Turbulence: Systematically and Automatically Testing Instruction-Tuned Large Language Models for Code

Shahin Honarvar
Department of Computing
Imperial College London
London, UK
s.honarvar21@imperial.ac.uk

Mark van der Wilk
Department of Computer Science
University of Oxford
Oxford, UK
mark.vdwilk@cs.ox.ac.uk

Alastair F. Donaldson
Department of Computing
Imperial College London
London, UK
alastair.donaldson@imperial.ac.uk

***Abstract*—We present a method for systematically evaluating the correctness and robustness of instruction-tuned large language models (LLMs) for code generation via a new benchmark, Turbulence. Turbulence consists of a large set of natural language *question templates*, each of which is a programming problem, parameterised so that it can be asked in many different forms. Each question template has an associated *test oracle* that judges whether a code solution returned by an LLM is correct. Thus, from a single question template, it is possible to ask an LLM a *neighbourhood* of very similar programming questions, and assess the correctness of the result returned for each question. This allows gaps in an LLM's code generation abilities to be identified, including *anomalies* where the LLM correctly solves *almost all* questions in a neighbourhood but fails for particular parameter instantiations. We present experiments against five LLMs from OpenAI, Cohere and Meta, each at two temperature configurations. Our findings show that, across the board, Turbulence is able to reveal gaps in LLM reasoning ability. This goes beyond merely highlighting that LLMs sometimes produce wrong code (which is no surprise): by systematically identifying cases where LLMs are able to solve some problems in a neighbourhood but do not manage to generalise to solve the whole neighbourhood, our method is effective at highlighting *robustness* issues. We present data and examples that shed light on the kinds of mistakes that LLMs make when they return incorrect code results.**

***Index Terms*—Large language models, correctness, robustness, AI evaluation, code generation**

I. Introduction

Large language models (LLMs) have proven effective in tasks such as translating between programming languages [1] and answering programming questions [2]. Their effectiveness has been increased via *instruction tuning* [3], which uses supervision to teach a pre-trained LLM to follow particular kinds of instructions and apply this capability to unseen tasks [4]. However, current instruction-tuned LLMs often generate *incorrect* code [5]–[8]. To further enable AI-based code generation in mainstream software development, where correctness and robustness are essential, it is important to address the issue of developers' lack of trust in LLMs with respect to code generation tasks [9], [10]. To this end, several works have focused on assessing the ability of LLMs to generate correct code [5], [6], [11]–[14], whereas other studies have investigated *robustness* while preserving the semantics of prompts or code in their evaluations [15]–[25]. Our paper complements this body of work by introducing a novel dimension to correctness and robustness testing. While existing studies primarily assess how models handle semantically equivalent inputs—ensuring that variations do not alter the meaning or functionality of the code or descriptions—we explore how models perform when faced with neighbourhoods of similar but *non-equivalent* tasks.

Our contribution. Inspired by Gardner et al. [26], the key idea behind our approach is that instead of evaluating an LLM using separate, isolated coding problems, we use sets of related problems, where all problems in a set are variations on a theme—they are all in the same *neighbourhood*. Rather than being interested in whether an LLM can solve any *particular* problem, we are interested in identifying *discontinuities* in the LLM's ability to solve a neighbourhood of problems—e.g. cases where the LLM correctly solves most problems in a neighbourhood but fails for certain cases. As opposed to merely identifying problems with isolated code generation prompts (the fact that problematic cases exist is no surprise), identifying discontinuities within a neighbourhood reveals the limits of an LLM's (in)ability to generalise.

Our approach is based on the notion of a *question template*. A question template is a natural language programming specification parameterised by one or more values. An example is shown in Figure 1a. This question template is parameterised by two integer values, p_1 and p_2, and can be instantiated for any $0 \leq p_1 \leq p_2 \leq K$, where K is a reasonable upper limit for Python list sizes. An instantiation of the question template of Figure 1a with $p_1 = 1$ and $p_2 = 8$ is shown in Figure 1b. This is called a *question instance*.

Each question template is paired with an associated *oracle template*. This includes a suite of parameterised unit tests, featuring the same parameters that appear in the question

This work was supported by UK Research and Innovation [grant number EP/S023356/1], in the UKRI Centre for Doctoral Training in Safe and Trusted Artificial Intelligence (www.safeandtrustedai.org).

Write a function called 'sum_even_ints_inclusive' that takes one argument, a list of integers, and returns the sum of all even integers from index $\boldsymbol{p_1}$ to index $\boldsymbol{p_2}$, both inclusive. If no even integers exist in the specified range, the function should return 0.

(a) A question template featuring two parameters p_1 and p_2.

Write a function called 'sum_even_ints_inclusive' that takes one argument, a list of integers, and returns the sum of all even integers from index **1** to index **8**, both inclusive. If no even integers exist in the specified range, the function should return 0.

(b) A question instance from (a) with $p_1 = 1$ and $p_2 = 8$.

```
def test_odd_range():
  odd_list = [i for i in range(-10001, p2*10, 2)]
  assert sum_even_ints_inclusive(odd_list) == 0
```

(c) A test case template for (a) featuring p_2.

```
def test_odd_range():
  odd_list = [i for i in range(-10001, 8*10, 2)]
  assert sum_even_ints_inclusive(odd_list) == 0
```

(d) A test case instance from (c) with $p_2 = 8$.

```
def sum_even_ints_inclusive(lst):
  lst = lst[p1 : p2 + 1]
  return sum([i for i in lst if i % 2 == 0])
```

(e) Model solution template for (a) featuring p_1 and p_2.

```
def sum_even_ints_inclusive(lst):
  lst = lst[1 : 8 + 1]
  return sum([i for i in lst if i % 2 == 0])
```

(f) A model solution instance from (e) with $p_1 = 1$ and $p_2 = 8$.

Fig. 1: An example of a question template, test case template and model solution template, and an instantiation of each

template. Figure 1c shows an example parameterised test case for the question template of Figure 1a. The parameterised test suite can be instantiated to yield a set of concrete tests for a question instance. For example, Figure 1d shows the concrete test case obtained by instantiating the test case of Figure 1c with $p_1 = 1$ and $p_2 = 8$ (as p_1 does not occur in the test case template its value is irrelevant to this instantiation). This test is suitable for checking the correctness of solutions to the question instance of Figure 1b. An oracle template also includes a *model solution*, which we discuss in Section II.

Given a (question template, oracle template) pair, an LLM can be asked, via multiple independent queries, to solve a neighbourhood of e.g. 100 different question instances derived from the question template, each of which can be automatically checked for correctness via the corresponding instantiated oracle. The results might be extreme, suggesting that the LLM is completely incapable of solving this neighbourhood of questions (if every solution fails the oracle), or that the LLM can easily solve this neighbourhood of questions (if all solutions pass). More intriguingly, the LLM might successfully solve many instances of a question template, but unexpectedly yield an incorrect solution for specific parameter values. Conversely, it may exhibit a lack of success in addressing the majority of instances in a neighbourhood, yet unexpectedly deliver a correct solution for certain parameter values. Our method for identifying these discontinuities may offer valuable insights into the limitations of the LLM's reasoning capabilities and has the potential to serve as a source of data for training or fine-tuning. Furthermore, our approach may feed into discussions as to whether LLMs are truly exhibiting *emergent* reasoning powers, as some researchers have speculated [27]–[29]. It seems implausible that an LLM that can truly reason would be capable of solving the programming question of Figure 1a for many values of p_1 and p_2 but not, say, for the particular case of $p_1 = 100$ and $p_2 = 200$. Prior methods for testing LLM-based code generation using standalone problems (see Section VII) cannot yield such insights: key to our method is that it assesses both LLM correctness (by testing each generated code response) and LLM robustness (by assessing how correctness varies across a neighbourhood).

The Turbulence benchmark. Conceptually, the method we propose is both LLM- and programming language-agnostic. We have put it into practice by building a new benchmark, Turbulence, for assessing the capability of instruction-tuned LLMs at generating Python code. Turbulence comprises (1) infrastructure for automatically assessing LLMs against a set of question and oracle templates, and (2) a set of 60 question and oracle templates that we have curated. We expect the long-lasting impact of our work to come from (1), because our method and infrastructure can be used with any suitable set of question and oracle templates in the future. Our curated question templates allow us to report results across various state-of-the-art LLMs. The questions were created from scratch by the paper's authors to avoid direct similarities to existing online questions or code, thus preventing training bias. They were refined based on feedback from a number of experienced Python programmers to minimise any potential ambiguity.

