. Each dataset comprises a collection of program-input pairs, with the ground truth being the corresponding output. We ensure that each program is deterministic and thus has only one possible output. Fig. 1 shows an example program from each of our datasets. Fig. 6 in Appendix A.3 contains extra examples of problems from each dataset (along with their mutated versions).

### 4.1 DSL-List

We define a list-processing DSL with the primitives and their associated types shown in Fig. 2. The semantics of each primitive is standard and corresponds to their naming. We choose these primitives because they are expressive enough to represent a rich class of list transformations while being sufficiently simple so that sampled programs are likely to be valid (i.e., do not produce runtime errors). We apply a variety of constraints designed to exclude trivial constructs from the sampled programs (e.g., expressions like  $0 == 0$ ); these are detailed in Appendix A.1.1. To include programs of different types and sizes in our dataset, we

<b>if:</b> $\text{bool} \rightarrow \text{t0} \rightarrow \text{t0} \rightarrow \text{t0}$	<b>init:</b> $\text{L}(\text{t0}) \rightarrow \text{L}(\text{t0})$	<b>&lt;:</b> $\text{int} \rightarrow \text{int} \rightarrow \text{bool}$
<b>map:</b> $(\text{t0} \rightarrow \text{t1}) \rightarrow \text{L}(\text{t0}) \rightarrow \text{L}(\text{t1})$	<b>tail:</b> $\text{L}(\text{t0}) \rightarrow \text{L}(\text{t0})$	<b>&gt;:</b> $\text{int} \rightarrow \text{int} \rightarrow \text{bool}$
<b>empty:</b> $\text{L}(\text{t0})$	<b>length:</b> $\text{L}(\text{t0}) \rightarrow \text{int}$	<b>&amp;&amp;:</b> $\text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$
<b>append:</b> $\text{t0} \rightarrow \text{L}(\text{t0}) \rightarrow \text{L}(\text{t0})$	<b>index:</b> $\text{int} \rightarrow \text{L}(\text{t0}) \rightarrow \text{t0}$	<b>  :</b> $\text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$
<b>extend:</b> $\text{L}(\text{t0}) \rightarrow \text{L}(\text{t0}) \rightarrow \text{L}(\text{t0})$	<b>==:</b> $\text{int} \rightarrow \text{int} \rightarrow \text{bool}$	<b>!:</b> $\text{bool} \rightarrow \text{bool}$

Figure 2: Domain-specific language for the DSL-List dataset.  $\text{L}$  denotes `List`, and  $\text{t0}$  and  $\text{t1}$  are polymorphic types. We allow integers in the range  $[-1, 5]$ .

specify the programs to be of type  $\text{List}(\text{int}) \rightarrow \text{List}(\text{int})$  or  $\text{List}(\text{int}) \rightarrow \text{List}(\text{int}) \rightarrow \text{List}(\text{int})$  (i.e., list transformations that take in one or two inputs) and to be of depth 4 or 5. We assign probabilities to the productions in the compiled CFG so that `if` and `map` have a weight of 5, `extend` has a weight of 0.05, and all other primitives have a weight of 1. We implement our language and sampler in Python by adapting the framework from Fijalkow et al. (2022). We translate each sampled DSL program to imperative Python code using Algorithm 2 in Appendix A.1.2.

When sampling each program, we also sample three inputs according to the program’s type signature. Lists in these inputs consist of three to five elements, with each element being an integer in  $[0, 5]$ ; the list lengths and elements are uniformly sampled with replacement. We consider a program to be valid if it executes without error on all three inputs.

To create our dataset, we sample 1000 valid programs for each combination of program type and program depth. For each type signature, we select a random subset of 50 programs such that there are 10 programs each when binning by lines of code from 4 to 24 (with a bin width of 4). In total, we obtain 100 programs with three inputs for each program.

## 4.2 LLM-List

We obtain the LLM-List dataset in a three-step process: brainstorming, code generation, and input generation. All prompts described below are shown in Appendix A.2.

We first use GPT-4o to brainstorm 100 common list functions in Python. Specifically, we instruct the model to come up with functions that take in a list of integers as one of the parameters and do not contain random operations, substantial floating-point operations, or additional imports. The model produces a numbered list of function headers (e.g., `remove(lst, value)`) and their natural language descriptions. Examples of brainstormed operations in this phase include `merge`, `slice`, and `intersection`. Afterward, we add manually specified headers and descriptions for ten sorting algorithms and two search algorithms to the brainstormed list, obtaining a collection of function headers and descriptions for 112 common list operations.

Next, we instruct GPT-4o to generate the code for the brainstormed operations. We provide the model with each function’s header and description, and instruct it to implement a deterministic function with limited usage of built-ins, while ensuring the logic in the function is as explicit as possible. The model generates well-structured, readable programs with meaningful variable names.

The generated list programs have a variety of type signatures. We use GPT-4o to generate three test inputs for each program and use these inputs in our dataset. The prompt specifies that each input should produce no errors when executed, should not include floating-point numbers, and that lists should only contain a few elements, but not be empty. We execute the generated inputs and regenerate them in case any of these constraints are violated. In total, we obtain 112 programs with three inputs for each program.

## 4.3 LeetCode

LeetCode is a dataset of human-submitted solutions to competitive programming problems that are collected from the LeetCode contest website (LeetCode). LiveCodeBench (Jain et al., 2025) also used LeetCode contest solutions to evaluate LLMs on code execution prediction. However, their dataset only covers the time period between May 2023 and November 2023 (inclusive), which is insufficient for evaluating newer models whose

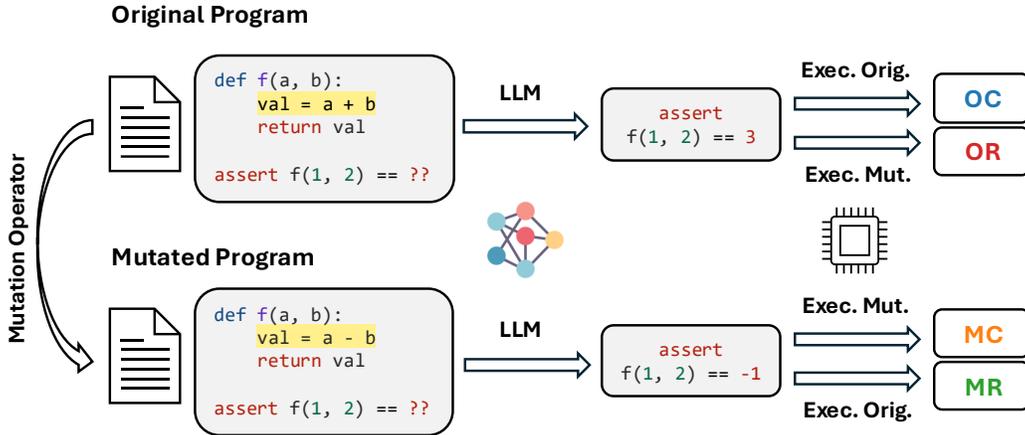


Figure 3: Overview of the execution prediction experiment. Given an original program  $P$ , its mutated version  $P'$ , and an input  $x$ , we instruct an LLM to predict the outputs of the original program  $P(x)$  and the mutated program  $P'(x)$ . For the original program, we check if the model prediction correctly matches  $P(x)$  (OC) or incorrectly matches  $P'(x)$  (OR). For the mutated program, we check if the model prediction correctly matches  $P'(x)$  (MC) or incorrectly matches  $P(x)$  (MR).

knowledge cutoffs are more recent. Our dataset consists of solutions submitted between May 2023 and January 2025 (inclusive). We divide this dataset into two halves: one containing programs from May 2023 to August 2023 (inclusive), before the knowledge cutoffs of all the LLMs we consider; and one containing programs during or after August 2024, after the cutoffs of all the LLMs we consider.

LeetCode runs six contests a month (including weekly and biweekly contests), and each contest has four questions. Consistent with LiveCodeBench, we select the subset of these questions that overlap with their code generation dataset. For each question, we collect up to the top 20 ranked solutions written in Python3. We exclude solutions that do not pass all of the public test cases and deduplicate solutions with identical abstract syntax trees. We also apply several compile-time and runtime filters to ensure that problems in our dataset are reasonable. We exclude problems whose code is outside the range of 100 to 800 characters or have more than 1000 bytecode operations (excluding imports) and apply other filters identical to LiveCodeBench. We note that LiveCodeBench excluded problems whose code is outside the range of 100 to 500 characters or have more than approximately 500 bytecode operations (excluding imports). We increased the difficulty of our problems because of the increased code reasoning capabilities of more recent LLMs. To ensure that our dataset contains a diverse set of problems while remaining manageable in size, for each question, we include up to three solutions (each paired with an input from the public test cases) that are selected uniformly at random. In total, we assemble 184 and 190 problems before and after cutoff, respectively.

