The "Question Neighbourhood" Approach for Systematic Evaluation of Code-Generating LLMs

Shahin Honarvar *Member, IEEE*, Marek Rei, Alastair Donaldson *Member, IEEE*

***Abstract*—We present the concept of a *question neighbourhood* for systematically evaluating 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 *many* questions in a neighbourhood but fails for particular parameter instantiations. We present experiments against 22 state-of-the-art proprietary and open-source LLMs, each at two temperature configurations. Our evaluation is based on three complementary scores: accuracy score, correctness-potential score, and consistent-correctness score. Our findings show that, across the board, Turbulence is able to reveal cases where LLMs do not behave in a correct and consistent manner, highlighting gaps in their 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 provides detailed insight into the behavioural characteristics of current code-generating LLMs. 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, potential correctness, consistent correctness, robustness, AI evaluation, code generation**

I. Introduction

LARGE Language Models (LLMs) have demonstrated significant advancements in code generation tasks, including translating between programming languages [1] and answering complex programming questions [2]. Although these models have become increasingly effective, they generate *incorrect* code [3]–[6], which poses significant challenges to their safe and reliable integration into mainstream software engineering, particularly in safety-critical environments where consistent and correct behaviour is essential.

In practice, models may produce correct code only intermittently or yield inconsistent behaviour, making it difficult for developers to trust their output [7], [8]. To support the wider adoption of LLMs in code-related tasks, there is a pressing need for evaluation frameworks that go beyond aggregate correctness and capture more nuanced behavioural properties.

While several works [3], [4], [9]–[12] have primarily focused on assessing correctness, and others [13]–[23] have investigated model behaviour under semantically equivalent prompt variations, our work complements these efforts by introducing a new perspective: rather than evaluating LLMs on isolated prompts or semantically equivalent variations, we assess their behaviour across *question neighbourhoods*—sets of related but semantically *non-equivalent* tasks derived from a shared template.

To characterise LLM performance in this setting, we use three complementary scores: (1) *Accuracy Score*, which measures the overall rate of correctness across all generated outputs; (2) *Correctness-Potential Score*, which captures whether the model produces at least one correct output for a given input; and (3) *Consistent-Correctness Score*, assessing the model's ability to consistently produce correct outputs for the same input across successive generations. This multifaceted evaluation provides a more complete picture of model behaviour than aggregate correctness alone.

Our contribution. Inspired by Gardner et al. [24], 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*. In Figure 1, the parameters p_1 and p_2, along with their respective instantiations, are highlighted in bold for clarity.

Each question template is paired with an associated *oracle template*. This includes a suite of parameterised unit tests,

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).

Shahin Honarvar, Marek Rei, and Alastair Donaldson are with the Department of Computing, Imperial College London, London, UK (email: s.honarvar21@imperial.ac.uk; marek.rei@imperial.ac.uk; alastair.donaldson@imperial.ac.uk)

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) 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) 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) 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) 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) Model solution instance from (e) with $p_1 = 1$ and $p_2 = 8$.

Fig. 1. Example of a question template, test case template, and model solution template, with an instantiation of each.

featuring the same parameters that appear in the question template. Figure 1c shows an example of a parameterised test case for the question template in 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, an LLM might successfully solve many instances of a question template, yet consistently yield incorrect solutions for specific parameter values. Conversely, it may generally fail to solve most instances in a neighbourhood, yet unexpectedly produce a correct solution for certain parameter values. Intuitively, since question instances within a neighbourhood differ only in their parameter values, they should be equally easy or difficult to solve. Thus, it is noteworthy when an LLM solves some, but not all, instances from a template. 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 [25]–[27]. 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 stand-alone problems (see Section VII) cannot yield such insights. Central to our method is the use of *question neighbourhoods*, which allow us to evaluate: (1) LLM *accuracy* (the proportion of correct code generations across all instances in a neighbourhood); (2) LLM *correctness potential* (the proportion of instances for which the LLM produces at least one correct response); and (3) LLM *consistent correctness* (the proportion of instances for which the LLM consistently generates correct responses across multiple generations).

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. It is equally important to note that (2) plays a critical role at present, as no alternative template sets yet exist. 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 [28]. 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 22 state-of-the-art proprietary and open source LLMs each of which was evaluated at two temperature settings. Our evaluation is guided by the following research questions about the instruction-tuned LLMs:

- **RQ1:** How do LLMs perform in terms of accuracy, correctness potential, and consistent correctness when faced with alterations within a question neighbourhood?
- **RQ2:** How does setting an LLM's temperature to zero for

maximum determinism affect its accuracy, correctness-potential, and consistent-correctness scores 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 reveal that although models such as *GPT-4o*, *GPT-4*, *Claude 3.5 Haiku*, *Claude 3.5 Sonnet*, *Mistral Large 2*, and *Qwen2.5-Coder-32B* exhibited strong performance across the dimensions of accuracy, correctness potential, and consistent correctness, all 22 LLMs assessed in this study demonstrated notable deficiencies in these areas when evaluated across diverse question neighbourhoods. 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. Statistical analysis revealed that setting the temperature to zero improved consistent correctness in a model-dependent manner, while effects on accuracy and correctness potential were limited, with only minor gains in some cases and no meaningful differences in most.

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. In Section V, we analyse the common issues in incorrect code generated by LLMs.

In summary, the main contributions of this paper are:

- A new approach to assessing accuracy, correctness potential, and consistent correctness 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 accuracy, correctness potential, and consistent correctness of 22 state-of-the-art instruction-tuned LLMs of varying sizes and a comprehensive analysis 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).