Research questions and summary of findings. We have used Turbulence to evaluate a variety of LLMs: the *GPT-4* [30] and *GPT-3.5-turbo* [31] models from OpenAI [32], the *Command* model [33] from Cohere [34], the 4-bit quantised version of *CodeLlama:7B*, and the 4-bit quantised version of *CodeLlama:13B* with the full precision models and the 4-bit quantised versions being provided by Meta [35] and Ollama [36], respectively. Our evaluation is guided by the following research questions about the instruction-tuned LLMs:

- **RQ1:** How robust are LLMs in code generation when confronted with alterations in fixed values such as numerical values or string characters within prompts?
- **RQ2:** How does setting an LLM's temperature to zero for maximum determinism affect its performance compared to the default temperature?
- **RQ3:** What are the primary errors in the code responses

of the LLMs that render the responses incorrect?

Our findings show that *GPT-4* outperformed other LLMs. Nevertheless, all LLMs exhibited a lack of robustness when faced with variations in many questions. Certain question neighbourhoods posed challenges that were either entirely solvable or entirely unsolvable for the LLMs. However, a significant portion of the question neighbourhoods were only partially solved by the LLMs. Lowering the temperature reduced the number of partially solved question neighbourhoods, with more falling into either the fully solved or unsolved categories. This makes sense, because at a lower temperature an LLM will behave more deterministically, so that it is more likely to consistently fail or consistently succeed at a task, whereas at a higher temperature the LLM may behave in a more creative manner, leading to more fluctuation in its ability to solve a given task successfully. Despite the stochastic nature inherent in LLMs, the partial resolution of some question neighbourhoods *could potentially* highlight gaps in the training data used for the LLMs or flaws in their reasoning. We present an analysis of the common problems associated with incorrect code generated by LLMs in Section V.

In summary, the main contributions of this paper are:

- A new approach to assessing correctness and robustness of the code generation capabilities of instruction-tuned LLMs via *neighbourhoods* of related problem instances.
- Turbulence, a benchmark and automated testing framework based on our approach, for assessing the Python code generation capabilities of instruction-tuned LLMs.
- A study using Turbulence to evaluate the correctness and robustness of five state-of-the-art instruction-tuned LLMs of varying sizes and a deep dive into the key sources of errors in incorrect solutions.

In the rest of the paper, we give an overview of our approach (Section II), present the Turbulence benchmark (Section III), present results applying Turbulence to a range of instruction-tuned LLMs (Section IV), and discuss characteristics of incorrect code returned by LLMs (Section V). We discuss threats to validity (Section VI) and related work (Section VII) before concluding (Section VIII).

II. Our Benchmarking Approach

We now describe our general approach to benchmarking LLMs for code, an overview of which is shown in Figure 2. In Section III we describe Turbulence, a concrete benchmark based on this approach, tailored towards testing LLMs concerning Python code generation. However, our approach is LLM- and programming language-agnostic, allowing future testing of other LLMs across various programming languages.

Question Templates and Instances. A *question template* is a programming problem expressed in natural language, featuring one or more parameters. Recall the template of Figure 1a, which takes integer parameters p_1 and p_2. Instantiating this template with $p_1 = 1$ and $p_2 = 8$ yields the *question instance* of Figure 1b (instantiated parameters are shown in bold for clarity). Each question template should be accompanied by a

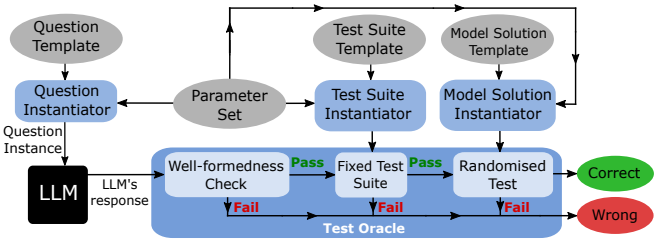

Fig. 2: Overview of our benchmarking approach

parameter set: a set of suitable parameter valuations that are meaningful for the question template. The Turbulence benchmark described in Section III is equipped with a generator that automatically produces a suitable parameter set of a desired size for a given question template. A question template can be automatically instantiated with a range of parameter values drawn from its parameter set, leading to a set of question instances that can be presented to an LLM (see Figure 2).

Intuitively, since question instances from the same template differ only by parameter values, they should all be just as easy or hard to solve as each other. Thus, it is noteworthy when an LLM solves some but not all instances from a template.

Assessing Correctness: Oracle Templates. To assess whether an LLM has returned a *correct* solution to a question instance, the benchmark designer must provide an *oracle template* for each question instance. This comprises: (1) a *fixed test suite*—a set of unit tests, parameterised with the same parameters as the question template, which when instantiated provides a concrete test suite for the question instance; (2) a *model solution template*, which can be instantiated to provide a correct solution for any question instance; and (3) a *random input generator*, which facilitates fuzz testing of solutions as described further below.

To illustrate this, consider the question template of Figure 1a. The oracle template associated with this question template comprises multiple *parameterised* test cases. One of these is shown in Figure 1c, and refers to parameter p_2 from the question template. When the question template is instantiated with $p_1 = 1$ and $p_2 = 8$, as shown in Figure 1b, the oracle template is also instantiated with these parameters. This transforms the parameterised test case of Figure 1c into the *concrete* test case of Figure 1d, which is suitable for assessing the correctness of a solution to the concrete question instance of Figure 1b. Furthermore, Figure 1e shows a parameterised model solution for the question template, again expressed in terms of the parameters p_1 and p_2, while Figure 1f shows a concrete instantiation of this model solution for the given parameter values. This concrete model solution facilitates experimental comparison with an LLM-generated solution on arbitrary input values. The random input generator component of the oracle template (not shown in Figure 1) provides a means of generating a stream of input values at random to support this kind of comparison.

Armed with these components, a code solution returned by an LLM in response to a question instance is deemed *correct*

if and only if all of the following hold (see the "Test Oracle" component of Figure 2): the LLM solution is *well-formed* (syntactically correct and conforming to any static typing rules of the programming language); the LLM solution passes all tests in the *fixed test suite* (instantiated with the parameters associated with the question instance); and the LLM solution yields the same result as the *model solution* (again, instantiated with the parameters of the question instance) when applied to a number of random inputs generated by the input generator. The approach of comparing the LLM solution with a model solution using randomly-generated inputs is a special case of fuzzing known as *random differential testing* [37]. The combination of testing via a fixed test suite and through random differential testing helps to ensure that the LLM solution works on particular important edge cases (provided by the fixed tests), as well as on a wider range of examples (from the randomised input generator). The user can control the amount of randomised testing per question instance.

Avoiding Ambiguity. It would be unfair to penalise an LLM for failing test cases that check aspects of a question whose solution is open to multiple interpretations. The designer of a question and oracle template must either (a) state the question precisely, without ambiguity, or (b) design the oracle template to avoid testing solutions in ambiguous parts of the input space. For example, the question template in Figure 1a avoids ambiguity by specifying that list indices are *inclusive*. Alternatively, this clarification could be omitted, and the oracle template could be adjusted to exclude test cases with even integers at indices p_1 and p_2, ensuring the oracle does not distinguish between solutions treating index ranges as inclusive or exclusive.

In Section III we explain how we used feedback from human programmers to avoid ambiguity in Turbulence.

Assigning Correctness Score. An oracle template provides a means for assigning a pass/fail result to an LLM's solution for a question instance. We explain how these results are combined into an overall score for each question template, reflecting the LLM's effectiveness in solving that question neighbourhood. Given the non-deterministic nature of LLMs, multiple independent queries per question instance are necessary.