## 5 Experimental Methodology

We detail the two sets of experiments that we run, followed by a discussion of our experimental setup.

### 5.1 Execution Prediction

Our first set of experiments measures the ability of an LLM to, given a program and an input, correctly predict the output of the program when executed on the input. To instruct the models on this task, we adapt the prompt from Jain et al. (2025). For models not specifically trained for reasoning (DeepSeek-Coder, Qwen2.5-Coder, and GPT models), we utilize chain-of-thought prompting (Wei et al., 2022) along with few-shot prompting (Brown et al., 2020) to maximize the code reasoning capabilities of these models. In line

with best practices, we do not provide any examples (i.e., use a zero-shot setting) for the reasoning models to allow them maximum flexibility in their reasoning chain. Our complete prompts are in Appendix B.2.

Fig. 3 provides an overview of our experimental procedure. Each problem (i.e., program-input pair) in each dataset has both a version with the original program  $P$  and another with the mutated program  $P'$ . Given an input  $x$ , we instruct the model to predict the outputs of the original and mutated programs in separate inference passes. Then, we measure:

- **Original Correctness (OC):** Whether the LLM predicts  $P(x)$  when given the original program.
- **Original Reversion (OR):** Whether the LLM reverts to (i.e., incorrectly predicts)  $P'(x)$  when given the original program.
- **Mutation Correctness (MC):** Whether the LLM predicts  $P'(x)$  when given the mutated program.
- **Mutation Reversion (MR):** Whether the LLM reverts to (i.e., incorrectly predicts)  $P(x)$  when given the mutated program.

Recall in Section 3.2 that we ensure  $P(x) \neq P'(x)$  by construction; thus, reversion always indicates an incorrect prediction.

## 5.2 Execution Choice

We perform a second set of experiments, which measures the ability of an LLM to, given two programs and an input, first pick a program, then correctly predict its output when executed on the input. For each problem, we provide the original program and its corresponding mutated program as the two programs. We specifically instruct the model to pick whichever program it is more confident in reasoning about, without providing any information on the existence of mutations. The prompts we use are shown in Appendix B.3. Then, we measure:

- **Preference:** Whether the LLM chooses to reason about the original or the mutated program.
- **Correctness:** Whether the LLM correctly predicts the output for the program it chooses.
- **Reversion:** Whether the LLM predicts the output for the other program (that it does not choose).

## 5.3 Experimental Setup

### 5.3.1 Benchmarked Large Language Models

We benchmark a representative set of 10 large language models that includes state-of-the-art open-access and closed-access models over the past year. These models cover five model families, different model sizes within the same family, and both traditional and recent reasoning LLMs. Our open-access models include DeepSeek-Coder- $\{33, 7\}$ B (state-of-the-art at the beginning of 2024) and Qwen2.5-Coder- $\{32, 14, 7\}$ B (state-of-the-art toward the end of 2024), as well as QwQ-32B and DeepSeek-R1, the newest class of reasoning models trained with reinforcement learning to perform extensive chain-of-thought reasoning. We use the instruction-tuned versions of DeepSeek-Coder and Qwen2.5-Coder. Our closed-access LLMs include OpenAI models that span traditional and new reasoning models: GPT-4o-mini, GPT-4o, and o3-mini. Table 2 presents an overview of these models, including their approximate knowledge cutoff and release dates. The cutoff date means that the model was only trained on data that was available prior to the displayed month. Table 6 in Appendix B.1 lists the specific identifiers for each model.

Table 2: Overview of the LLMs used in our evaluation.

Model Family	Model Name	Size	Approximate Cutoff Date	Release Date
DeepSeek-Coder (DC)	DC-7B	7B	2023-09	2023-11
	DC-33B	33B	2023-09	2023-11
Qwen2.5-Coder (QC)	QC-7B	7B	2024-07	2024-09
	QC-14B	14B	2024-07	2024-11
	QC-32B	32B	2024-07	2024-11
QwQ	QwQ-32B	32B	2024-08	2025-03
DeepSeek-R1	DeepSeek-R1	671B	2024-07	2025-01
OpenAI	GPT-4o-mini	-	2023-10	2024-07
	GPT-4o	-	2023-10	2024-05
	o3-mini	-	2023-10	2025-01

### 5.3.2 Generation Parameters

Following [Jain et al. \(2025\)](#), we use a temperature of 0.2 and top- $p$  value of 0.95 to sample responses from traditional models (DeepSeek-Coder, Qwen2.5-Coder, and GPT models). For QwQ and DeepSeek-R1, we use the recommended temperature setting of 0.6 to prevent endless repetitions or incoherent outputs, along with a top- $p$  of 0.95. For o3-mini, parameters like temperature and top- $p$  are not supported. Instead, a parameter called reasoning effort controls how much thinking the model conducts; we set this parameter to high. We constrain the maximum generation length to 4096 tokens for DeepSeek-Coder and Qwen2.5-Coder, as we observe that outputs above this length contain endless looping; we do not constrain the generation length for other models and allow them to perform maximal chain-of-thought thinking. All other parameters are set at their default values.

### 5.3.3 Inference Setup

To obtain our results on DeepSeek-Coder, Qwen2.5-Coder, and QwQ, we download the models from [HuggingFace](#) and use vLLM ([Kwon et al., 2023](#)) to perform efficient inference on a cluster of A100-80GB GPUs. For DeepSeek-R1 and the OpenAI models, we use the respective APIs of the two companies.

### 5.3.4 Evaluation Metrics

For the execution prediction experiments, we generate five responses for each problem and compute the pass@1 metric ([Chen et al., 2021](#); [Kulal et al., 2019](#)) for OC, OR, MC, and MR (explained in Section 5.1). This metric measures the probability that a model produces the answer that fits the corresponding criterion in one generation, which is estimated by dividing the number of fitting answers by five. When reporting reversion rates, we exclude problems whose output is a Boolean variable; this choice makes the reversion metric more meaningful since it does not include cases where a model simply produces an incorrect result.

For the execution choice experiments, we conduct two runs for each problem and swap the order of the original and mutated programs to mitigate any bias a model may have for the ordering of the programs. To quantify preference (see Section 5.2), we assign a value of 0 if a model chooses the mutated program and a value of 1 if it chooses the original program. In this setting, we use OC to denote the percentage of correct responses when a model chooses to reason about the original program; OR denotes the percentage of responses that predict the output of the mutated program when a model chooses to reason about the original program. MC and MR are defined analogously. Similar to the execution prediction setting, we exclude problems with Boolean outputs when calculating reversion.

Table 3: Execution prediction and choice results on list datasets.

Dataset	Model	Execution Prediction				Execution Choice				
		OC ( $\uparrow$ )	MC ( $\uparrow$ )	OR ( $\downarrow$ )	MR ( $\downarrow$ )	Pref	OC ( $\uparrow$ )	MC ( $\uparrow$ )	OR ( $\downarrow$ )	MR ( $\downarrow$ )
LLM- List	DC-7B	85.4	31.6	3.1	55.5	79.0	81.7	18.6	5.0	64.1
	DC-33B	91.7	37.3	2.3	54.0	83.0	85.3	33.3	3.8	44.6
	QC-7B	94.1	58.7	1.7	29.7	90.9	90.3	35.7	3.4	56.2
	QC-14B	97.1	70.3	0.8	22.4	94.4	98.2	42.3	0.3	52.2
	QC-32B	99.6	73.4	0.0	26.1	96.8	99.3	40.0	0.0	69.2
	QwQ-32B	100.0	95.7	0.0	3.0	98.9	100.0	100.0	0.0	0.0
	DeepSeek-R1	100.0	98.2	0.0	0.6	95.0	100.0	95.7	0.0	0.0
	GPT-4o-mini	99.3	68.1	0.3	28.5	92.7	98.6	41.2	0.6	59.3
	GPT-4o	100.0	80.1	0.0	16.0	98.1	99.8	100.0	0.3	0.0
o3-mini	100.0	99.9	0.0	0.0	97.4	100.0	100.0	0.0	0.0	
DSL- List	DC-7B	24.6	24.2	4.9	4.8	46.6	12.8	16.5	9.1	5.4
	DC-33B	45.8	45.5	7.0	7.6	55.4	26.8	30.8	9.3	9.4
	QC-7B	61.6	58.1	2.5	4.7	55.8	52.4	58.1	8.4	6.0
	QC-14B	77.3	76.7	1.3	0.8	53.8	77.8	76.6	1.4	1.2
	QC-32B	85.7	87.2	1.7	0.8	49.2	81.2	85.5	4.6	3.0
	QwQ-32B	98.1	97.9	0.2	0.1	52.5	99.3	98.8	0.0	0.0
	DeepSeek-R1	97.7	98.9	0.5	0.0	48.4	99.6	99.3	0.0	0.0
	GPT-4o-mini	76.4	77.4	1.4	1.4	50.4	70.0	74.9	4.5	3.0
	GPT-4o	91.3	93.5	1.4	0.2	54.9	88.0	90.4	2.4	2.5
o3-mini	98.7	98.5	0.0	0.0	50.8	100.0	100.0	0.0	0.0	

## 6 Results

We present the results from our experiments. In each section, we first provide an overview of the tables and figures that contain the results. Then, we discuss the execution prediction results, execution prediction results as a function of program complexity, and execution choice results in separate subsections. In each subsection, we highlight important trends in the results and discuss their implications.