Contribution over our prior work. This work extends a paper [29] published at ICST 2025, the 18th IEEE International Conference on Software Testing, Verification and Validation. Our main additional contributions are as follows. First, we formalise the concept of a question neighbourhood (Section II). Second, we provide formal definitions for three evaluation metrics: Accuracy Score, Correctness-Potential Score, and Consistent-Correctness Score. The previous paper used a metric called CorrSc, which corresponds to the Accuracy Score in this submission, while the Correctness-Potential Score and the Consistent-Correctness Score are newly introduced here (Section II). Third, we expand our evaluation from five models in the prior work to 22 models, increasing the dataset of LLM responses from 300,000 to 1,320,000 entries (Section IV-A). Fourth, we apply rigorous statistical analysis using established methods, both for pairwise comparisons across the 22 models (Section IV-B) and for investigating the effect of reducing the temperature to zero. This analysis includes visualisations such as box plots and heatmaps to illustrate performance distributions across question neighbourhoods (Section IV-B). Fifth, we formalise four distinct categories of LLM performance (Section IV-C). Finally, we update the related work to include recent studies (Section VII).

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 for Python code generation. However, our approach is LLM- and programming language-agnostic, allowing future testing of other LLMs across various programming languages.

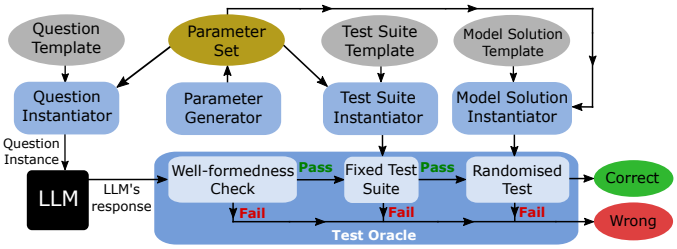

Fig. 2. Overview of our benchmarking approach.

Question Template and Instance. A *question template* T is a programming problem expressed in natural language, parameterised by one or more parameters $p_1, \ldots, p_n$ $(n > 0)$, together with a *constraint* C_T. Each parameter p_i has an associated value set V_i, which may have infinite size. The constraint C_T is a predicate on $V_1 \times \cdots \times V_n$ that can be used to restrict the template to certain combinations of parameter values (if any combination of parameter values is acceptable this can be captured by the trivial constraint $C_T(\ldots) = \mathit{true}$).

A *question instance* q is derived from a template T by substituting a sequence of parameter values $(v_1, \ldots, v_n)$ for placeholder(s) p_i in T, as long as $C_T(v_1, \ldots, v_n)$ holds. Hence, $q = T(p_i = v_i)$ where $1 \leq i \leq n$.

For example, the template in Figure 1a 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. Here, the associated constraint C requires that both p_1 and p_2 be greater than zero, and that p_2 be greater than or equal to p_1.

Definition 1 (Question Neighbourhood): Let T be a question template with parameters $p_1, \ldots, p_n$ drawn from value sets $V_1, \ldots, V_n$ $(n > 0)$ and associated constraint C_T. The *neighbourhood* of T, denoted $\mathcal{N}_T$, is defined as the set of all instances of T:

$$\mathcal{N}_T = \{T(p_i = v_i) \mid v_i \in V_i \ (1 \leq i \leq n) \ \wedge \ C_T(v_1, \ldots, v_n) \text{ holds}\}$$

According to Definition 1, two instantiated questions are in the same neighbourhood $\mathcal{N}_T$ if they are instances of the same template T with different parameter values.

An important property of a question neighbourhood, which distinguishes this study from previous works [13]–[23], is the *semantic closeness and non-equivalence* of the question instances within a neighbourhood. Specifically, all question instances $q \in \mathcal{N}_T$ are semantically close as they share the same structural template but differ in specific details (i.e., parameter values). Consequently, the instantiated questions are not semantically equivalent, as changing parameter values alters the particular task or problem instance. Moreover, since differences are only in parameter values at fixed positions, and the rest of the template remains identical, intuitively, the semantic difference between any two question instances in the neighbourhood is minimal compared to arbitrary questions outside the neighbourhood.

In practice, each question neighbourhood $\mathcal{N}_T$ is accompanied by a corresponding parameter set, consisting of parameter valuations that are meaningful and suitable 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. To make this concrete, the generator associated with each template samples parameter tuples that satisfy the constraint C_T, using restricted ranges to keep the parameter space finite and tractable (e.g., 1–3 digit integers for numeric parameters, single-character strings for string parameters). In a few templates, we additionally provided a small pool of manually selected values to guarantee coverage of edge cases. Sampling is random but constrained by C_T: for example, in the template of Figure 1a, pairs (p_1, p_2) are drawn from digit-based ranges and resampled if $0 < p_2 \leq p_1$ or if duplicates occur. Random seeds can be fixed to ensure reproducibility. This process yields finite, diverse parameter sets that respect each template's semantics and are suitable for automatic instantiation of question instances.

A question template can be instantiated automatically with a range of parameter values drawn from its parameter set, leading to a collection of question instances that can be presented to an LLM (see Figure 2).

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 once 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* [30]. 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 Accuracy, Correctness-Potential and Consistent-Correctness Scores. 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 three overall scores 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 2 (Accuracy Score (AS)): Let L be an LLM under evaluation. Let T be a question template with M associated parameter valuations (so that M distinct question instances are derived from T). Let $\mathcal{N}_T$ denote the set of these M instances—i.e., a finite subset of the corresponding full neighbourhood of T as defined in Definition 1. Suppose that the LLM is queried R times per question instance, and let $L_i^j(\mathcal{N}_T)$ denote the result returned by L the jth time it is queried with question instance i of T. Let $Oracle(L_i^j(\mathcal{N}_T))=1$ if this result is deemed correct according to the oracle template, and 0 otherwise. The *Accuracy Score, AS*, for question neighbourhood $\mathcal{N}_T$ for a given LLM L, $AS(\mathcal{N}_T, L)$, is then defined as follows:

$$AS(\mathcal{N}_T, L) = \frac{\sum_{i=1}^{M}\sum_{j=1}^{R} Oracle(L_i^j(\mathcal{N}_T))}{M \times R}$$

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. $AS(\mathcal{N}_T, L)$ yields a score in the range $[0, 1]$ for each question neighbourhood $\mathcal{N}_T$. When calculating *AS*, we do not evaluate $M \times R$ unique instances, but rather generate R solutions for each of the M question instances to account for stochasticity in the model. This design reflects the inherent non-deterministic nature of LLMs, allowing multiple attempts on the same question instance to assess the model's ability to generate correct responses.