Definition 1 (Correctness Score): Let L be an LLM under evaluation. Let Q be a question template with M associated parameter valuations (so that M distinct question instances are derived from Q). Suppose that the LLM is queried N times per question instance, and let $L_i^j(Q)$ denote the result returned by L the jth time that it is queried with question instance i of Q. Let $Oracle(L_i^j(Q))=1$ if this result is deemed correct according to the oracle template and 0 otherwise. The *correctness score*, $CorrSc$, for question template Q with respect to LLM L, $CorrSc(Q, L)$, is then defined as follows:

$$CorrSc(Q, L) = \frac{\sum_{i=1}^{M} \sum_{j=1}^{N} Oracle(L_i^j(Q))}{M \times N}$$

This is the mean over the correctness of all solutions returned by the LLM, where an individual solution is given a score of either 0 or 1. It yields a score in the range $[0, 1]$ for each question template Q. Rather than featuring $M \times N$ discrete question instances, our approach entails the usage of N identical collections of M unique question instances. This design choice is motivated by the inherent non-deterministic nature of LLMs, allowing multiple attempts with the same question instance to assess the LLM's ability to generate correct responses.

While the $pass@k$ metric [5], commonly used for evaluating LLMs in code generation, is a well-regarded measure, it is not suitable in the context of our study as our goal was to assess the overall correctness of each model. The $pass@k$ metric [5] focuses on whether at least one correct solution is found within the first k attempts. In our study, each prompt was sent five times ($k = 5$). Consider the following hypothetical results for a single prompt: incorrect (attempt 1), correct (attempt 2), incorrect (attempt 3), correct (attempt 4), and incorrect (attempt 5). In this case, $pass@5$ would be 100% since at least one correct answer is provided, but the overall correctness (i.e. the proportion of correct answers) is 40% (or 0.4, as described in Definition 1). We focused on the CorrSc metric to evaluate overall correctness.

III. The Turbulence Benchmark

Based on the approach described in Section II we have created a novel benchmark, Turbulence, for evaluating the correctness and robustness of instruction-tuned LLM for code. Turbulence focuses on the generation of Python code due to the language's popularity and the ample Python training data available for LLMs.

To create the benchmark, we developed a diverse set of 60 Python problem-solving questions encompassing fundamental concepts and basic data structures ensuring comprehensive and balanced coverage of key topics. No specific methodology or framework was followed for the selection of these questions, and to the best of our knowledge, no existing research outlines best practices for question formulation in this context. Furthermore, as discussed later, we deliberately avoided reusing publicly available questions to mitigate potential training bias.

Table I provides a broad categorisation of the 60 question templates into six distinct problem groups, further subdivided into subgroups. The questions utilise a wide variety of Python data types, either explicitly mentioned in the questions or implicitly required in the solutions. These include list (43 questions), integer (35 questions), boolean (60 questions), string (39 questions), set (9 questions), tuple (4 questions), and NumPy matrix (2 questions). Since some questions span multiple problem groups and data types, the counts in Table I and the data type totals exceed 60. Additionally, the subgroup counts within each problem group may exceed the total for their respective group, as certain questions pertain to more than one subgroup.

Turbulence comprises 60 question templates, each featuring at least one parameter, where each parameter is either a numerical value or a string. Each question template is equipped with an associated oracle template: a fixed test suite, random

input generator and model solution, as described in Section II. Every question template is accompanied by a parameter set of size 100, yielding 100 question instances per template. Hence, a total of 6,000 question instances are generated by Turbulence. For each question template, the parameter set was created by choosing a number of natural or evidently interesting parameter valuations (e.g. to exercise edge case behaviour), and thereafter populated with random valuations, restricted to well-formed valuations. For example, with respect to the question template of Figure 1a we would not allow negative values or values such that $p_2 < p_1$.

To avoid problems of bias occurring due to LLMs having been exposed to questions during the training, we decided to write question templates ourselves, from scratch, rather than seeking existing questions available on the internet. This was done to ensure that the LLMs were not able to simply regurgitate previously learned information, but instead had to generate new and creative responses. We were also careful not to put our questions online publicly before running experiments against LLMs. We used only a small selection of the most trivial questions when undertaking preliminary evaluation against LLMs during the construction of Turbulence, to guard against the possibility of these (closed source) LLMs learning from our interaction with them.

To ensure the clarity of question templates and the correctness of test oracles, we asked two experienced Python programmers to solve an instance of each question template independently, cross-checking their solution against our test oracle, and soliciting their feedback about potential ambiguity in the question. This led us to improve the wording of several question templates and fix several bugs in our test oracles.

Creating our own questions has its pros and cons. As argued above, using previously-unseen questions minimises problems of training-related bias, but it could arguably be more interesting to have a benchmark based on real-world programming challenges faced by developers "in the field". While the true role of LLMs in software engineering is solving real-world programming tasks, to have any chance of being useful in such contexts they should *at least* be capable of solving the kinds of programming problems that beginner to intermediate programmers would be capable of solving. Also, we emphasise that Turbulence is just one example of our proposed approach in Section II. The enduring value of our research lies in the approach itself, which could be retargeted to use alternative questions.

We deliberately included questions that, while uncommon in typical development scenarios, are pertinent to evaluating the reasoning capabilities of LLMs. For instance, a prompt like *"Write a function called 'all_ints_exclusive' that takes one argument, a list of integers, and returns the list of all elements from index 0 to index 1, both exclusive"* serves as an edge case designed to test the model's ability to comprehend and execute nuanced instructions. A primary goal of our benchmark is to assess whether LLMs are genuinely exhibiting emergent reasoning abilities. True reasoning capability should enable a model to solve not only standard problems but also edge cases that deviate from common patterns. While a human developer is unlikely to craft such an edge-case prompt, it is important to consider the evolving contexts in which LLMs are deployed. LLMs are increasingly being used in the back-ends of systems (such as integrated development environments) where prompts are generated programmatically rather than being written by humans. In these automated systems, the likelihood of encountering edge cases rises, as the prompts may not undergo human refinement or oversight. Auto-generated prompts are inherently more prone to exhibiting unusual or unexpected parameters, making it essential for LLMs to handle them effectively. Moreover, evaluating out-of-distribution robustness has been recognised as a critical aspect in the field of NLP: as highlighted by Yuan et al. [38], assessing how models perform on data that falls outside the distribution of their training data is necessary for understanding their generalisation capabilities and identifying potential weaknesses.

We distinguish between (a) the underlying conceptual framework of Turbulence and (b) the specific empirical find-

TABLE I: A classification of the Turbulence questions into problem groups and subgroups

Problem Group	**Problem Subgroup**	**Question Count**
List Manipulation	Total	40
	Slicing	21
	Indexing	14
	Filtering	16
	Element-based Operations	7
	Summation	6
	Sorting/Order-based Operations	6
	Element Insertion	2
	Count elements	2
	Circular Lists	1
String Manipulation	Total	16
	Character Insertion	2
	Character Removal	3
	Substring/Character Extraction	4
	Palindrome Operations	4
	Anagram Detection	2
	Sorting	2
Set Manipulation	Total	9
	Add Elements	6
	Subset/Superset Operation	1
	Counting Subsets	1
	Union	1
Searching	Total	36
	Linear Search	12
	Binary Search	8
	Index-based Search	10
	String Search	6
Copying	Total	10
	Deep Copy	4
	Shallow Copy	3
	Copy Sublist	3
Mathematical Problems	Total	31
	Arithmetic Operations	7
	Factorial Calculations	5
	Prime Checking	4
	Composite Checking	3
	Factorisation	4
	Special Sequences	3
	Combinatorial Problems	5

ings of this study. It is evident that (a), the concept of using neighbourhoods to identify reasoning discrepancies in LLMs, could be extended to test other LLMs (with a small amount of engineering effort specific to each model) and could also be adapted for other programming languages (requiring additional engineering effort for each language). However, concerning (b), we do not expect our specific findings to generalise directly to other LLMs or programming languages. Instead, we anticipate discovering different deficiencies, similar to how applying a software testing technique to different systems under test reveals distinct bugs. Our findings show that our approach effectively provides valuable insights into code generation.

Practical Issues. Implementing our approach necessitates some level of prompt engineering [39] to enhance the chances of obtaining source code from an LLM. In our initial experiments with the models discussed in Section IV, we observed that appending a simple prefix requesting that Python code be enclosed within triple backticks proved effective. The returned code could then be extracted by locating the section between the triple backticks.