### 6.1 List Datasets

Table 3 presents the results for the execution prediction and choice experiments on the LLM-List and DSL-List datasets. The table contains a row of results for each dataset-model combination; the first and second columns identify the dataset and the model, respectively. The following columns contain results on the two sets of experiments, with each entry consisting of the corresponding metric described in Section 5.3.4, averaged over all problems in the dataset. Fig. 4 presents the execution prediction results on the list datasets as a function of program complexity; we use lines of code as a simple proxy for program complexity. In each subfigure, the left and right columns show results on the LLM-List and DSL-List datasets, respectively, while each row shows results for a different model. Each graph contains four lines: the blue, orange, green, and red lines correspond to OC, MC (correctness metrics) and OR, MR (reversion metrics).

#### 6.1.1 Execution Prediction Results

In Table 3, the OC and MC columns under Execution Prediction present how often a model predicts the correct output when given the original and mutated programs, respectively (higher is better). The OR and MR columns present how often a model, when given the original and mutated programs, respectively, predicts the output of the other program (lower is better).

On LLM-List, correctness is generally high for the original problems. Earlier open-access models (DC) exhibit a sharp drop in correctness (around 54 percentage points) from the original to mutated problems. Later traditional open-access models (QC) still exhibit significant drops (between 26–35 points), but to a much lesser degree than DC. There is also a gap between the original and mutated scores for GPT models, with the performance of GPT-4o-mini being comparable to QC-14B and GPT-4o being better than QC-32B. In the same family, correctness increases with model size (especially on mutated problems). The latest reasoning models (QwQ, DeepSeek-R1, o3-mini) show very little degradation in correctness from original to mutated problems (at most a 4-point drop); notably, o3-mini achieves essentially perfect performance. OR is close to zero for all models. DC models exhibit substantial reversion on mutated programs (above 50%); QC models still exhibit high reversion but to a much lesser degree than DC (from 22–30%). GPT-4o-mini has a reversion rate close to that of QC-7B, while GPT-4o has the lowest reversion compared to the other traditional LLMs, at around 16%. In contrast, the latest reasoning models exhibit very low reversion (at most 3%), with o3-mini showing no reversion.

On DSL-List, the correctness rates for the original and mutated problems are comparable for each model. DC models achieve very low scores (24–46%), while QC models are notably better in comparison (61–87%). Again, GPT-4o-mini’s performance is comparable to that of QC-14B, and GPT-4o is slightly better than QC-32B. In the same family, correctness increases significantly with model size. The latest reasoning models exhibit near-perfect performance. OR is comparable to MR for each model, and the reversion rate is in the single digits for all models.

These results highlight several important phenomena. They are consistent with the propositions that LLM-List programs are in-distribution and DSL-List programs are relatively out-of-distribution; also, mutation moves LLM-List programs out-of-distribution, while not significantly affecting the distribution of DSL-List programs. Furthermore, earlier models exhibit behavior consistent with significant pattern matching on in-distribution LLM-List problems, while the latest models exhibit behavior consistent with generalization. Earlier models have difficulty reasoning about relatively simple out-of-distribution DSL-List problems, while the latest models succeed at a near-perfect rate.

### 6.1.2 Execution Prediction Results vs. Program Complexity

To investigate how execution prediction ability varies with program complexity, we plot the results discussed above against lines of code for a subset of open- and closed-access models in Fig. 4. The results for the other models are shown in Fig. 7a in Appendix C. On LLM-List, correctness for original problems remains high regardless of program size; correctness for mutated problems drops and reversion increases as program size increases. This effect is extremely pronounced for earlier open-access models. For instance, on DC-33B, the gap between original and mutated problems is 80 points and reversion on mutated problems is 77% for programs around 15 lines. Even state-of-the-art traditional LLMs like QC-32B and GPT-4o still exhibit a substantial original-mutation drop and high reversion at higher program complexity. On the other hand, reasoning models like DeepSeek-R1 and o3-mini demonstrate near-perfect correctness and low reversion across all complexities on both the original and mutated problems. On DSL-List, original and mutated scores are close to identical and reversion is negligible for each model. The correctness of traditional LLMs decreases with program size; DC scores rapidly decline, while QC and GPT scores decline at a slower rate. The newest reasoning models are highly accurate at all program sizes.

These results are consistent with traditional LLMs somewhat generalizing on smaller programs, but their reasoning capabilities quickly falling with increasing problem complexity. In contrast, the latest reasoning LLMs are competent at reasoning across all program sizes, showing only a very slight drop in performance as problem complexity increases.

### 6.1.3 Execution Choice Results

In Table 3, the Pref column under Execution Choice presents the percentage of problems for which the LLM chooses to predict results for the original instead of the mutated code. We emphasize that (as described in Section 5.2), there is no mention of original vs. mutated code in our prompt, and a model is only instructed

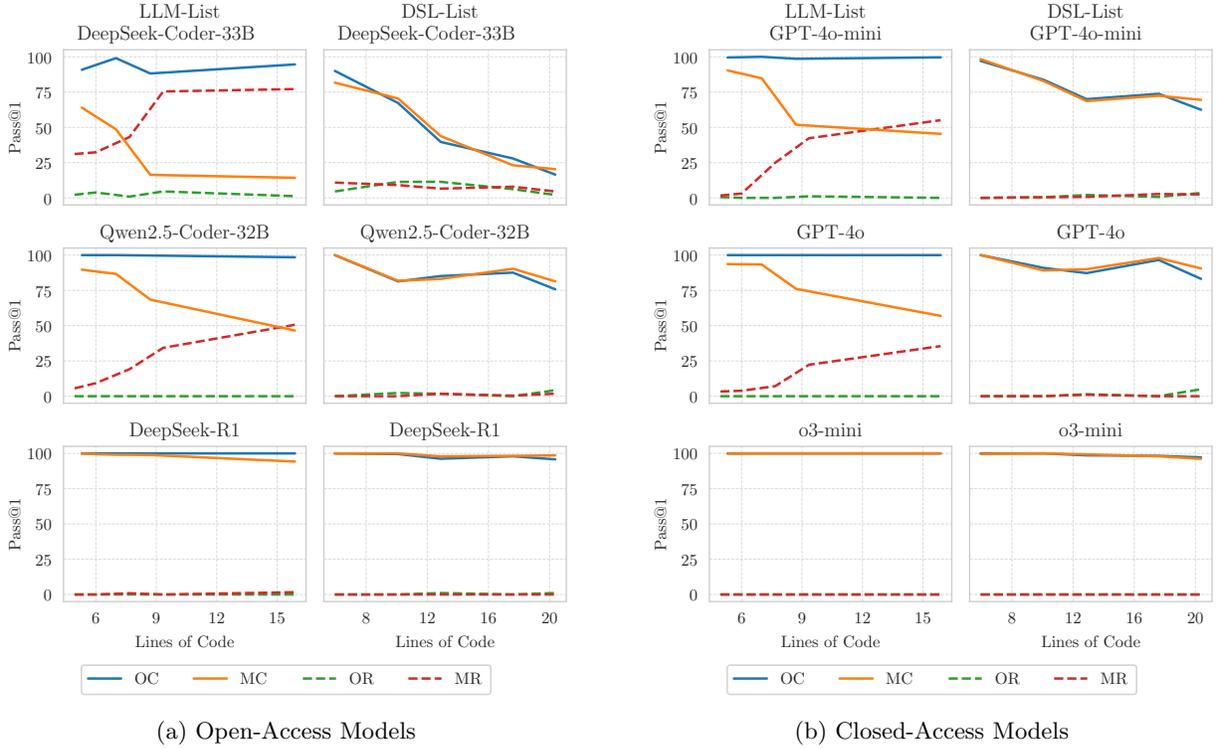


Figure 4: Execution prediction results on list datasets as a function of lines of code.

to choose whichever program it is more confident in reasoning about. The OC and MC columns present how often a model predicts the correct output when choosing to reason about the original and mutated programs, respectively. The OR and MR columns present how often a model predicts the output of the other program when choosing to reason about the original and mutated programs, respectively.