Definition 3 (Correctness-Potential Score (CPS)): Continuing from Definition 2, let $\mathbf{1}(.)$ be the indicator function (1 if true, 0 if false). The *Correctness Potential Score, CPS*, for question neighbourhood $\mathcal{N}_T$ with respect to LLM L, $CPS(\mathcal{N}_T, L)$, is then defined as follows:

$$CPS(\mathcal{N}_T, L) = \frac{\sum_{i=1}^{M} \mathbf{1}\Big(\sum_{j=1}^{R} Oracle(L_i^j(\mathcal{N}_T)) \geq 1\Big)}{M}$$

This is the proportion of question instances for which the LLM produces at least one correct answer across *R* independent queries, relative to the total number of question instances. $CPS(\mathcal{N}_T, L)$ produces a score in the range $[0, 1]$ for each question neighbourhood $\mathcal{N}_T$.

Definition 4 (Consistent-Correctness Score (CCS)): Continuing from Definition 3, the *Consistent-Correctness Score, CCS*, for question neighbourhood $\mathcal{N}_T$ with respect to LLM L, $CCS(\mathcal{N}_T, L)$, is then defined as follows:

$$CCS(\mathcal{N}_T, L) = \frac{\sum_{i=1}^{M} \mathbf{1}\Big(\sum_{j=1}^{R} Oracle(L_i^j(\mathcal{N}_T)) = R\Big)}{M}$$

This is the proportion of question instances that the LLM can consistenly answer correctly across *all R* independent queries, relative to the total number of question instances. $CCS(\mathcal{N}_T, L)$ thus yields a score in the range $[0, 1]$ for each question neighbourhood $\mathcal{N}_T$.

Although it is straightforward to interpret $1 - CPS$ as the proportion of question instances for which an LLM fails to produce a correct output across all R queries—i.e., a measure of consistent incorrectness—this study deliberately focuses on consistent correctness. Our evaluation framework is motivated by the goal of assessing the safety-related properties of LLMs in code generation, where the primary concern is whether a model can reliably and repeatedly produce correct outputs. While consistent incorrectness may reveal systematic failure patterns, it does not directly inform safety. In safety-critical contexts, a model that is occasionally correct presents a higher utility and lower risk profile than one that is consistently incorrect. Accordingly, our analysis prioritises behaviours aligned with trustworthy and reliable performance, which are best captured through consistent correctness rather than its inverse.

Taken together, *AS*, *CPS*, and *CCS* capture distinct yet interrelated aspects of LLM performance over a question neighbourhood and enable a more comprehensive understanding of model behaviour in code generation tasks. The Accuracy Score measures the overall proportion of correct generations across all question instances and all repeated queries. This gives a general sense of how often the model succeeds across the entire neighbourhood but does not reveal whether success is widespread or concentrated in a few instances. The Correctness-Potential Score complements this by measuring the proportion of question instances for which the model generates at least one correct output across multiple queries. This score reflects whether the model is capable of solving a given task instance at all—even if inconsistently—offering insight into the model's latent capacity across varied parameter values. Finally, the Consistent-Correctness Score captures the proportion of instances in the neighbourhood for which the model consistently produces correct outputs in all repeated attempts. This reflects how reliably the model maintains correctness even under stochastic generation settings, where different outputs may be sampled across runs.

Using these scores in combination allows us to diagnose nuanced behavioural patterns that would be missed by any single aggregate measure. For example, a model might exhibit a high *CPS*—indicating it is capable of solving most instances—but a low *CCS*, revealing that it rarely solves those instances reliably. This discrepancy may point to instability in generation or a sensitivity to sampling variability. Conversely, a model might achieve high *AS* by consistently solving a small subset of question instances, while failing on the rest—something only visible when considering *accuracy* alongside *correctness potential*. In another scenario, low scores across all three metrics may indicate a structural failure to generalise across the space spanned by the neighbourhood. Together, the three metrics help disentangle these cases, revealing whether performance is broad but unreliable, narrow but consistent, or absent altogether. This multi-dimensional view is crucial for identifying discontinuities in performance, pinpointing specific weaknesses in generalisation, and understanding the reliability of an LLM's behaviour across structured problem sets.

This multi-dimensional analysis sets our approach apart from standard evaluation metrics such as $pass@k$ [3], which simply measures whether a correct solution appears in the top-k outputs for a single input. While $pass@k$ is useful for assessing best-case success on isolated prompts, it does not provide insight into how performance varies across a structured space of related tasks, nor does it capture consistency under repeated sampling. In contrast, our scores provide instance-level and neighbourhood-level resolution, exposing both what a model can do and whether it does so reliably—critical considerations

for deploying LLMs in code generation settings that demand both correctness and reliability.

III. The Turbulence Benchmark

Based on the approach described in Section II we have created a novel benchmark, Turbulence, for evaluating the accuracy, correctness potential, and consistent correctness of instruction-tuned LLMs 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, thereby ensuring comprehensive and balanced coverage of key topics. No specific 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 [28].

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 stated in the question text or implicitly required by the solution logic. 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, subgroup counts within each problem group may exceed the group total, since some questions belong to multiple subgroups.

Turbulence comprises 60 question templates, each featuring at least one parameter, where each parameter is either a numerical value or a string. Of these, 58 templates include numerical parameters and 5 include string parameters, with 3 templates incorporating both types. 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 $p_2 < p_1$. When such hand-chosen parameter values were used, they are documented in a file called `manually_chosen_params.txt`, provided alongside the corresponding question template in the source code folder of our replication package [31], [32], ensuring transparency and reproducibility.