IV. Experimental Evaluation

We now present results from running Turbulence against a range of LLMs.

A. Experimental Setup

We have gathered results running Turbulence against five LLMs: *GPT-4*, *GPT-3.5-turbo*, *Command*, *CodeLlama:7B:4-bit-quantised*, and *CodeLlama:13B:4-bit-quantised*.

GPT-4 [30], by OpenAI [40], is a large multimodal text generation model. GPT-3.5-turbo is the most advanced in the 3.5 series, trained on text and code up to Q4 2021 [31]. Cohere's 52-billion-parameter Command model [33] generates text from user commands. Meta's CodeLlama family [41] offers coding-specialised models (7B, 13B, 34B, 70B), including instruct-tuned versions. Ollama [36] provides quantised versions like CodeLlama:7B:4-bit-quantised and CodeLlama:13B:4-bit-quantised, which run on standard machines with 4GB and 8GB of RAM. We selected these models due to their low resource demands: *GPT-4*, *GPT-3.5-turbo*, and *Command* are hosted remotely, while *CodeLlama:7B:4-bit-quantised* and *CodeLlama:13B:4-bit-quantised* are small enough that they can be run locally as described below.

To maintain conciseness, in the rest of this paper, we refer to *CodeLlama:7B:4-bit-quantised*, *CodeLlama:13B:4-bit-quantised*, and *GPT-3.5-turbo* as *CodeLlama-7*, *CodeLlama-13*, and *GPT-3.5*, respectively. *LLM configuration* denotes an LLM combined with a specific temperature setting and $t=0$ and $t=D$ refer to the configurations of the LLM with temperature settings of 0 and default, respectively.

We accessed proprietary models *GPT-4*, *GPT-3.5*, and *Command* via their commercial APIs. Initially, we minimised Turbulence-related queries to these models to prevent potential bias, making only a few simple queries to test and debug the Turbulence infrastructure.

We downloaded *CodeLlama-7* and *CodeLlama-13* from the Ollama website [36] and ran them on a MacBook Pro with an Apple M1 Pro CPU and 16GB RAM, running macOS 14.0. To prevent bias, our queries never included sample solutions or hints about the correctness of previous responses from the LLM under test.

We evaluated all LLM-generated solutions for correctness on a desktop machine with an Intel Core i7-12700 CPU and 16GB RAM, running Ubuntu 22.04.2.

Every LLM has a user-determined *temperature* parameter that controls output randomness. Lower temperatures reduce randomness, improving quality but decreasing diversity [42], [43], while higher temperatures increase randomness, enhancing creativity. In addressing **RQ2**, we focused exclusively on comparing the models' behaviour at two specific settings: their default temperature and a temperature of zero. A temperature of zero was chosen to evaluate the performance of LLMs in as deterministic a context as possible, causing the models to select the most probable next token at each step. The default temperatures varied across LLMs, as developers fine-tuned them for optimal performance, balancing diverse outputs and coherence. Since these default temperatures reflect the intended behaviour envisioned by the developers, we used them to assess the performance of LLMs in their standard operational settings. Our goal was to analyse the shift from non-deterministic behaviour at default temperature to maximum determinism at a temperature of zero, offering clear insights into how determinism affects LLM performance without the complexity of varying randomness. We excluded broader temperature values to keep our experiments tractable.

Due to the stochastic nature of LLMs, repeat runs of experiments are necessary. At the same time, access to commercial LLMs (i.e. *GPT-3.5*, *GPT-4*, and *Command*) is costly, with variable query times. We ran the full benchmark 5 times for each LLM configuration.

While a temperature of 0 should cause the LLMs to behave in a highly deterministic manner, our initial mock tests revealed that LLMs occasionally yielded varying answers. This non-deterministic behaviour may be due to several factors, including non-deterministic GPU operations, memory access patterns, and numerical precision [44] and the inherent randomness from sampling, even at a temperature of 0 [45].

Our results are thus based on a comprehensive set of 300,000 LLM responses (i.e. number of models $\times$ number of configurations per model $\times$ number of prompts per each model's configuration $\times$ number of repeat runs $= 5 \times 2 \times 6000 \times 5 = 300000$). For a consistent comparison of experimental results, we used the same random seed when generating parameters for each question template.

B. Results Based on CorrSc

Figure 3 summarises how well each LLM configuration solved question templates. Recall from Definition 1 that we obtain the correctness score, $CorrSc$, for each question template and LLM configuration. In Figure 3, the first and last bars of each graph indicate how many question templates each

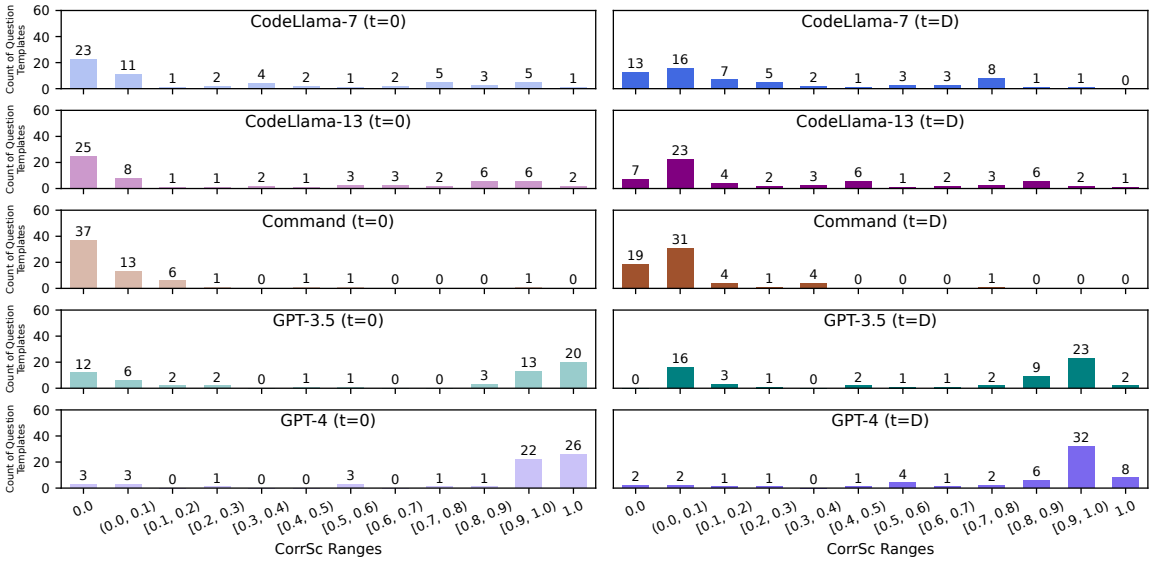

Fig. 3: *CorrSc* of question templates across the LLM configurations evaluated

QT: *Write a function called 'sum_of_multiples' that takes one argument, a positive integer, and returns the sum of the first* ***p*** *positive multiples of the given integer.*

(a) $p = 51$, correct answer

```
def sum_of_multiples(n):

  if n <= 0:
    return 0

  sum = 0
  for i in range(1,52):
    sum += i * n

  return sum
```

(b) $p = 56$, wrong answer

```
def sum_of_multiples(n):
  if n <= 0:
    raise ValueError("n
        must be a
        positive
        integer")
  sum = 0
  for i in range(1, 57):
    if i % n == 0:
      sum += i
  return sum
```

Fig. 4: Consistent failure, *CodeLlama-7* ($t = 0$)

LLM configuration either completely failed (scoring 0) or perfectly succeeded (scoring 1.0). The intermediate bars represent scores in ranges of size 0.1. For each LLM configuration, the corresponding graph shows the number of question templates whose *CorrSc* fell within each range. The density of question templates is mostly clustered around high or low score ranges, with no abrupt changes in the middle. For *CodeLlama-7*, *CodeLlama-13*, and *Command* at both temperatures, the data density is mostly in the lower score ranges, indicating weaker performance. In contrast, *GPT-3.5* and *GPT-4* show superior performance with more templates in the higher score ranges. However, there are numerous question templates where the LLMs failed to address all instances. Regarding **RQ1**, the many question templates not scoring 1.0 highlight the limited robustness of LLMs in addressing question neighbourhoods.