On LLM-List, all models consistently choose the original program. The DC models choose the original code at around 80% frequency, while all other models select the original code at over 90% frequency. Original correctness is generally high and comparable to that of execution prediction. All traditional models with the exception of GPT-4o experience a large drop in correctness when choosing to reason about the mutated programs. Notably, for these models, MC in execution choice is much lower than MC in execution prediction. On the other hand, GPT-4o and the reasoning models achieve very high MC (96% for DeepSeek-R1 and 100% for the others). OR is negligible for all models. All traditional models except GPT-4o exhibit substantial reversion when choosing to reason about the mutated programs. This phenomenon is especially pronounced on the QC models and GPT-4o-mini, which revert at about twice the rate compared to execution prediction. In stark contrast, GPT-4o and the reasoning models do not revert at all.

On DSL-List, all models choose the original program roughly half of the time. The correctness scores when choosing to reason about the original and mutated programs are comparable for each model. The correctness scores of the DC models are noticeably lower compared to the execution prediction experiments; we attribute this drop to their difficulty in reasoning correctly about the second program in the prompt in this scenario. The performances of other models are comparable across the prediction and choice experiments. Reversion is low both when choosing to reason about the original and mutated programs for all models.

These results highlight the introspection capabilities of LLMs, since they can recognize in-distribution programs as those they are more confident in reasoning about — with the models’ preferences for original LLM-List programs generally increasing with their reasoning capabilities. On LLM-List, earlier and less capable models generally exhibit much lower correctness and higher reversion when choosing to reason about

Table 4: Execution prediction and choice results on LeetCode.

Time Period	Model	Execution Prediction				Execution Choice				
		OC ( $\uparrow$ )	MC ( $\uparrow$ )	OR ( $\downarrow$ )	MR ( $\downarrow$ )	Pref	OC ( $\uparrow$ )	MC ( $\uparrow$ )	OR ( $\downarrow$ )	MR ( $\downarrow$ )
Before Cutoff	DC-7B	29.8	16.1	6.8	18.5	60.7	36.7	20.1	10.5	19.0
	DC-33B	46.8	24.0	10.6	29.8	63.6	39.1	24.0	7.9	17.6
	QC-7B	59.7	48.9	6.0	14.0	73.6	55.7	38.3	7.8	33.3
	QC-14B	74.9	68.7	2.8	12.1	86.3	72.2	59.2	5.1	19.0
	QC-32B	80.4	68.0	4.1	18.3	93.3	78.4	50.0	3.1	31.8
	QwQ-32B	97.9	95.8	0.1	2.0	95.3	97.9	100.0	0.0	0.0
	DeepSeek-R1	99.3	95.8	0.0	3.6	96.4	100.0	100.0	0.0	0.0
	GPT-4o-mini	83.2	69.3	2.5	13.9	78.2	77.1	59.0	3.7	23.6
	GPT-4o	91.2	77.9	1.4	12.2	93.6	85.1	82.6	1.4	4.5
o3-mini	99.9	99.9	0.0	0.0	89.4	99.7	100.0	0.0	0.0	
After Cutoff	DC-7B	21.6	18.4	9.3	10.2	59.9	29.0	23.1	9.6	15.7
	DC-33B	34.5	25.5	12.8	16.0	66.8	34.8	33.3	10.2	15.0
	QC-7B	51.4	50.8	10.2	12.4	71.0	50.8	45.4	10.5	13.5
	QC-14B	67.9	64.1	6.9	11.5	82.4	64.9	60.6	5.5	18.9
	QC-32B	75.8	69.1	8.3	13.3	88.0	77.8	66.7	6.6	26.3
	QwQ-32B	98.6	95.5	0.0	3.3	89.3	98.5	100.0	0.0	0.0
	DeepSeek-R1	99.5	98.5	0.4	1.5	87.7	99.4	100.0	0.0	0.0
	GPT-4o-mini	75.4	67.9	4.9	9.2	73.3	65.3	57.0	10.7	14.1
	GPT-4o	89.4	79.6	3.7	8.6	85.6	79.1	81.5	3.6	6.5
o3-mini	99.8	99.9	0.1	0.0	85.0	100.0	100.0	0.0	0.0	

the mutated programs, further accentuating their reliance on pattern matching. In contrast, the latest reasoning models achieve high correctness for all problems despite overwhelmingly selecting the original programs when given the choice. This trend is consistent with mutation indeed moving in-distribution programs out-of-distribution, and also with these models exhibiting strong reasoning abilities that generalize to new problems.

## 6.2 LeetCode Dataset

Table 4 presents the results for the execution prediction and choice experiments on the LeetCode dataset, separated into results before and after cutoff. Fig. 5 presents the execution prediction results on the LeetCode dataset as a function of lines of code. In each subfigure, the left and right columns, respectively, show results before and after cutoff. The layout of these results is otherwise identical to those of the list datasets.

### 6.2.1 Execution Prediction Results

Observing the results before cutoff, earlier open-access models (DC) struggle to correctly reason about code execution even on original problems (30–47%); later traditional open-access models (QC) improve significantly but only up to around 80%. The performance of GPT-4o-mini is comparable with that of QC-32B, while GPT-4o is noticeably better at around 91%. In the same family, OC significantly increases with model size. The latest reasoning models are extremely capable, with OC in the range of 98–100%. DC models exhibit a sharp drop in correctness (between 14–23 points) from the original to the mutated problems. QC and GPT models still exhibit significant drops (between 6–12 points and around 13 points, respectively), but to a lesser degree. In the same family, MC generally increases with model size. However, QC-14B and QC-32B attain similar MC; thus, their code reasoning capabilities on these problems may be more similar than their difference in OC suggests. QwQ and DeepSeek-R1 show slight degradation in correctness (at

most a 4-point drop from original to mutated), while o3-mini remains essentially perfect. For all traditional LLMs, MR is significantly higher than OR by 10–20 points. All of these LLMs exhibit substantial reversion on mutated programs from around 12% for GPT-4o and QC-14B to almost 30% for DC-33B. The latest reasoning models exhibit little reversion, with o3-mini showing no reversion at all.

For the results after cutoff, traditional LLMs exhibit a noticeable drop in OC (compared to problems before cutoff); this drop is larger for DC models (8–12 points), but still present for QC models (5–8 points) and GPT models (2–8 points). This consistent drop indicates that some of the performance of these models on problems before cutoff is likely attributable to memorization. Therefore, benchmarking these models on problems after cutoff reduces the influence of data contamination on evaluation results. However, for traditional LLMs, there still exists a sizable gap between correctness on original and mutated problems after cutoff. For example, GPT-4o has a 10-point gap between OC and MC on problems after cutoff. This result (especially given the similar OC values before and after cutoff for GPT-4o) highlights that the strategy of evaluating models on problems after cutoff may be insufficient for truly measuring their out-of-distribution code reasoning capabilities. We posit that this is because certain patterns in the code remain similar for problems before and after cutoff. Conversely, our mutation strategy is effective at creating programs with out-of-distribution coding patterns. The correctness scores of reasoning models on problems before and after cutoff are comparable, with o3-mini still being near-perfect on both the original and mutated problems. Traditional LLMs exhibit lower reversion on mutated problems, which is consistent with reduced memorization on problems after cutoff. However, MR is still typically higher than OR (by 1–5 points). This bias toward original programs reveals that these models still exhibit some degree of pattern matching (toward common programming patterns), even on problems they have likely not seen before. The latest reasoning models exhibit reversion close to zero, similar to problems before cutoff.

These results highlight the importance of using techniques to create out-of-distribution problems (e.g., mutation) beyond using data cutoffs. They also continue to support the hypothesis that while traditional models (especially earlier ones) rely on memorization and pattern matching, the latest reasoning models exhibit strong generalization abilities on code execution prediction in comparison.

### 6.2.2 Execution Prediction Results vs. Program Complexity

We plot the results discussed above against lines of code for a subset of open- and closed-access models in Fig. 5. The results for other models are shown in Fig. 7b in Appendix C. We first discuss trends on the traditional LLMs. Both before and after cutoff, correctness on original and mutated problems generally decreases as program size increases. For a given complexity, correctness on problems before cutoff is higher for less capable models (e.g., DC-33B, GPT-4o-mini) while correctness before and after cutoff are comparable for more capable models (e.g., QC-32B, GPT-4o). The gap between original and mutated correctness is generally smaller for problems after cutoff. Before cutoff, reversion on mutated problems is generally lower for smaller programs and higher for larger programs; however, after cutoff, MR does not meaningfully change with program size. For a given complexity, reversion on mutated problems is higher before cutoff (especially for less capable models). Reasoning models like DeepSeek-R1 and o3-mini demonstrate near-perfect performance on both original and mutated problems and low reversion across all programs.