In designing the question templates, we deliberately chose not to include `import` statements in the problem prompts. Providing imports risks biasing the model toward particular solution strategies, since the presence of a library import can serve as a strong cue for its use. By omitting such scaffolding, we require models to independently determine whether external functionality is necessary and to produce self-contained solutions within the standard Python environment. Although this increases the likelihood of execution-time failures (e.g., when a model relies on a library it has not imported), these outcomes are themselves informative, as they expose limitations in the model's ability to generate executable code without external guidance. Our design choice therefore supports the benchmark's primary aim: to evaluate correctness and robustness under minimal prompting, such that reported performance reflects the model's intrinsic code generation capability rather than reliance on provided context.

To avoid problems of bias occurring due to LLMs having been exposed to questions during the training [28], 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 memorised training data, 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. During the construction of Turbulence, we used only a small set of trivial questions for preliminary LLM evaluation, to avoid the risk of the models learning from our interactions.

To ensure the clarity of question templates and the correctness of test oracles, we conducted a validation exercise with two experienced Python programmers. Each participant independently solved an instance of every question template, after which their solutions were cross-checked against our test oracles. Discrepancies between their solutions and the oracle were carefully analysed to determine whether the issue arose from (i) ambiguity or imprecision in the natural language specification of the template, or (ii) errors in the construction of the oracle itself. Based on this process, we identified 20 cases requiring intervention: 10 question templates were revised to improve wording and eliminate potential ambiguity and 10 test oracles were corrected to address implementation bugs or coverage gaps. This validation step provided assurance that the curated benchmark is both unambiguous and aligned with its test oracles, thereby reducing the risk of unfairly penalising correct model outputs.

Creating our own questions has its pros and cons. As argued above, using previously-unseen questions minimises problems of training-related bias [28], 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

TABLE I
CLASSIFICATION OF THE TURBULENCE QUESTION TEMPLATES INTO PROBLEM GROUPS AND SUBGROUPS.

Problem Group	Problem Subgroup	Number of Question Templates
List Manipulation	Total	44
	Slicing	27
	Indexing	26
	Filtering	24
	Element-based Operations	26
	Summation	6
	Sorting/Order-based Operations	13
	Element Insertion	2
	Count elements	1
	Circular Lists	1
String Manipulation	Total	16
	Character Insertion	2
	Indexing	7
	Character Removal	3
	Substring/Character Extraction	12
	Palindrome Operations	3
	Anagram Detection	2
	Sorting	3
Set Manipulation	Total	9
	Add Elements	6
	Subset/Superset Operation	1
	Counting Subsets	1
	Intersection	1
	Union	1
Searching	Total	46
	Linear Search	34
	Index-based Search	11
	String Search	16
Copying	Total	20
	Shallow Copy	2
	Copy Sublist	18
Mathematical Problems	Total	22
	Arithmetic Operations	9
	Factorial Calculations	1
	Prime Checking	5
	Composite Checking	1
	Factorisation	4
	Special Sequences	6
	Combinatorial Problems	2

"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. [33], 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 findings 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). While Turbulence is instantiated for Python in this study, the underlying methodology is inherently language-agnostic. Extending the framework to other programming languages would entails engineering effort in two principal respects. First, some question templates rely on Python-specific constructs—such as list comprehensions, tuples, or NumPy features—that do not always have direct analogues in other languages. These question templates would need to be reformulated to use idiomatic equivalents in the new target language (e.g. a question template involving a Python list comprehension could be re-expressed to refer to a loop when targeting C++). Resources such as the MultiPL-E dataset [34], which provide systematic cross-language mappings, could support such adaptations where straightforward correspondences exist. Second, certain question templates are intrinsically tied to the semantics of Python; these kinds of templates could be swapped out for suitable templates designed to exercise specific features of interest in the new target language. With respect to our empirical results, however, we do not claim that the specific findings generalise directly to other LLMs or languages. Rather, we expect that applying the methodology across contexts will reveal different classes of deficiencies, much as distinct software systems exhibit unique faults when subjected to the same testing technique. Overall, our results show that the proposed approach effectively reveals key strengths and limitations of code generation.

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 requires modest prompt engineering [35] to enhance the chances of obtaining source code from an LLM. In initial experiments with the models in Section IV, we found that adding a simple prefix requesting Python code within triple backticks was effective, allowing the code to be extracted from between the backticks.

Availability. The Turbulence source code, all question and oracle templates, and the complete results are available at https://github.com/ShahinHonarvar/Turbulence-Benchmark-v2.

IV. Experimental Evaluation

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

A. Experimental Setup

We gathered results by running Turbulence against 22 instruction-tuned LLMs, including both closed- and open-source models of varying sizes. Our selection spans a diverse range of development teams, with each model chosen to represent either the flagship or largest version within its series—highlighting the full capabilities of its model family. The selected models are either explicitly trained for code generation or are general-purpose LLMs with demonstrated proficiency in code generation and reasoning tasks.

Given the rapid pace of progress in LLM development across both industry and academia, new models are introduced frequently. The models evaluated in this study represent the state-of-the-art options available during the period of our experiments. We therefore emphasise that the significance of our work lies not in the specific set of models evaluated, but in the novelty and flexibility of our approach, which enables the systematic assessment of any code-generating LLM. We demonstrate the effectiveness of this approach in uncovering issues related to accuracy, correctness potential, and consistent correctness in such models. As the field evolves and newer models are released, our evaluation framework remains applicable and valuable for analysing future generations of LLMs under the same rigorous criteria.

The instruction-tuned LLMs evaluated in this study are as follows: *Command*, *Command R+*, *GPT-3.5-turbo*, *GPT-4*, *GPT-4o*, *Claude 3.5 Haiku*, *Claude 3.5 Sonnet*, *Gemini 1.5 Pro*, *Gemma-2-27B*, *CodeGemma-7B*, *DeepSeek-Coder-V2*, *Mistral Large 2*, *Codestral*, *Dolphin 2.9.2 Mixtral 8x22b*, *Llama-3.1-70B*, *Llama-3.1-405B*, *Phi-3-medium*, *Qwen2.5-72B*, *Qwen2.5-Coder-7B*, *Qwen2.5-Coder-32B*, *DBRX*, and *StarCoder2-15B*.