Addressing **RQ2**, examining each vertical pair of plots in Figure 3 shows that with temperature zero, the data distribution shifts to the left side for *CodeLlama-7*, *CodeLlama-13*, and *Command*; and to both sides for *GPT-3.5* and *GPT-4* indicating that LLMs are more likely to either fail consistently or succeed consistently for a given question neighbourhood.

C. Results Based on Distinct Categories

Recall that each question template is instantiated to yield a neighbourhood of 100 instances, each using distinct parameters, and each instance is given to the LLM across 5 rounds. We categorise LLM performance for a question into four distinct categories: *perfect failure*, where the LLM does not ever return a correct result ($CorrSc = 0.0$, i.e. the LLM cannot solve this neighbourhood of questions at all); *perfect success*, where the LLM always returns a correct result ($CorrSc = 1.0$, i.e. the LLM can solve this neighbourhood effortlessly); *consistent failure*, where the LLM returns at least one correct result but where there is at least one instance for which the LLM returns an incorrect result across all 5 rounds (the LLM appears to be completely blocked on at least one question instance); and *random failure*, where the LLM returns at least one incorrect result, but there is no instance for which the LLM returns an incorrect result across all 5 rounds (the LLM is not completely blocked on any question instance). This category describes cases where the LLM *does* appear capable of generalisation, solving every instance in a neighbourhood in at least one round, with sporadic failures due to its stochastic nature, not necessarily a lack of reasoning ability.

The *consistent failure* category is particularly noteworthy as it suggests a reasoning gap—an inability to generalise, since the LLM *can* sometimes solve instances arising from the question template, but there are certain instances that it does not manage to solve on any round. Figure 4 shows a *consistent failure* example: when the question template (referred to as QT in the figure) was instantiated with $p = 51$ (Figure 4a), *CodeLlama-7* at $t = 0$ solved the question instance

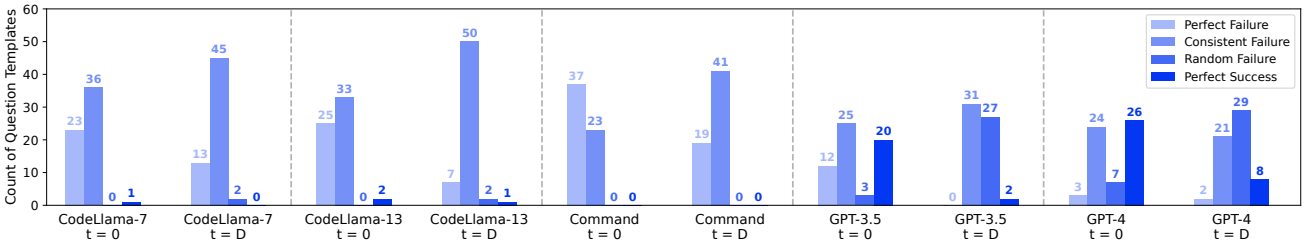

Fig. 5: Distribution of Turbulence question templates across result categories for the LLM configurations evaluated

correctly in all five rounds (that is at least one correct answer); however, when the question template was instantiated with 56 (Figure 4b), the LLM consistently generated wrong answers in all five rounds. The *consistent failure* category highlights LLM robustness issues where failures are consistent for particular question instances, and not solely due to the LLM's stochastic nature. Resource and financial constraints limited our experiments to five rounds; consistent failure across more rounds would offer stronger evidence of LLM reasoning gaps.

Figure 5 shows results according to these categories. The *perfect failure* and *perfect success* bars mirror the $CorrSc = 0.0$ and $CorrSc = 1.0$ bars of Figure 3, respectively.

To answer **RQ1**, Figure 5 indicates a lack of robustness in the performance of the LLMs considered in this study. Notably, there is a significant count of question templates categorised as *consistent failure* for each LLM configuration. *CodeLlama-13* ($t{=}D$) exhibits the highest count (50 question templates), while *GPT-4* ($t = D$) has the smallest count (21 question templates).

To address **RQ2**, lowering the temperature from the default value to zero led to a reduction in the number of question templates classified under the *consistent failure* category, with the exception of *GPT-4*. A similar trend was observed in the *random failure* category, where all models except *Command* showed a decrease in the number of question templates. This effect was especially noticeable in the *GPT* models.

V. Exploring Reasons for Failure

In this section, we address **RQ3** by examining the main errors in the LLM's code responses that caused them to be incorrect. Table II presents 9 failure categories, which we now illustrate with selected examples. The initial three rows in Table II, i.e. *no function*, *wrong function name*, and *wrong count of arguments*, align with the *well-formedness check* of the Turbulence test oracle (Figure 2). Recall that each question instance asks the LLM to write a *Python function* bearing the *specified name* and *count of arguments* (Figure 1b).

For the remaining six categories, the *syntax error* category is identified via the Python parser, the *static type error* category is identified using the Pylint linter [46], and the remaining categories are identified by running the test oracle for a question instance.

No function. This category includes cases where the LLM failed to generate a Python function, occurring with *Command* ($t{=}D$) and *GPT-3.5* ($t{=}D$) for certain questions.

Wrong function name. This occurs when the LLM generates a function with a different name than requested, causing failures as each Turbulence test oracle requires specified function names, and there is no systematic, reliable way to fix function names, especially when responses include multiple functions. At $t{=}D$, all LLMs exhibit this problem in some cases.

Wrong number of arguments. Here, the LLM generates a correctly-named function but with the wrong number of arguments, making it incompatible with the test oracle. All LLMs except *GPT-3.5* and *GPT-4* experienced this problem. Table II shows this issue was less frequent at $t{=}0$.

Syntax errors. Here, the LLM output could not be parsed due to Python syntax errors, such as unmatched parentheses, misaligned brackets, incorrect indentation, missing comment indicators (#), using `else if` instead of `elif`, missing `except` or `finally` clauses after a `try` block, invalid variable names (e.g. `6th_number`), and invalid assignment targets (e.g. `len(binary) -= 1`). Syntax errors decreased when LLMs were run at temperature 0.

Static type errors. This is where the integrated Pylint linter [46] identified static type errors in generated code, including undefined variables (e.g. using `math.gcd` without importing the `math` library), and the use of Python keywords as variable names (e.g. `sum = sum(multiples)`). Lowering the temperature from default to 0 in all LLMs (except *CodeLlama-13* and *Command* models) reduced both the quantity and variety of static type errors in the generated responses.

Resource exhaustion error. This category involves cases where the generated code exceeded time or memory resources during execution, observed across all LLMs, despite more efficient solutions being available. For example, *GPT-4* ($t = D$) was asked to write a function to return the number of subsets of size 54 from a set of elements. The LLM used `len(list(combinations(elements,54)))`, which becomes resource-intensive for sets larger than 60. A more efficient solution is to use `math.comb(len(elements),54)`, which works efficiently with any size of a set.

Runtime errors. This category covers cases where executing the generated code caused Python errors. For example, `for i in range(2,x+1)` where `x` was a tuple, making concatenation of an 'int' and a 'tuple' invalid.