These results are consistent with the code reasoning abilities of traditional LLMs decreasing as program size increases on LeetCode problems. Also, for more capable models, their performance on problems before and after cutoff are highly similar, while the mutated problems challenge their generalization capabilities to a greater degree. The latest reasoning models can reason highly accurately for all program sizes that we consider, on both original and mutated problems.

### 6.2.3 Execution Choice Results

Both before and after cutoff, all models consistently prefer to reason about the original program. Before cutoff, earlier models like DC choose the original code at around 60% frequency, while later models select the original code at around 90% frequency. After cutoff, the preferences toward original programs for each model are largely similar but slightly lower than before cutoff.

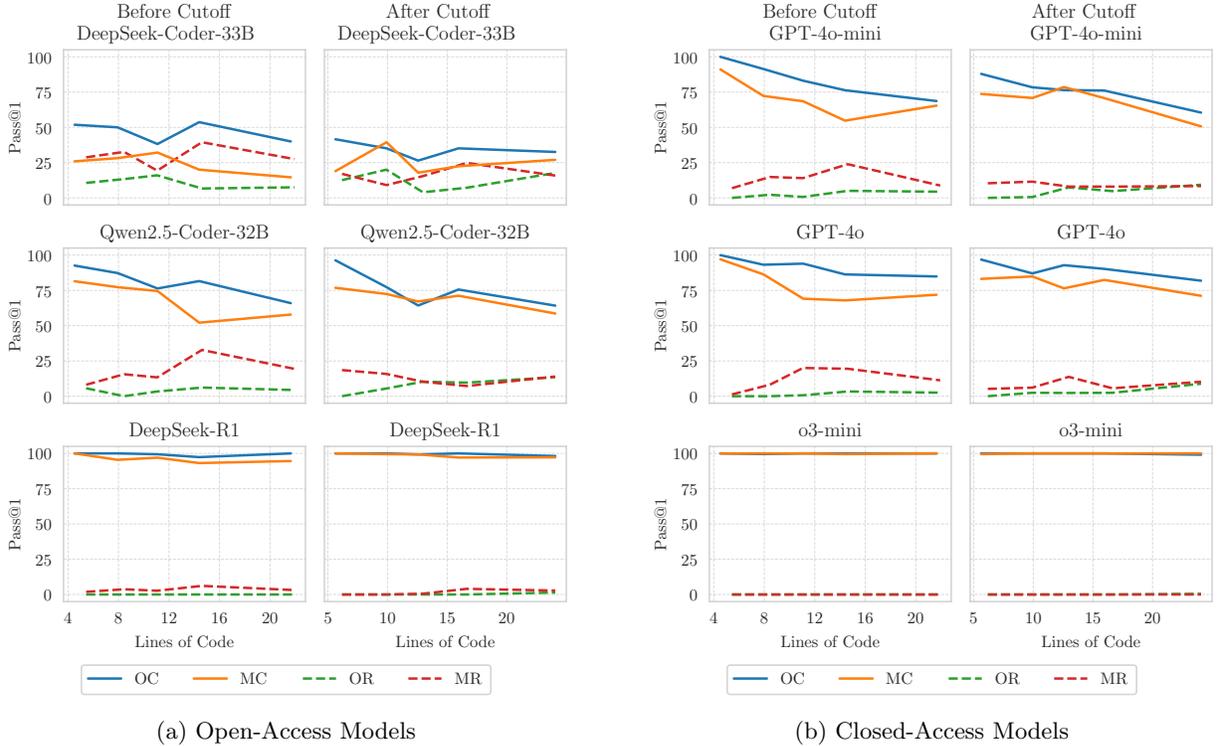


Figure 5: Execution prediction results on LeetCode as a function of lines of code.

Before cutoff, original correctness is generally comparable to and follows the same trend as in the execution prediction experiments. All traditional models except GPT-4o experience a large drop in correctness when choosing to reason about the mutated programs. In contrast, GPT-4o attains MC comparable with OC, and the reasoning models attain 100% MC. OR scores are low and comparable to those in the execution prediction experiments. All traditional models except GPT-4o exhibit substantial reversion (up to 33%) when choosing to reason about the mutated programs. This phenomenon is especially striking for the QC models and GPT-4o-mini, which revert at up to twice the rate compared to execution prediction. GPT-4o only reverts 5% of the time, while the reasoning models do not revert at all. The qualitative differences comparing performance results before and after cutoff are similar as in the execution prediction experiments.

These results once again indicate that the models (especially more capable ones) can clearly distinguish between original and mutated code, and prefer to reason about the original code. Low correctness and high reversion when choosing to reason about mutated programs are consistent with traditional LLMs' reliance on pattern matching. The ability of new reasoning models (especially o3-mini) to achieve near-perfect correctness for all problems is consistent with strong generalization capabilities on code reasoning.

## 7 Conclusion

Evaluating the generalization capabilities of LLMs on code reasoning is crucial toward understanding their ultimate impact on AI-assisted software engineering. We presented a range of techniques to obtain different classes of programs and an experimental methodology to evaluate LLM generalization by comparing their performance on these programs. Our results provided insights into the code reasoning capabilities of various LLMs on different classes of code and were consistent with the hypothesis that while prior models rely on pattern matching, the latest class of reasoning models exhibit strong generalization capabilities.

## Acknowledgments

This research was supported in part by the MIT–HPI Designing for Sustainability research program and by DARPA Grants HR001120C0191 and HR001124C0486. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

## References

- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Neural Information Processing Systems (NeurIPS)*, pages 1877–1901, 2020.
- Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting training data from large language models. In *USENIX Security Symposium (USENIX Security)*, pages 2633–2650, 2021.
- Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. Reasoning runtime behavior of a program with LLM: How far are we? In *International Conference on Software Engineering (ICSE)*, pages 140–152, 2025.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. Generalization or memorization: Data contamination and trustworthy evaluation for large language models. In *Findings of the Association for Computational Linguistics (ACL Findings)*, pages 12039–12050, 2024.
- Nathanaël Fijalkow, Guillaume Lagarde, Théo Matricon, Kevin Ellis, Pierre Ohlmann, and Akarsh Nayan Potta. Scaling neural program synthesis with distribution-based search. *AAAI Conference on Artificial Intelligence (AAAI)*, 36(6):6623–6630, 2022.
- Fengjuan Gao, Yu Wang, and Ke Wang. Discrete adversarial attack to models of code. *Proceedings of the ACM on Programming Languages*, 7(PLDI):172–195, 2023.
- Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. Evaluation of LLMs on syntax-aware code fill-in-the-middle tasks. In *International Conference on Machine Learning (ICML)*, pages 15907–15928, 2024.
- Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In *International Conference on Machine Learning (ICML)*, pages 16568–16621, 2024.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, YK Li, et al. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

- Ashish Hooda, Mihai Christodorescu, Miltiadis Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. Do large code models understand programming concepts? counterfactual analysis for code predicates. In *International Conference on Machine Learning (ICML)*, pages 18738–18748, 2024.
- Yiming Huang, Zhenghao Lin, Xiao Liu, Yeyun Gong, Shuai Lu, Fangyu Lei, Yaobo Liang, Yelong Shen, Chen Lin, Nan Duan, et al. Competition-level problems are effective LLM evaluators. In *Findings of the Association for Computational Linguistics (ACL Findings)*, pages 13526–13544, 2024.
- HuggingFace. HuggingFace models. <https://huggingface.co/models>.
- Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. GPT-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations (ICLR)*, 2025.
- Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering (TSE)*, 37(5):649–678, 2010.
- Bowen Jiang, Yangxinyu Xie, Zhuoqun Hao, Xiaomeng Wang, Tanwi Mallick, Weijie Su, Camillo Taylor, and Dan Roth. A peek into token bias: Large language models are not yet genuine reasoners. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4722–4756, 2024.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world Github issues? In *International Conference on Learning Representations (ICLR)*, 2024.
- Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. SPoC: Search-based pseudocode to code. In *Neural Information Processing Systems (NeurIPS)*, 2019.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Symposium on Operating Systems Principles (SOSP)*, pages 611–626, 2023.
- LeetCode. LeetCode contest. <https://leetcode.com/contest/>.
- Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. CodeI/O: Condensing reasoning patterns via code input-output prediction. *arXiv preprint arXiv:2502.07316*, 2025.
- Ziyu Li and Donghwan Shin. Mutation-based consistency testing for evaluating the code understanding capability of LLMs. In *International Conference on AI Engineering – Software Engineering for AI (CAIN)*, pages 150–159, 2024.
- Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. CodeMind: A framework to challenge large language models for code reasoning. *arXiv preprint arXiv:2402.09664*, 2024.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Neural Information Processing Systems (NeurIPS)*, pages 21558–21572, 2023.
- Seyed Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. GSM-Symbolic: Understanding the limitations of mathematical reasoning in large language models. In *International Conference on Learning Representations (ICLR)*, 2025.

- Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. LEVER: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning (ICML)*, pages 26106–26128, 2023.
- Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. NExT: Teaching large language models to reason about code execution. In *International Conference on Machine Learning (ICML)*, pages 37929–37956, 2024.
- OpenAI. GPT-4o mini: Advancing cost-efficient intelligence. <https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/>, 2024.
- OpenAI. OpenAI o3-mini system card. <https://openai.com/index/o3-mini-system-card/>, 2025.
- Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.
- Zhenting Qi, Hongyin Luo, Xuliang Huang, Zhuokai Zhao, Yibo Jiang, Xiangjun Fan, Himabindu Lakkaraju, and James R Glass. Quantifying generalization complexity for large language models. In *International Conference on Learning Representations (ICLR)*, 2025.
- Qwen Team. QwQ-32B: Embracing the power of reinforcement learning. <https://qwenlm.github.io/blog/qwq-32b/>, 2025.
- Md Rafiqul Islam Rabin, Aftab Hussain, Mohammad Amin Alipour, and Vincent J Hellendoorn. Memorization and generalization in neural code intelligence models. *Information and Software Technology*, 153: 107066, 2023.
- Martin Riddell, Ansong Ni, and Arman Cohan. Quantifying contamination in evaluating code generation capabilities of language models. In *Association for Computational Linguistics (ACL)*, pages 14116–14137, 2024.
- Manley Roberts, Himanshu Thakur, Christine Herlihy, Colin White, and Samuel Dooley. To the cutoff... and beyond? a longitudinal perspective on LLM data contamination. In *International Conference on Learning Representations (ICLR)*, 2024.
- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- Laboni Sarker, Mara Downing, Achintya Desai, and Tevfik Bultan. Syntactic robustness for LLM-based code generation. *arXiv preprint arXiv:2404.01535*, 2024.
- Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. ReCode: Robustness evaluation of code generation models. In *Association for Computational Linguistics (ACL)*, pages 13818–13843, 2023.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. In *Neural Information Processing Systems (NeurIPS)*, pages 24824–24837, 2022.
- Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Benjamin Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddhartha Venkat Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. LiveBench: A challenging, contamination-limited LLM benchmark. In *International Conference on Learning Representations (ICLR)*, 2025.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024a.

Zhou Yang, Zhensu Sun, Terry Zhuo Yue, Premkumar Devanbu, and David Lo. Robustness, security, privacy, explainability, efficiency, and usability of large language models for code. *arXiv preprint arXiv:2403.07506*, 2024b.

Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, Donggyun Han, and David Lo. Unveiling memorization in code models. In *International Conference on Software Engineering (ICSE)*, pages 1–13, 2024c.

# A Dataset Details

## A.1 DSL-List

### A.1.1 Additional Constraints

We apply the following syntactic constraints to all programs: (1) the first argument to a comparison operator must not be an integer literal (which avoids expressions like `0 == 0`), (2) the last argument to `extend`, `length`, and `map` must not be `empty`, (3) the integer literal `-1` is only used as an argument to `index`. We also optimize the compilation so that calling `index`, `init`, or `tail` on an empty list is disallowed (since these computations are guaranteed to lead to a runtime error), which makes program sampling much more efficient (as it rules out these programs by construction). The aforementioned rules are enforced when compiling the DSL to the CFG. Additional syntactic constraints are enforced during program sampling: (1) the same expression cannot appear on both sides of comparison or logical operators (which avoids expressions like `v1 == v1`), (2) a list cannot extend itself, (3) the same expression cannot be on both sides of a branch in a conditional, (4) a program must contain all function parameters. We also enforce the runtime constraint that a program must not produce the same output for all of its inputs. Programs that do not satisfy these constraints are discarded during sampling.

### A.1.2 Translation Algorithm

---

**Algorithm 2** DSL to Python Program Translation

---

```
Input: DSL program  $D$ 
Output: Python program  $P$ 
1: procedure TRANSLATEPROGRAM( $D$ )
2:    $T \leftarrow \text{MakeAST}(D)$ 
3:    $V \leftarrow \text{SetVariables}(T)$ 
4:    $E \leftarrow \text{SetExpressions}(T, V)$ 
5:    $P \leftarrow \text{Init}(T, E)$ 
6:   for  $T' \in \text{ReverseTopologicalOrder}(T)$  do
7:     if  $\text{IsPartialNode}(T')$  then
8:       continue
9:     end if
10:    if  $\text{IsMapNode}(T')$  then
11:       $P \leftarrow P \cdot \text{"for i in range(len(E[T'])):"}$ 
12:       $E[T'_{1,-1}] \leftarrow E[T'] \cdot \text{"[i]"}$ 
13:      if  $\text{IsStatementNode}(T'_1)$  then
14:         $P \leftarrow P \cdot \text{ToPythonStatement}(T', E, \text{map} = 1)$ 
15:      else
16:         $P \leftarrow P \cdot (E[T'_{1,-1}] \cdot \text{"="} \cdot \text{ToPythonExpression}(T', E))$ 
17:      end if
18:    else if  $\text{IsStatementNode}(T')$  then
19:       $P \leftarrow P \cdot \text{ToPythonStatement}(T', E, \text{map} = 0)$ 
20:    end if
21:  end for
22:   $P \leftarrow P \cdot \text{"return "}$   $\cdot E[T]$ 
23:  return  $P$ 
24: end procedure
```

---

Algorithm 2 describes our procedure for translating a functional program in our DSL to imperative Python code. First, we construct the abstract syntax tree of the DSL program (Line 2).

Table 5: Translation of statement nodes from DSL to Python.  $T'$  is the current AST node, and  $T'_i$  is the  $i^{\text{th}}$  child node of  $T'$ .  $E$  is the mapping from an AST node to the expression representing the result of the computation at that node.

DSL Program	Python Program
<code>append <math>T'_1</math> <math>T'_2</math></code>	<code><math>E[T'_2].append(E[T'_1])</math></code>
<code>extend <math>T'_1</math> <math>T'_2</math></code>	<code><math>E[T'_2].extend(E[T'_1])</math></code>
<code>init <math>T'_1</math></code>	<code><math>E[T'_1].pop()</math></code>
<code>tail <math>T'_1</math></code>	<code><math>E[T'_1].pop(0)</math></code>
	<code>if <math>E[T'_1]</math>:</code>
	<code>    <math>E[T'] = E[T'_2]</math></code>
<code>if <math>T'_1</math> <math>T'_2</math> <math>T'_3</math></code>	<code>else:</code>
	<code>    <math>E[T'] = E[T'_3]</math></code>

In our procedure, only the `empty` and `if` primitives create new variables, while other primitives are translated to statements or expressions on existing variables. On Line 3, we sort the AST nodes in reverse topological order and assign variable names  $v\{i\}$  to `empty` and `if` nodes, starting from  $v1$ . We also assign names  $a\{i\}$  to function parameters, starting from  $a1$ . The resulting  $V$  is a map from an applicable AST node to its corresponding variable name.

On Line 4, we create the mapping  $E$  from an AST node to the expression representing the result of the computation at that node. The `length` and `index` primitives, integers, and comparison and boolean operators are directly mapped to their corresponding Python expressions. At this stage, we only assign complete expressions and handle partial applications (e.g., the function argument to `map`) at a later stage. On the other hand, list operations and `map` will later be translated to Python statements; the key, however, is that the *result* of the computation at that node is just the expression consisting of the list being operated on (e.g.,  $v1$  in  `$v1.append(0)$` ). Thus, we loop through the AST nodes in reverse topological order, and assign (1) corresponding Python expressions to expression nodes and (2) the expression of the list being operated on to statement nodes.

Next, we start writing the translated Python code  $P$ . We first write the function header according to the type signature of the program and initialize all variables corresponding to `empty` nodes at the beginning of the function as a tuple (e.g.,  $v1, v2 = [], []$ ) on Line 5. Note that due to Python’s scoping rules, variables initialized in conditionals are accessible within the entire function, and do not need to be first declared (and hence why we do not write them on Line 5).