OpenAI [36] introduced the *GPT-3.5-turbo* [37], *GPT-4* [37], and *GPT-4o* [37] models. The *GPT-3.5-turbo* model is optimised for chat, handling both natural language and code generation. The multimodal *GPT-4* model improves problem-solving with broader knowledge and advanced reasoning. *GPT-4o* ("o" for omni), OpenAI's flagship model, excels in English text and code. We accessed all three models via OpenAI's commercial API.

Command [38], Cohere's [39] default generation model, produces text based on user instructions. *Command R+* [38], an advanced instruction-following model, offers superior performance in language tasks, maths, code, and reasoning. Both models were accessed via Cohere's commercial API.

The Claude 3.5 family [40], developed by Anthropic [41], offers enhanced performance, speed, and versatility. Key models include *Claude 3.5 Sonnet* [42], optimised for high-quality content and code generation, and *Claude 3.5 Haiku* [43], designed for fast, accurate code suggestions. We accessed these models through Anthropic's commercial API.

Gemini 1.5 Pro [44], by Google DeepMind [45], is a multimodal LLM optimised for reasoning tasks. We accessed it via Gemini's commercial API. *Gemma-2-27B*, with 27B-parameter is the largest instruction-tuned model in the open-source Gemma 2 series [46]. We accessed it via the deepinfra [47] commercial API. CodeGemma [48] is an open model for code, with the *CodeGemma-7B* variant excelling in mathematics, coding, and language understanding. We accessed it via Google Cloud Vertex AI's API [49].

DeepSeek-Coder-V2 [50], developed by DeepSeek AI [51], is an open-source code model with 236B parameters in total, 21B of which are active during inference. It is pre-trained for coding, mathematical reasoning, and general language tasks. We accessed it via DeepSeek's commercial API.

Mistral Large 2 [52], a 123B-parameter open-source model, is Mistral-AI's [53] latest flagship, excelling in code generation, mathematics, reasoning, and precise instruction following. *Codestral* [54], also introduced by Mistral-AI [53], a 22B-parameter model trained for code, is available as an open-weight model for research and testing. We accessed both via Mistral-AI's commercial API.

Dolphin 2.9.2 Mixtral 8x22b [55], developed by Cognitive Computations [55], is a fine-tuned version of Mixtral-8x7B [56], trained on diverse data, including coding. We accessed it via OpenRouter's [57] commercial API.

Phi-3-medium [58], a 14B-parameter open-source model by Microsoft [59], is trained on diverse datasets to improve language understanding, reasoning, maths, and coding. We accessed it through Microsoft Azure's serverless API [60].

Llama-3.1-70B [61] and *Llama-3.1-405B* [61], with 70 and 405 billion parameters respectively, are part of Meta's Llama 3.1 [61] open-source collection. Both are pre-trained and instruction-tuned for coding and reasoning, with *Llama-3.1-405B* the largest publicly available LLM [61]. We accessed them via Microsoft Azure's API [60].

Qwen2.5-72B, the largest instruct model in the Qwen 2.5 series [62], and *Qwen2.5-Coder-32B* and *Qwen2.5-Coder-7B*, the largest and a medium-sized model in the Qwen 2.5 Coder series [63], were developed by Alibaba Cloud's Qwen team [62]. We accessed *Qwen2.5-72B* and *Qwen2.5-Coder-32B* via DeepInfra's API [47] and *Qwen2.5-Coder-7B* via Hugging Face's endpoint [64].

DBRX [65], an advanced open-source model by Databricks [66], has 132B parameters, with 36B active during inference for efficiency. Trained on 12T tokens of text and code, it excels in programming and mathematical reasoning. We accessed it via OpenRouter's commercial API [57].

The StarCoder2 series [67], by the BigCode project [68], is an open-source family of LLMs for code generation and comprehension. We accessed its largest instruction-tuned model, *StarCoder2-15B*, via Hugging Face's endpoint [64].

This selection allows us to evaluate both proprietary and open models of varying sizes. It is worth noting that, although OpenAI's o1 model [69] was released at the time of our experiments, we were unable to include it due to API access restrictions (limited to Tier 5, while our account was Tier 3). Moreover, even if access had been available, the cost of using the o1 model via its API would have been prohibitively high at the time. Further details are provided in Section VI.

Every LLM has a user-determined *temperature* parameter that controls output randomness. Lower temperatures reduce randomness, improving quality but decreasing diversity [70], [71], 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 0. A temperature of 0 was chosen to evaluate the LLMs' performance 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 selected them to balance output diversity and coherence for optimal performance. Since these default temperatures reflect the intended behaviour envisioned by the developers, we used them to assess the LLMs' performance in their standard operational settings. Our goal was to analyse the shift from non-deterministic behaviour at default temperatures to maximum determinism at a temperature of 0, offering clear insights into how determinism affects LLM performance without the complexity of varying randomness. Broader temperature ranges were excluded as outside the study's scope.

Due to the stochastic nature of LLMs, repeat runs of experiments are necessary. At the same time, access to LLMs via commercial APIs is costly, with variable query times. We ran the full benchmark five times ($R = 5$ in Definition 2, Definition 3, and Definition 4) for each combination of LLM and temperature setting.

While a temperature of 0 should theoretically render the LLMs deterministic, one might ask why each LLM was run through the full benchmark five times at this temperature. Our initial mock tests revealed that LLMs occasionally produced varying answers even at a temperature of 0. This non-deterministic behaviour may be due to several factors, including non-deterministic GPU operations, memory access patterns, and numerical precision [72] and the inherent randomness from sampling, even at a temperature of 0 [73].

Our results are thus based on a comprehensive set of 1,320,000 LLM responses (i.e. number of models $\times$ number of temperature settings per model $\times$ number of prompts per each model's temperature setting $\times$ number of repeat runs $= 22 \times 2 \times 6000 \times 5 = 1320000$). For a consistent comparison of experimental results, we used the same random seed when generating parameters for each question template.

In the rest of this paper, *LLM configuration* refers to an LLM combined with a specific temperature setting and $t = 0$ and $t{=}D$ denote the configurations with temperature settings of 0 and the default, respectively.