Assertion errors and fuzzing failures. This category covers cases where the code executed but was functionally incorrect,

TABLE II: Percentage of responses per LLM configuration out of 30,000, categorised by test failures and passes

Categories	*CodeLlama-7*		*CodeLlama-13*		*Command*		*GPT-3.5*		*GPT-4*	
	$t=0$	$t=D$	$t=0$	$t=D$	$t=0$	$t=D$	$t=0$	$t=D$	$t=0$	$t=D$
no function	0.00%	0.00%	0.00%	0.00%	0.00%	0.45%	0.00%	0.13%	0.00%	0.00%
wrong function name	0.00%	0.02%	0.00%	0.14%	0.00%	0.02%	0.00%	0.07%	0.00%	0.24%
wrong count of arguments	3.73%	4.04%	0.82%	2.69%	0.00%	0.02%	0.00%	0.00%	0.00%	0.00%
syntax error	4.90%	11.72%	0.47%	0.83%	1.14%	1.96%	0.00%	0.04%	0.00%	0.07%
static type error	3.57%	9.34%	10.28%	9.29%	7.68%	5.88%	0.80%	0.91%	0.03%	0.05%
resource exhaustion	0.63%	0.63%	0.02%	0.12%	0.04%	0.37%	2.26%	1.22%	0.93%	0.91%
runtime error	7.65%	8.10%	4.74%	6.85%	10.81%	13.40%	1.95%	2.98%	0.56%	1.04%
assertion error	45.65%	36.35%	44.09%	43.90%	66.91%	64.35%	27.15%	26.65%	9.31%	10.89%
fuzzing failure	4.92%	4.98%	6.85%	6.93%	7.86%	7.28%	6.17%	6.75%	4.38%	4.66%
passed	28.95%	24.81%	32.73%	29.25%	5.57%	6.27%	61.67%	61.26%	84.79%	82.15%

QI: *Write a function called 'prime_factors' that takes one argument, a list of positive integers, and returns the set of all prime factors of the integer at index **85**.*

```
def prime_factors(lst):
  number = lst[85]
  factors = set()
  for i in range(2, int(number/2) + 1):
    while number % i == 0:
      factors.add(i)
      number = number/i
  if number > 2:
    factors.add(number)
  return factors
```

Fig. 6: Functional bug, code generated by *GPT-3.5* (t=D)

not fitting the errors in the first seven rows of Table II. We identified five main causes of functional errors and grouped them accordingly. The first group involves incorrect ranges. For questions requiring an index or number range, the common bug was either excluding the range or inaccuracies in the lower or upper bounds. For example, when *CodeLlama-13* ($t=0$) was asked to return all list elements from index 55 to 98, both inclusive, it returned elements from $[54:99]$ instead of the expected range, $[55:99]$.

The second group includes responses where redundant code rendered the functionality incorrect, though the code would have been correct without it.

The third group includes responses with an incorrect logical order of tasks. For example, when *CodeLlama-7* (t=D) was asked to return the second largest number from a specified index range in a list, its generated code sorted the entire list first, then sliced it and returned the second element. The correct approach is to slice the specified range first, then sort it, and return the second element.

The fourth group includes responses where the LLM misunderstood the question. For example, *Command* ($t=0$) was asked if the integer at index 85 of a list was a perfect number, it generated the incorrect code: `return nums[85]==64820`, where `nums` was a placeholder for the list.

The fifth group includes partially correct responses that miss certain input cases. In Figure 6 (where QI refers to *question instance*), the code fails to return $\{2\}$ when the list contains 2 at index 85 due to the condition `if number > 2` instead of `if number >= 2`, which would correctly identify 2 as a prime number.

Passed. As shown in the "passed" row in Table II, reducing the temperature to 0 increased the number of correct answers for all models except *Command*. *GPT-4*, which outperformed *GPT-3.5*, solved over 82% of question instances in both settings, while *Command* underperformed across all setups. Additionally, *CodeLlama-13* outperformed *CodeLlama-7*.

Patterns in parameter values leading to errors. Across most question templates we could not discern patterns among parameter values that led to incorrect LLM answers. The values were highly diverse with no identifiable trends. However, when examining cases where the LLM succeeded with most of the parameters in a question neighbourhood but failed with specific parameters, i.e. $0.9 \leq CorrSc < 1.0$, we identified certain patterns. *CodeLlama-7*, *CodeLlama-13*, *GPT-3.5*, and *GPT-4* generated incorrect answers for question instances involving indices or number ranges, particularly at temperature 0. In contrast, *Command* showed no pattern at any temperature. The pattern involved indices or bounds that were identical, where the lower bound was 0, or where the difference between bounds was 1. In other words, if x denotes a non-negative integer, the ranges were: $(0,x), (x,x), [0,x], [x,x]$, and $(x, x+1)$, with parentheses indicating exclusivity and square brackets indicating inclusivity. Additionally, *GPT-3.5* made errors when inserting a character before another in a string if either was a space character.

VI. Threats to Validity

Our assessment of LLM correctness relies on (a) unambiguous questions, (b) accurate test oracles that do not mis-classify a correct answer as incorrect, and (c) strong test oracles capable of catching errors. For (a) and (b), although skilled Python programmers reviewed our questions and test oracles (see Section III), this may not have eliminated all potential ambiguities and oracle errors. For (c), we combine regression testing and random differential testing to thoroughly evaluate LLM responses, but testing is inherently incomplete.

Our findings are limited to the LLMs we evaluated; however, Turbulence supports integrating additional LLMs in the

future. To balance costs, we set M and N (Definition 1) to 100 parameter settings per question for diversity and 5 runs to address variance, reflecting our resource constraints. Larger values would be preferable with increased resources.

To avoid training data bias, we developed custom question templates instead of using internet-sourced "real-world" questions. While our questions may resemble online ones, they are tailored for the question neighbourhood approach. In practical tasks, parameters in Turbulence questions are usually kept as formal parameters, awaiting user input. However, artificial questions in Turbulence offer key advantages: (i) focusing on hard-to-find edge cases and (ii) enabling easy reproducibility and comparison across LLMs. Our framework remains flexible for future studies with different question sets.

VII. Related Work

We categorise prior work on LLM robustness and correctness for code into the following themes.

Correctness and evaluation benchmarks. Various benchmarks and datasets, such as HumanEval [5], APPS [6], MBPP [11], CodeContests [47], CodeXGLUE [48], and examples from LeetCode [7] have been used to evaluate the correctness of LLM-generated code, and the effectiveness of LLMs with respect to code generation has been investigated for a specific languages such as Python [49] and Verilog [50], as well as for particular programming paradigms, such as the use of classes in an object-oriented setting [51]. Xu et al. systematically evaluated multilingual LLMs for code correctness and perplexity [12]. Moradi et al. compared Copilot's solutions to human-generated code on algorithmic problems [52]. The focus of CCTest [53] is on improving LLM-based code completion by ensuring structural consistency using Levenshtein edit distance [54], and detecting errors through mutations that maintain consistency of program structure. Wong et al. assessed Copilot's code quality by formally verifying that generated code meets predefined specifications [55]. Rajan et al. proposed KONTEST to detect inconsistencies in LLM outputs using knowledge graph-based test cases and metamorphic and ontological oracles [56]. Dozono et al. evaluated LLMs for detecting common weaknesses and introduced CODEGUARDIAN to enhance accuracy and speed in VS Code [57]. Recent works have focused on the importance of fine-tuning in LLM performance [14], and on evaluating self-consistency of LLMs in code generation and comprehension tasks [58].

In contrast to these works, which mainly focus on the absolute performance of LLMs on specific code generation tasks, our approach evaluates LLM code generation capabilities across *neighbourhoods* of related question instances, allowing the identification of discontinuities in reasoning ability. This can offer insights into how LLMs handle a variety of strongly-related tasks in a given problem space, which is under-explored by these prior works. Additionally, unlike most previous approaches that rely on manually crafted test suites, we automate testing by combining fixed test suites with fuzzing as complementary techniques [59].

Robustness. Several studies have investigated LLM robustness to syntactic variations that preserve semantics, e.g. by modifying problem descriptions without altering semantics [20], [23], perturbing prompts (with the finding that slight perturbations can significantly impact model performance) [18], [19], and changing method names [60]. Various works have focused on enhancing robustness: CLAWSAT utilises contrastive learning with adversarial views and staggered adversarial training for this purpose [61], the CoTR framework [21] defends code translation models against adversarial attacks through syntactic transformations and data augmentation with semantically equivalent code examples, CodeBERT-Attack highlights vulnerabilities and suggests adversarial training examples for model improvement [62], the CODA framework aims to enhance model robustness by generating adversarial examples from semantically similar inputs [63], and the CARROT framework focuses on robustness detection, measurement, and enhancement in the context of code-focused LLMs [64]. Numerous other approaches focus on the problem of identifying a lack of robustness in models [16], [22], [65]–[70].