We now loop through the AST nodes in reverse topological order (Lines 6–21). We skip any partial nodes, which are those with fewer children than the usual number of parameters (Lines 7–9); for example, a `len` node without any children because it is the first child of a `map` node is partial. For statement nodes (`append`, `extend`, `init`, `tail`, `if`) that are not direct children of a `map` node, we directly translate them to Python according to the rules in Table 5 and append the Python statement to  $P$  (Line 19). We write  $T'_i$  to mean the  $i^{\text{th}}$  child node of  $T'$  (where the index  $-1$  refers to the last child);  $T'_{i,j}$  refers to the  $j^{\text{th}}$  child node of  $T'_i$ . We use `.` to denote string concatenation and `.*` to denote string concatenation followed by the concatenation of a newline character (and preceded by an indentation if inside a loop). To translate `map`, we first write the loop header for the list that is being operated on (Line 11). Second, we assign the expression of this list (followed by the indexing `[i]` to retrieve an element of the list) to the expression of the missing argument in the mapping function (Line 12). There are now two cases: one if the mapping function is otherwise translated to statements and the other if the mapping function is `len` or `index`. In the first case, we use the same conversion shown in Table 5, except that conditionals are additionally followed by the line  `$E[T'_{i,-1}] = E[T'_i]$`  (Line 14). In the second case, we convert the `len` or `index` node to a Python expression and prepend it with an assignment to the mapped list element (Line 16).

Finally, we return  $E[T]$  as the last line of our translated program, which is the expression corresponding to the root node of the AST (Line 22).

## A.2 LLM-List

We use the following prompts to create the LLM-List dataset. For input generation, we prepend our request with an example to show the model the correct output format. We use the function `def add(a, b):\nreturn a + b` with the description “returns the sum of two numbers”, and specify the response as `3, 5\n-2, 7\n0, 0`. If any generated input contains floating-point numbers or leads to a runtime error, we regenerate the inputs and append the following sentence to the end of the prompt instruction (before the function code): `Do NOT include the following inputs:` followed by a comma-separated list of the erroneous inputs.

### LLM-List Brainstorming Prompt

Your task is to brainstorm a list of 100 known / common list functions in Python. These could be

- ↪ standard textbook algorithms or simple utility functions. Some examples are `length`, `reverse`,
- ↪ `unique`, `compact`, `flatten`, `insert`, `index`, `union`, `tail`, `permutations`, `order-by`, `mean`, `median`, `range`,
- ↪ `argmax`.

Each function you come up with must satisfy the following conditions:

- Takes in a list of integers as one of the parameters and returns a list, integer, or boolean after
  - ↪ doing some processing on the input.
- Does NOT contain random operations.
- Does NOT involve substantial floating-point operations.
- Does NOT rely on any imports (e.g., `numpy` or the Python standard library).

Try to have as much variability in the types of operations; for any class or variations of operations,

- ↪ have at most 2-3. Structure your response in the following manner. The name should be a function
- ↪ signature (e.g., `length(lst)`), and the description should encapsulate the expected behavior of the
- ↪ function.

1. "[name]": "[description]"
2. "[name]": "[description]"
3. "[name]": "[description]"
- ...

### LLM-List Code Generation Prompt

Your task is to write a Python function `{function_header}` that `{function_description}`. You may use

- ↪ built-ins, but limit your usage so the function has enough logic in it; you are not allowed to use
- ↪ `numpy`. Make the logic in your function as explicit as possible, and make sure that the result
- ↪ returned by your function is deterministic. Do not include comments, and do not output any extra
- ↪ information.

### LLM-List Input Generation Prompt

You are given a Python function named `{function_name}` below, which `{function_description}`. Your goal

- ↪ is to generate 3 simple test inputs for this function that comprehensively test all functionality
- ↪ of the `{function_name}` function and produce no errors when executed. Do NOT include any extra
- ↪ information and put each input on a separate line. If the input contains multiple arguments,
- ↪ separate them by commas. Do NOT include floating-point values. Make sure that lists contain only a
- ↪ few elements, but are not empty.

```
“python
{function_code}
““
```

### A.3 Example Problems

<pre>def slice(lst, start, end):     result = []     for i in range(len(lst)):         if i &gt;= start and i &lt; end:             result.append(lst[i])     return result assert slice(lst = [10, 20, 30, 40, 50],     ↪ start = 2, end = 5) == [30, 40, 50]</pre>	<pre>def slice(lst, start, end):     result = []     for i in range(len(lst)):         if i &gt; start and i &lt; end:             result.append(lst[i])     return result assert slice(lst = [10, 20, 30, 40, 50],     ↪ start = 2, end = 5) == [40, 50]</pre>
<pre>def f1(a1, a2):     v1 = []     v1.append(a1)     a2.pop()     a2.pop(0)     for i in range(len(v1)):         v1[i] = v1[i][3]     a2.append(2)     if len(a1) &lt; a1[0]:         v2 = a2     else:         v2 = v1     for i in range(len(v2)):         if a1[0] == 5:             v3 = a2[len(a1)]         else:             v3 = v2[i]         v2[i] = v3     return v2 assert f1(a1 = [2, 0, 3, 1, 4], a2 = [0, 5,     ↪ 3]) == [1]</pre>	<pre>def f1(a1, a2):     v1 = []     v1.append(a1)     a2.pop()     a2.pop(0)     for i in range(len(v1)):         v1[i] = v1[i][2]     a2.append(2)     if len(a1) &lt; a1[0]:         v2 = a2     else:         v2 = v1     for i in range(len(v2)):         if a1[0] == 5:             v3 = a2[len(a1)]         else:             v3 = v2[i]         v2[i] = v3     return v2 assert f1(a1 = [2, 0, 3, 1, 4], a2 = [0, 5,     ↪ 3]) == [3]</pre>
<pre>def maximizeTheProfit(n: int, offers:     ↪ List[List[int]]) -&gt; int:     f = [0] * (n + 1)     t = 0     for x, y, z in sorted(offers,     ↪ key=lambda it: it[1]):         x += 1         y += 1         while t &lt; y:             f[t + 1] = f[t]             t += 1         f[y] = max(f[x - 1] + z, f[y])      return max(f) assert maximizeTheProfit(n = 5, offers =     ↪ [[0, 0, 1], [0, 2, 2], [1, 3, 2]]) == 3</pre>	<pre>def maximizeTheProfit(n: int, offers:     ↪ List[List[int]]) -&gt; int:     f = [0] * (n + 1)     t = 0     for x, y, z in sorted(offers,     ↪ key=lambda it: it[1]):         x += 1         y -= 1         while t &lt; y:             f[t + 1] = f[t]             t += 1         f[y] = max(f[x - 1] + z, f[y])      return max(f) assert maximizeTheProfit(n = 5, offers =     ↪ [[0, 0, 1], [0, 2, 2], [1, 3, 2]]) == 4</pre>

Figure 6: Example original (left column) and mutated (right column) problems from the LLM-List (top row), DSL-List (center row), and LeetCode (bottom row) datasets.

## B Experimental Details

### B.1 Benchmarked Models

We provide the specific identifiers for the large language models we evaluate in Table 6. For each model, we provide its name on HuggingFace (for open-access models) or specific snapshot (for closed-access models).

Table 6: Names and identifiers of the large language models used in our evaluation.

Model Name	Model Identifier
DeepSeek-Coder-7B	deepseek-ai/deepseek-coder-6.7b-instruct
DeepSeek-Coder-33B	deepseek-ai/deepseek-coder-33b-instruct
Qwen2.5-Coder-7B	Qwen/Qwen2.5-Coder-7B-Instruct
Qwen2.5-Coder-14B	Qwen/Qwen2.5-Coder-14B-Instruct
Qwen2.5-Coder-32B	Qwen/Qwen2.5-Coder-32B-Instruct
QwQ-32B	Qwen/QwQ-32B
DeepSeek-R1	deepseek-ai/DeepSeek-R1
GPT-4o-mini	gpt-4o-mini-2024-07-18
GPT-4o	gpt-4o-2024-08-06
o3-mini	o3-mini-2025-01-31

### B.2 Execution Prediction Prompts

We present our execution prediction prompts below. We use the zero-shot prompt for reasoning models and the one-shot prompt for traditional models.