B. Results Based on AS, CPS, and CCS

In this section, we address **RQ1** and **RQ2**. As outlined in Section II, each parameter p_i in a question template T has an associated value set V_i, which may be finite or infinite depending on the nature of the constraint C_T. In Turbulence, for the majority of the 60 question templates, the value sets V_i are theoretically infinite. In the context of an infinite population of all possible parameter values, it is not feasible to sample values uniformly such that every value has an equal probability of selection. To address this, we restricted the value sets V_i for each parameter in a question template, rendering them finite. For example, in cases where the parameter values were natural numbers, we defined V_i to include only 1-digit, 2-digit, and 3-digit numbers. Similarly, for questions involving string parameters, we limited V_i to single-character strings. This restriction affected only 5 of the 60 curated question templates, and none of these involved tasks (such as sorting) where multi-character strings would be essential. In these cases, single characters were sufficient to preserve the intended semantics of the task while also keeping the parameter space tractable and simplifying oracle design.

By limiting the sets V_i in this manner, we ensured a finite population of parameter values, which also facilitated the management of computational costs associated with our evaluation process. Consequently, all findings, statistical estimates, and inferences presented herein apply strictly to the chosen sets V_i and should not be extrapolated to parameter values outside these sets without additional justification or analysis. However, the approach presented in this study is not dependent on the specific selection of V_i. Using the principles of our method, it is possible to choose alternative regions of the parameter value space for each question, evaluate the performance of LLMs, and present the corresponding statistics results.

To enable a robust and multi-dimensional analysis of LLM behaviour across code generation tasks, we employ both box plots and heatmaps for each of our evaluation scores: *AS*, *CPS*, and *CCS*. These visualisations are complementary, each offering distinct analytical strengths.

We begin by presenting our results using box plots to provide a distribution-level view of how each LLM configuration performs across the 60 question neighbourhoods. Box plots enable us to assess central tendency (median), variability (interquartile range), and the presence of outliers—offering insight into how different configurations behave across neighbourhoods. For instance, two configurations may have similar average scores yet differ markedly in score dispersion: one may yield tightly clustered results, while another exhibits high variability. Box plots thus support an accessible, distributional comparison across models. To extend the analysis beyond visual interpretation, we use heatmaps to conduct and visualise a formal inter-model comparison. These allow us to assess whether one configuration significantly and meaningfully outperforms another, addressing the limitations of distributional overlap that box plots alone cannot resolve. By systematically comparing all configuration pairs, the heatmaps enable us to investigate the effect of temperature reduction on LLM code generation—revealing whether setting the temperature to zero leads to statistically and practically meaningful differences in accuracy, correctness potential, or consistent correctness.

AS, CPS, and CCS Distributional Analysis via Box Plots: Figure 3 comprises three stacked box plots, each corresponding to a distinct evaluation score—*AS*, *CPS*, and *CCS*—used to assess LLM performance on code generation tasks. Each box represents the distribution of a given score across 60 question neighbourhoods for a specific LLM configuration. There are 44 configurations in total (22 models, each evaluated at two temperature settings). A lighter blue represents the