Among these works, the study by Shirafuji et al. [20] is the most closely related to our research. However, there is a key difference in focus. Their study examines how syntactic modifications, such as altering variable names or rephrasing prompts, influence the correctness and quality of generated code while maintaining the underlying task. In contrast, our research explores how LLMs perform when faced with a question neighbourhood—a set of *semantically similar but distinct tasks*. By leveraging parameterised question templates, we systematically investigate the models' ability to generalise and identify gaps in their performance.

VIII. Conclusions and Future Work

We have introduced a new method for assessing the correctness and robustness of LLMs with respect to code generation, based on the notion *question neighbourhoods*. Being able to assess the performance of an LLM across a question neighbourhood makes it possible not only to identify specific problem instances that an LLM cannot solve, but to identify gaps in an LLM's ability to perform general reasoning in a particular problem space. We have put this into practice via Turbulence, the first benchmark to systematically evaluate code-generating LLMs using question neighbourhoods. Experiments with five models showed that *GPT-4* consistently outperformed the other evaluated models, but that all models demonstrated a lack of robustness in certain question neighbourhoods. Lowering the temperature to zero improved correctness scores (except for *Command*) and reduced error diversity.

Interesting avenues for future research include assessing the impact of quantisation on LLM performance, and developing Turbulence-like benchmarks for code-infilling models.

IX. Data Availability Statement

The source code for Turbulence, all question and oracle templates, together with all results, are available in the Zenodo repository [71].

REFERENCES

[1] J. D. Weisz, M. J. Muller, S. Houde, J. T. Richards, S. I. Ross, F. Martinez, M. Agarwal, and K. Talamadupula, "Perfection not required? Human-AI partnerships in code translation," in *IUI*. ACM, 2021, pp. 402–412. [Online]. Available: https://doi.org/10.1145/3397481.3450656

[2] V. Lomshakov, S. V. Kovalchuk, M. Omelchenko, S. I. Nikolenko, and A. Aliev, "Fine-tuning large language models for answering programming questions with code snippets," in *ICCS*, ser. Lecture Notes in Computer Science, J. Mikyska, C. de Mulatier, M. Paszynski, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, Eds., vol. 14074. Springer, 2023, pp. 171–179. [Online]. Available: https://doi.org/10.1007/978-3-031-36021-3_15

[3] S. Zhang, L. Dong, X. Li, S. Zhang, X. Sun, S. Wang, J. Li, R. Hu, T. Zhang, F. Wu, and G. Wang, "Instruction tuning for large language models: A survey," *CoRR*, vol. abs/2308.10792, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.10792

[4] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," in *ICLR*. OpenReview.net, 2022. [Online]. Available: https://openreview.net/forum?id=gEZrGCozdqR

[5] M. Chen *et al.*, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[6] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with APPS," in *NeurIPS Datasets and Benchmarks*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html

[7] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub Copilot's code suggestions," in *MSR*. ACM, 2022, pp. 1–5. [Online]. Available: https://doi.org/10.1145/3524842.3528470

[8] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? Rigorous evaluation of large language models for code generation," in *NeurIPS*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html

[9] S. I. Ross, F. Martinez, S. Houde, M. J. Muller, and J. D. Weisz, "The programmer's assistant: Conversational interaction with a large language model for software development," in *IUI*. ACM, 2023, pp. 491–514. [Online]. Available: https://doi.org/10.1145/3581641.3584037

[10] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?" in *Proceedings of the 2023 ACM SIGSAC, CCS*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 2785–2799. [Online]. Available: https://doi.org/10.1145/3576915.3623157

[11] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[12] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming*, S. Chaudhuri and C. Sutton, Eds. ACM, 2022, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3520312.3534862

[13] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie, "A survey on evaluation of large language models," *ACM Trans. Intell. Syst. Technol.*, vol. 15, no. 3, pp. 39:1–39:45, 2024. [Online]. Available: https://doi.org/10.1145/3641289

[14] Z. Yuan, J. Liu, Q. Zi, M. Liu, X. Peng, and Y. Lou, "Evaluating instruction-tuned large language models on code comprehension and generation," *CoRR*, vol. abs/2308.01240, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.01240

[15] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on GitHub Copilot," in *ICSE*. IEEE, 2023, pp. 2149–2160. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00181

[16] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *ICSE*. ACM, 2022, pp. 1482–1493. [Online]. Available: https://doi.org/10.1145/3510003.3510146

[17] Z. Tian, J. Chen, and Z. Jin, "Code difference guided adversarial example generation for deep code models," in *ASE*. IEEE, 2023, pp. 850–862. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00149

[18] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia, R. Nallapati, M. K. Ramanathan, D. Roth, and B. Xiang, "ReCode: Robustness evaluation of code generation models," in *ACL*, A. Rogers, J. L. Boyd-Graber, and N. Okazaki, Eds. ACL, 2023, pp. 13 818–13 843. [Online]. Available: https://doi.org/10.18653/v1/2023.acl-long.773

[19] J. Döderlein, M. Acher, D. E. Khelladi, and B. Combemale, "Piloting Copilot and Codex: Hot temperature, cold prompts, or black magic?" *CoRR*, vol. abs/2210.14699, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2210.14699

[20] A. Shirafuji, Y. Watanobe, T. Ito, M. Morishita, Y. Nakamura, Y. Oda, and J. Suzuki, "Exploring the robustness of large language models for solving programming problems," *CoRR*, vol. abs/2306.14583, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.14583

[21] G. Yang, Y. Zhou, X. Zhang, X. Chen, T. Han, and T. Chen, "Assessing and improving syntactic adversarial robustness of pre-trained models for code translation," *CoRR*, vol. abs/2310.18587, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.18587

[22] M. Yan, J. Chen, J. M. Zhang, X. Cao, C. Yang, and M. Harman, "COCO: Testing code generation systems via concretized instructions," *CoRR*, vol. abs/2308.13319, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.13319

[23] M. Anand, P. Kayal, and M. Singh, "On adversarial robustness of synthetic code generation," *CoRR*, vol. abs/2106.11629, 2021. [Online]. Available: https://arxiv.org/abs/2106.11629

[24] T. Y. Zhuo, A. Zebaze, N. Suppattarachai, L. von Werra, H. de Vries, Q. Liu, and N. Muennighoff, "Astraios: Parameter-efficient instruction tuning code large language models," *CoRR*, vol. abs/2401.00788, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.00788

[25] Z. Yang, Z. Sun, T. Y. Zhuo, P. T. Devanbu, and D. Lo, "Robustness, security, privacy, explainability, efficiency, and usability of large language models for code," *CoRR*, vol. abs/2403.07506, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.07506

[26] M. Gardner, Y. Artzi, V. Basmova, J. Berant, B. Bogin, S. Chen, P. Dasigi, D. Dua, Y. Elazar, A. Gottumukkala, N. Gupta, H. Hajishirzi, G. Ilharco, D. Khashabi, K. Lin, J. Liu, N. F. Liu, P. Mulcaire, Q. Ning, S. Singh, N. A. Smith, S. Subramanian, R. Tsarfaty, E. Wallace, A. Zhang, and B. Zhou, "Evaluating models' local decision boundaries via contrast sets," in *EMNLP*, ser. Findings of ACL, T. Cohn, Y. He, and Y. Liu, Eds., vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1307–1323. [Online]. Available: https://doi.org/10.18653/v1/2020.findings-emnlp.117

[27] A. Saparov and H. He, "Language models are greedy reasoners: A systematic formal analysis of chain-of-thought," in *ICLR*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/forum?id=qFVVBzXxR2V

[28] F. Shi, M. Suzgun, M. Freitag, X. Wang, S. Srivats, S. Vosoughi, H. W. Chung, Y. Tay, S. Ruder, D. Zhou, D. Das, and J. Wei, "Language models are multilingual chain-of-thought reasoners," in *ICLR*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/forum?id=fR3wGCk-IXp

[29] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *NeurIPS*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[30] OpenAI, "GPT-4 technical report," *CoRR*, vol. abs/2303.08774, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.08774

[31] OpenAI, "GPT-3.5," https://platform.openai.com/docs/models/gpt-3-5, 2023.

[32] OpenAI, "Models," https://platform.openai.com/docs/models/models, 2023.