#### Execution Prediction Prompt (Zero-Shot)

You are given a Python program and an assertion containing an input to a function. Replace the ?? in  
↪ the assertion with a literal (no unsimplified expressions, no function calls) representing the  
↪ function’s return value for the given input. Execute the program exactly as written, even if it is  
↪ incorrect or incomplete. For your final answer, provide the full assertion in [ANSWER] and [/  
↪ ANSWER] tags.

```
[PYTHON]
{program}
assert {function_name}({input}) == ??
[/PYTHON]
```

### Execution Prediction Prompt (One-Shot)

You are given a Python program and an assertion containing an input to a function. Replace the ?? in  
↪ the assertion with a literal (no unsimplified expressions, no function calls) representing the  
↪ function's return value for the given input. Execute the program exactly as written, even if it is  
↪ incorrect or incomplete. Execute the program step by step before arriving at an answer, and  
↪ provide the full assertion with the function output in [ANSWER] and [/ANSWER] tags, following the  
↪ example.

```
[PYTHON]
def performOperation(s):
    s = s + s
    return "b" + s + "a"
assert performOperation(s = "hi") == ??
[/PYTHON]
[THOUGHT]
Let's execute the code step by step:

1. The function performOperation is defined, which takes a single argument s.
2. The function is called with the argument "hi", so within the function, s is initially "hi".
3. Inside the function, s is concatenated with itself, so s becomes "hihi".
4. The function then returns a new string that starts with "b", followed by the value of s (which is
   ↪ now "hihi"), and ends with "a".
5. The return value of the function is therefore "bhihia".
[/THOUGHT]
[ANSWER]
assert performOperation(s = "hi") == "bhihia"
[/ANSWER]

[PYTHON]
{program}
assert {function_name}({input}) == ??
[/PYTHON]
```

### B.3 Execution Choice Prompts

We present our execution choice prompts below. We use the zero-shot prompt for reasoning models and GPT-4o and the one-shot prompt for the other models. We observed that the one-shot prompt prevented GPT-4o from performing chain-of-thought thinking, whereas including an example was beneficial to elicit chain-of-thought thinking from other traditional LLMs.

#### Execution Choice Prompt (Zero-Shot)

You are given two Python programs below and an assertion containing an input to a function. First,  
↪ choose either program, whichever one you are more confident in reasoning about. Then, replace the  
↪ ?? in the assertion with a literal (no unsimplified expressions, no function calls) representing  
↪ the function's return value for the given input on your chosen program. Execute the program  
↪ exactly as written, even if it is incorrect or incomplete. For your final answer, output the  
↪ letter of your chosen program (A or B) and the full assertion in the following json format:

```
{
  "chosen_program": chosen_program_letter,
  "assertion": full_assertion
}

[PROGRAM_A]\n{program_a}\n[/PROGRAM_A]
[PROGRAM_B]\n{program_b}\n[/PROGRAM_B]
[ASSERTION]\nassert {function_name}({input}) == ??[/ASSERTION]
```

## Execution Choice Prompt (One-Shot)

You are given two Python programs below and an assertion containing an input to a function. First,  
↳ choose either program, whichever one you are more confident in reasoning about. Then, replace the  
↳ ?? in the assertion with a literal (no unsimplified expressions, no function calls) representing  
↳ the function's return value for the given input on your chosen program. Execute the program  
↳ exactly as written, even if it is incorrect or incomplete. Execute the program step by step before  
↳ arriving at an answer, then output the letter of your chosen program (A or B) and the full  
↳ assertion in the following json format:

```
{
  "chosen_program": chosen_program_letter,
  "assertion": full_assertion
}
```

# Example

[PROGRAM\_A]

```
def performOperation(s):
    first = s[0].upper()
    rest = s[1:].upper()
    return first + rest
```

[/PROGRAM\_A]

[PROGRAM\_B]

```
def performOperation(s):
    first = s[0].upper()
    rest = s[1:].lower()
    return first + rest
```

[/PROGRAM\_B]

[ASSERTION]

```
assert performOperation(s = 'hELLO') == ??
```

[/ASSERTION]

[THOUGHT]

First, let's figure out which program I am more confident in reasoning about.

Looking at programs A and B, the difference is in the expression for rest. Program A defines rest as s

- ↳ [1:].upper() while program B defines rest as s[1:].lower(). Program B looks similar to how one
- ↳ might implement the capitalize() function, so I will choose program B as I am more confident in
- ↳ reasoning about this program behavior.

Now, let's execute the code step by step:

1. The function performOperation is defined, which takes a single argument s.
2. The function is called with the argument 'hELLO', so within the function, s is initially 'hELLO'.
3. The variable first is defined as the upper case of the first character of s, which is 'H'.
4. The variable rest is defined as the lower case of s[1:], which is 'ello'.
4. The function returns first ('H') concatenated with rest ('ello').
5. The return value of the function is therefore 'Hello'.

[/THOUGHT]

```
{
  "chosen_program": "B",
  "assertion": "assert performOperation(s = 'hELLO') == 'Hello'"
}
```

# Question

[PROGRAM\_A]\n{program\_a}\n[/PROGRAM\_A]

[PROGRAM\_B]\n{program\_b}\n[/PROGRAM\_B]

[ASSERTION]\nassert {function\_name}({input}) == ??\n[/ASSERTION]

## C Experimental Results

We present the execution prediction results as a function of lines of code for the remaining open-access models not shown in the main text in Fig. 7.

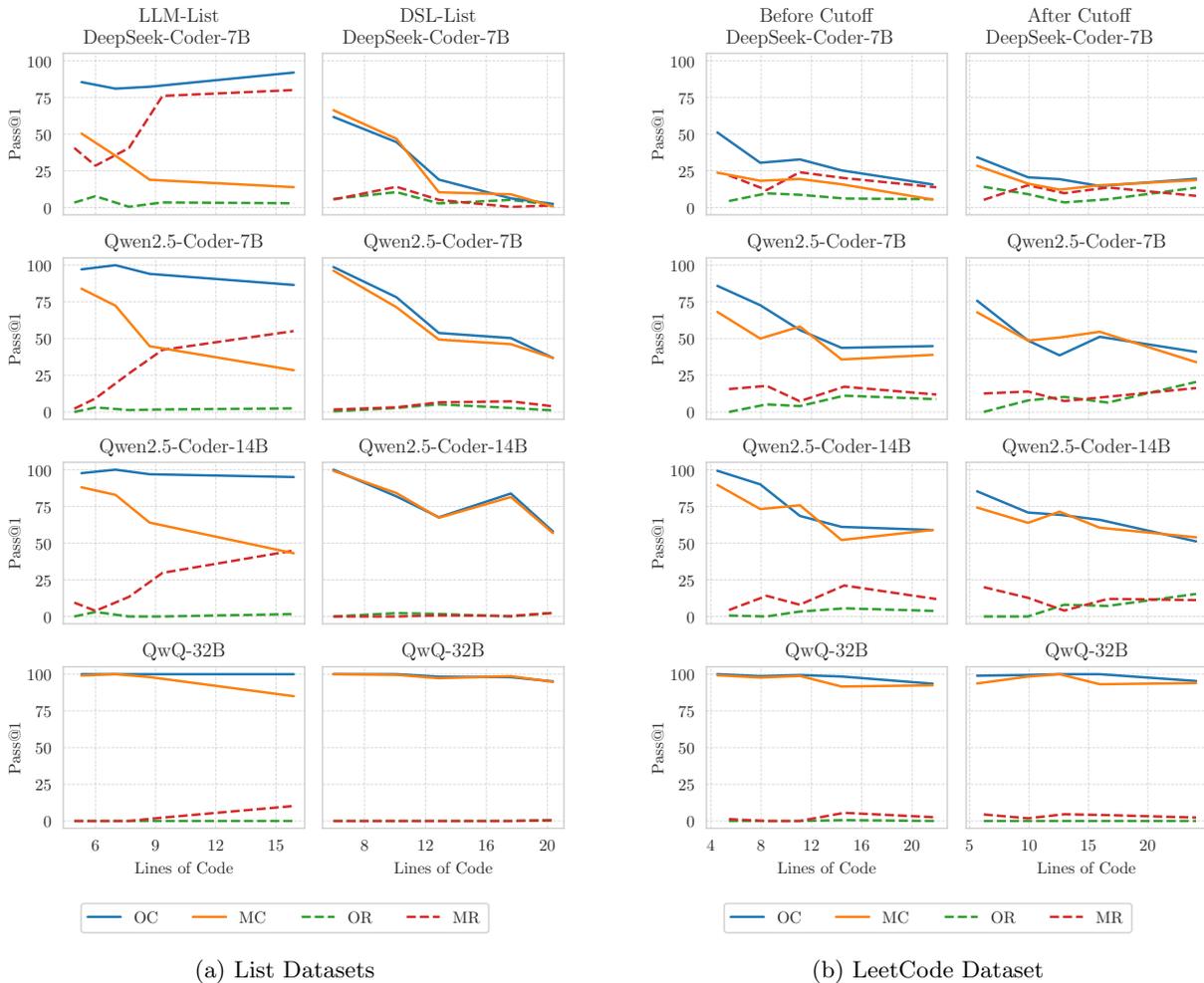


Figure 7: Execution prediction results on remaining open-access models as a function of lines of code.