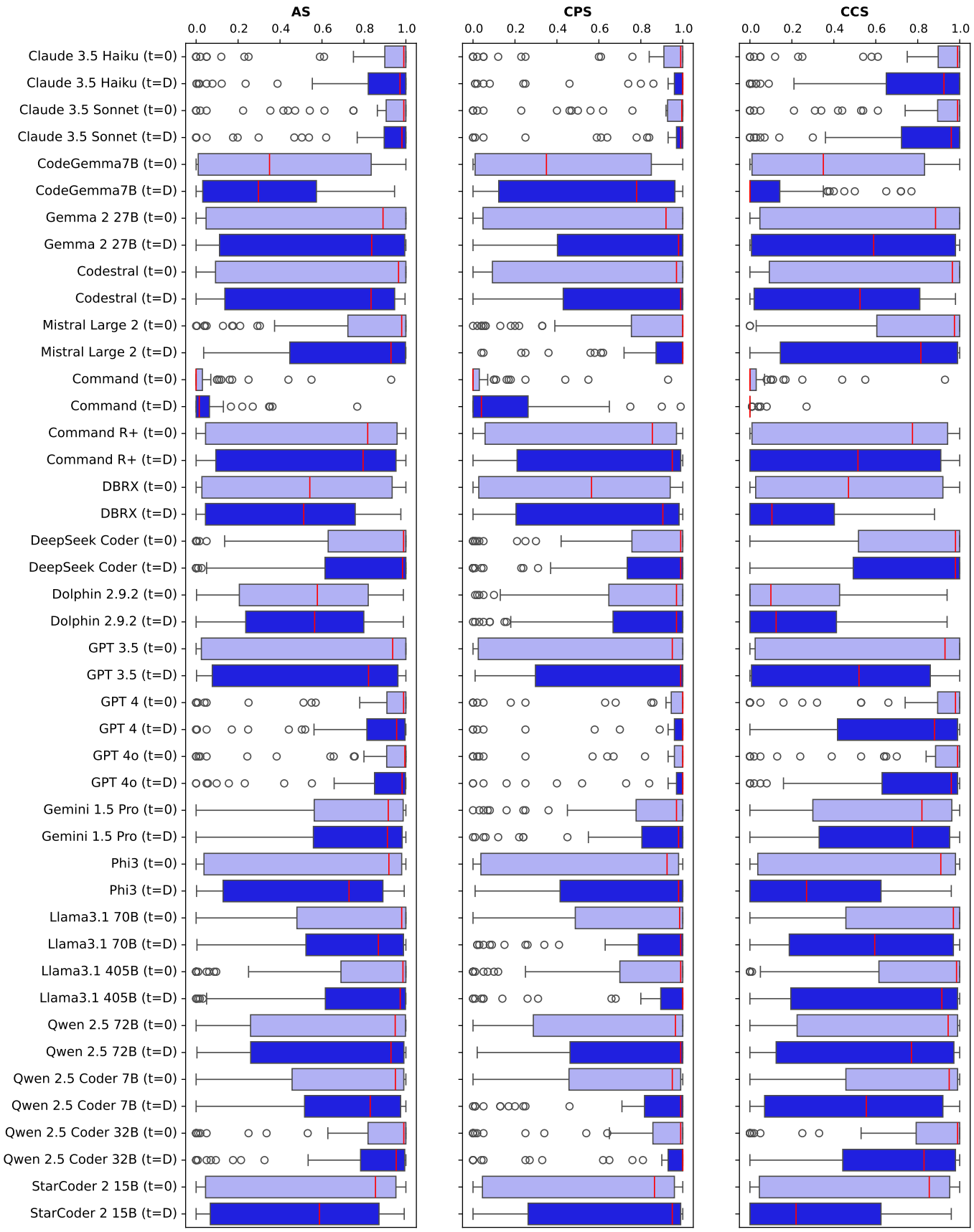

Fig. 3. Box plots of *AS*, *CPS*, and *CCS* across LLM configurations.

lower temperature configuration ($t = 0$) for each model. For clarity, the median line in each box plot is rendered in red.

AS Distribution. Recall that an *AS* of 1.0 means that, for a particular question neighbourhood, the LLM configuration is able to generate correct code for all instantiations and across all 5 generations. Conversely, an *AS* of 0.5 indicates that, on average, only half of the generated solutions across all instantiations are correct. An *AS* close to 0.0 therefore reflects systematic failure on the problem, whereas intermediate values (e.g., between 0.2 and 0.8) capture varying degrees of partial success, either due to stochastic behaviour across generations or differential performance across parameter instantiations. The left panel of Figure 3 presents the results for the *AS*. The distribution of box plots reveals that several LLM configurations exhibit skewed accuracy distributions, reflecting varying levels of accuracy across the LLM configurations.

LLM configurations such as *Claude 3.5 Haiku*, *Claude 3.5 Sonnet*, *GPT-4*, *GPT-4o*, and *Qwen2.5-Coder-32B*, each at both $t = 0$ and $t = D$, demonstrate high accuracy, with median *AS* scores exceeding 0.95. These models also exhibit narrow interquartile ranges (IQRs) between 0.091 and 0.21, along with short whiskers—indicating reliable accuracy across most neighbourhoods. Nonetheless, each configuration shows a number of outliers (9-14 of the 60 question neighbourhoods), suggesting that even the top-performing models occasionally struggle with specific question neighbourhoods.

Another group of LLM configurations, including *DeepSeek-Coder-V2* and *Llama-3.1-405B* (at both temperatures), *Mistral Large 2*, *Llama-3.1-70B*, and *Qwen2.5-Coder-7B* (at $t = 0$), also achieve high median accuracy scores but exhibit moderately wider IQRs (ranging from 0.27 to 0.55). This suggests increased variability in accuracy across different neighbourhoods compared to the previous group. Further down the accuracy spectrum, models such as *Gemini 1.5 Pro*, *Qwen2.5-72B*, *Qwen2.5-Coder-7B*, *Codestral*, *Phi-3-medium*, *GPT-3.5-turbo*, *Gemma-2-27B*, and *StarCoder2-15B*, across both temperature settings, display lower median *AS* values and substantially wider IQRs. These characteristics point to a greater variability in accuracy. Notably, *Phi-3-medium* at $t = 0$ exhibits a relatively high median (above 0.9) but also one of the widest IQRs, indicating considerable variation in accuracy.

The remaining LLM configurations exhibit low median *AS* scores coupled with wide IQRs, indicating poor and unstable accuracy. Among them, *Command* stands out as the worst-performing model, with near-zero median *AS* values (0.0 at $t = 0$ and 0.015 at $t = D$) despite relatively narrow IQRs. This reflects a consistent inability to generate correct solutions across most neighbourhoods.

CPS Distribution. Recall that a *CPS* of 1.0 for a particular question neighbourhood indicates that the model solves every instance at least once, even if it does not do so consistently across repeated generations. A *CPS* of 0.5 means that only half of the instances are solved at least once, while the remainder are never solved correctly. Values near 0.0 suggest failure on almost all instances, whereas intermediate values reflect uneven performance, with correctness achieved only sporadically across different instances.

The centre panel of Figure 3 illustrates the distribution of *CPS* values across LLM configurations. A group of these configurations achieves near-perfect *CPS* scores, with medians between 0.99 and 1.0 and exceptionally narrow interquartile ranges (IQRs) between 0.03 and 0.13. This group comprises *GPT-4o*, *GPT-4*, *Claude 3.5 Haiku*, and *Claude 3.5 Sonnet*, all evaluated under both $t=0$ and $t=D$ settings, as well as *Mistral Large 2*, *Llama-3.1-405B*, and *Qwen2.5-Coder-32B* at $t=D$. These configurations demonstrate strong potential to produce at least one correct solution across $R = 5$ trials in the majority of question neighbourhoods. However, each also exhibits between 8 and 13 outliers out of 60 neighbourhoods, indicating that despite their generally high correctness potential, they still fail consistently on a subset of question instances.

The next tier comprises LLM configurations such as *DeepSeek-Coder-V2*, *Llama-3.1-70B*, *Gemini 1.5 Pro*, and *Dolphin 2.9.2 Mixtral 8x22b*, evaluated under both $t=0$ and $t = D$ settings; *Qwen2.5-Coder-32B*, *Mistral Large 2*, and *Llama-3.1-405B*, each at $t = 0$; and *Qwen2.5-Coder-7B* and *Qwen2.5-72B*, each at $t=D$. These configurations attain high median *CPS* values but are associated with moderately wider interquartile ranges (ranging from 0.14 to 0.54), indicating greater variability in their ability to produce at least one correct solution per question instance across neighbourhoods. The other LLM configurations show reduced correctness potential and substantial variability. These models show no statistical outliers, suggesting that the variation is broadly distributed rather than driven by extreme cases. *Command* is a notable exception, with markedly lower scores and observable outliers.