[33] Cohere, "Cohere's Command Model," https://docs.cohere.com/docs/the-command-model, 2023.

[34] Cohere, "The Cohere Platform," https://docs.cohere.com/docs/the-cohere-platform, 2023.

[35] Meta, "Introducing Code Llama, an AI Tool for Coding," https://about.fb.com/news/2023/08/code-llama-ai-for-coding/, oct 2023.

[36] Ollama, "Codellama," https://ollama.ai/library/codellama, oct 2023.

[37] W. M. McKeeman, "Differential testing for software," *Digit. Tech. J.*, vol. 10, no. 1, pp. 100–107, 1998. [Online]. Available: https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[38] L. Yuan, Y. Chen, G. Cui, H. Gao, F. Zou, X. Cheng, H. Ji, Z. Liu, and M. Sun, "Revisiting out-of-distribution robustness in NLP: Benchmarks, analysis, and LLMs evaluations," in *NeurIPS*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/b6b5f50a2001ad1cbccca96e693c4ab4-Abstract-Datasets_and_Benchmarks.html

[39] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," in *CHI '21: CHI Conference on Human Factors in Computing Systems*, Y. Kitamura, A. Quigley, K. Isbister, and T. Igarashi, Eds. ACM, 2021, pp. 314:1–314:7. [Online]. Available: https://doi.org/10.1145/3411763.3451760

[40] OpenAI, "About OpenAI," https://openai.com/about, 2023.

[41] Meta, "Introducing Code Llama, a state-of-the-art large language model for coding," https://ai.meta.com/blog/code-llama-large-language-model-coding/, oct 2023.

[42] M. Caccia, L. Caccia, W. Fedus, H. Larochelle, J. Pineau, and L. Charlin, "Language gans falling short," in *ICLR*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=BJgza6VtPB

[43] T. B. Hashimoto, H. Zhang, and P. Liang, "Unifying human and statistical evaluation for natural language generation," in *NAACL-HLT*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 1689–1701. [Online]. Available: https://doi.org/10.18653/v1/n19-1169

[44] NVIDIA, "Determinism in deep learning," https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9911-determinism-in-deep-learning.pdf, dec 2010.

[45] J. Gawlikowski, C. R. N. Tassi, M. Ali, J. Lee, M. Humt, J. Feng, A. M. Kruspe, R. Triebel, P. Jung, R. Roscher, M. Shahzad, W. Yang, R. Bamler, and X. Zhu, "A survey of uncertainty in deep neural networks," *Artif. Intell. Rev.*, vol. 56, no. S1, pp. 1513–1589, 2023. [Online]. Available: https://doi.org/10.1007/s10462-023-10562-9

[46] Logilab and P. contributors, "Pylint," https://pylint.pycqa.org/en/latest/index.html#pylint, nov 2023.

[47] Y. Li *et al.*, "Competition-level code generation with AlphaCode," *CoRR*, vol. abs/2203.07814, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2203.07814

[48] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *NeurIPS Datasets and Benchmarks*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html

[49] F. Tambon, A. M. Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and G. Antoniol, "Bugs in large language models generated code: An empirical study," *CoRR*, vol. abs/2403.08937, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.08937

[50] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated Verilog RTL code generation," in *DATE*. IEEE, 2023, pp. 1–6. [Online]. Available: https://doi.org/10.23919/DATE56975.2023.10137086

[51] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *ICSE*. ACM, 2024, pp. 81:1–81:13. [Online]. Available: https://doi.org/10.1145/3597503.3639219

[52] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "GitHub Copilot AI pair programmer: Asset or liability?" *J. Syst. Softw.*, vol. 203, p. 111734, 2023. [Online]. Available: https://doi.org/10.1016/j.jss.2023.111734

[53] Z. Li, C. Wang, Z. Liu, H. Wang, D. Chen, S. Wang, and C. Gao, "CCTEST: Testing and repairing code completion systems," in *ICSE*. IEEE, 2023, pp. 1238–1250. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00110

[54] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet physics. Doklady*, vol. 10, pp. 707–710, 1965. [Online]. Available: https://api.semanticscholar.org/CorpusID:60827152

[55] D. Wong, A. Kothig, and P. Lam, "Exploring the verifiability of code generated by GitHub Copilot," *CoRR*, vol. abs/2209.01766, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2209.01766

[56] S. S. Rajan, E. O. Soremekun, and S. Chattopadhyay, "Knowledge-based consistency testing of large language models," in *EMNLP*. Association for Computational Linguistics, 2024, pp. 10 185–10 196. [Online]. Available: https://aclanthology.org/2024.findings-emnlp.596

[57] K. Dozono, T. E. Gasiba, and A. Stocco, "Large language models for secure code assessment: A multi-language empirical study," *CoRR*, vol. abs/2408.06428, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.06428

[58] M. J. Min, Y. Ding, L. Buratti, S. Pujar, G. E. Kaiser, S. Jana, and B. Ray, "Beyond accuracy: Evaluating self-consistency of code large language models with IdentityChain," in *ICLR*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=caW7LdAALh

[59] I. Ciupa, B. Meyer, M. Oriol, and A. Pretschner, "Finding faults: Manual testing vs. random+ testing vs. user reports," in *ISSRE*. IEEE Computer Society, 2008, pp. 157–166. [Online]. Available: https://doi.org/10.1109/ISSRE.2008.18

[60] G. Yang, Y. Zhou, W. Yang, T. Yue, X. Chen, and T. Chen, "How important are good method names in neural code generation? A model robustness perspective," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 3, pp. 60:1–60:35, 2024. [Online]. Available: https://doi.org/10.1145/3630010

[61] J. Jia, S. Srikant, T. Mitrovska, C. Gan, S. Chang, S. Liu, and U. O'Reilly, "ClawSAT: Towards both robust and accurate code models," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER*, T. Zhang, X. Xia, and N. Novielli, Eds. IEEE, 2023, pp. 212–223. [Online]. Available: https://doi.org/10.1109/SANER56733.2023.00029

[62] H. Zhang, S. Lu, Z. Li, Z. Jin, L. Ma, Y. Liu, and G. Li, "CodeBERT-Attack: Adversarial attack against source code deep learning models via pre-trained model," *J. Softw. Evol. Process.*, vol. 36, no. 3, 2024. [Online]. Available: https://doi.org/10.1002/smr.2571

[63] Z. Tian, J. Chen, and Z. Jin, "Code difference guided adversarial example generation for deep code models," in *ASE*. IEEE, 2023, pp. 850–862. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00149

[64] H. Zhang, Z. Fu, G. Li, L. Ma, Z. Zhao, H. Yang, Y. Sun, Y. Liu, and Z. Jin, "Towards robustness of deep program processing models - detection, estimation, and enhancement," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, pp. 50:1–50:40, 2022. [Online]. Available: https://doi.org/10.1145/3511887

[65] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," in *AAAI*. AAAI Press, 2020, pp. 1169–1176. [Online]. Available: https://doi.org/10.1609/aaai.v34i01.5469

[66] J. Chen, Z. Pan, X. Hu, Z. Li, G. Li, and X. Xia, "Reasoning runtime behavior of a program with LLM: How far are we?" in *ICSE 2025*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 140–152. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00012

[67] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *ISSTA '22: 31st ACM SIGSOFT*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 39–51. [Online]. Available: https://doi.org/10.1145/3533767.3534390

[68] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 162:1–162:30, 2020. [Online]. Available: https://doi.org/10.1145/3428230

[69] S. Srikant, S. Liu, T. Mitrovska, S. Chang, Q. Fan, G. Zhang, and U. O'Reilly, "Generating adversarial computer programs using optimized obfuscations," in *ICLR*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=PH5PH9ZO_4

[70] M. Wei, Y. Huang, J. Yang, J. Wang, and S. Wang, "CoCoFuzzing: Testing neural code models with coverage-guided fuzzing," *IEEE Trans. Reliab.*, vol. 72, no. 3, pp. 1276–1289, 2023. [Online]. Available: https://doi.org/10.1109/TR.2022.3208239

[71] S. Honarvar and A. Donaldson, "ShahinHonarvar/Turbulence-Benchmark: Version 1.0," Jan. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.14732749