CCS Distribution. Recall that a *CCS* of 1.0 for a given question neighbourhood indicates that the model answers every instance correctly in all 5 generations, demonstrating full consistency in its outputs. A *CCS* of 0.5 means that only half of the instances are solved correctly in every generation, while the remainder include at least one failure. Values close to 0.0 suggest that the model rarely achieves consistent correctness, even when it can occasionally produce correct solutions. Intermediate values reflect partial consistency, where some instances are always solved correctly while others exhibit variability across generations. The right panel of Figure 3 presents the distribution of *CCS* values across LLM configurations. Among the highest-scoring configurations are *Claude 3.5 Haiku*, *Claude 3.5 Sonnet*, *GPT-4*, and *GPT-4o*, all evaluated at $t=0$. These models achieve *CCS* median values between 0.98 and 0.99, with narrow interquartile ranges ranging from 0.10 to 0.12. Each configuration also shows 12–14 outliers among the 60 question neighbourhoods, indicating that although correctness is generally maintained, the models still face challenges in a non-negligible subset of neighbourhoods.

A second group of LLM configurations—including *Qwen2.5-Coder-32B* (at $t=0$), *Claude 3.5 Sonnet* and *GPT-4o* (each at $t = D$), *Llama-3.1-405B* and *Mistral Large 2* (each at $t = 0$), and *Claude 3.5 Haiku* (at $t=D$)—achieves relatively high *CCS* scores, though with moderately wider interquartile ranges ranging from 0.21 to 0.40. These models generally sustain correctness across repeated trials in many neighbourhoods, but the broader IQRs suggest a more variable

ability to maintain correctness compared to the first group.

A third group, comprising models such as *DeepSeek-Coder-V2* (at both temperatures), *Llama-3.1-70B*, *Qwen2.5-Coder-7B* (each at $t = 0$), and *GPT-4* (at $t = D$), achieves moderately high *CCS* values but exhibits substantial variability, with IQRs extending up to 0.57. These configurations do not show statistical outliers, which suggests that their lower scores result from widespread variation rather than isolated failures. While these models may be capable of producing correct solutions in some trials, they show reduced ability to maintain correctness across all five trials in a number of neighbourhoods. The remaining LLM configurations are associated with substantially lower *CCS* values and wide interquartile ranges, indicating a reduced capacity to maintain correctness across all trials. Most configurations in this group do not produce statistical outliers, suggesting that the observed variability reflects broadly distributed deviations rather than isolated anomalies. Among the lowest-scoring configurations are *Command* (at both temperatures), *CodeGemma-7B*, *DBRX*, and *StarCoder2-15B* (each particularly at $t{=}D$), and *Dolphin 2.9.2 Mixtral 8x22b* (at both temperatures). Notably, *Command* and *CodeGemma-7B* (at $t{=}D$) also produce outliers. These configurations collectively exhibit persistent difficulty in maintaining correct outputs across repeated attempts.

Overall, the patterns in Figure 3 provide an accessible distribution-level view of LLM behaviour across neighbourhoods for the three evaluation scores. However, these descriptive patterns should be interpreted with caution and complemented by the pairwise statistical analysis that follows, which rigorously tests whether differences are statistically significant and practically meaningful.

Pairwise Comparative Analysis of AS, CPS, and CCS via Heatmaps: We present a separate heatmap for each score—*AS*, *CPS*, and *CCS*—to enable a statistically grounded, pairwise comparison of all LLM configurations. While box plots provide distributional insights, they do not establish whether observed differences are statistically significant or practically meaningful. The heatmaps we present here address this limitation by combining non-parametric significance testing with effect size estimation, allowing for a rigorous assessment of both inter-model differences and the impact of temperature reduction within each model. This level of analysis is crucial for moving beyond visual inspection and drawing robust conclusions about comparative model behaviour.

Recall that each LLM configuration was evaluated on three metrics—*AS*, *CPS*, and *CCS*—across all 60 question neighbourhoods, resulting in three distributions of scores per configuration, each of size 60. To conduct the pairwise comparative analysis, we considered all $\binom{44}{2} = 946$ possible pairs among the 44 LLM configurations. This analysis was performed separately for each of the three evaluation metrics. Prior to selecting an appropriate statistical test, we assessed whether the distributions of each score followed a normal distribution. This evaluation was conducted using the Shapiro-Wilk test [74], which is known to be the most powerful normality test [75]. Our analysis indicated that the distributions did not conform to normality. Consequently, we employed the Mann–Whitney U test [76] to determine statistical significance between each pair of distributions, as this test is non-parametric and does not assume normality. We set the significance threshold at $p < 0.05$, corresponding to a 95% confidence level. Additionally, we used Cliff's δ [77] as a non-parametric measure of effect size, which remains valid regardless of the underlying data distribution.

The inclusion of effect size alongside statistical significance is crucial, as a statistically significant result ($p < 0.05$) merely indicates that an observed difference is unlikely to have occurred by chance but does not convey the magnitude or practical relevance of the difference. Effect size, in contrast, quantifies the strength of the observed effect, offering insights into its real-world importance independent of sample size.

The heatmaps in Figure 4, Figure 5, and Figure 6 present a comparative analysis of all LLM configurations in terms of accuracy, correctness potential, and consistent correctness, respectively. For conciseness, only the *Dolphin 2.9.2 Mixtral 8x22b* model is labelled as Dolphin 2.9.2 in all figures. In each figure, each cell represents the effect size of the difference between a given pair of LLM configurations for the corresponding metric. A star in the bottom-left corner of a cell indicates that the observed difference is statistically significant, meaning it is unlikely to have arisen by chance.

The colour scheme of the heatmap in each figure visually conveys both the direction and magnitude of the differences. A positive value, represented by a red-shaded cell with intensity varying according to magnitude, indicates that the LLM configuration on the vertical axis of the heatmap has a higher corresponding score value (i.e., *AS*, *CPS*, or *CCS*) than the model on the horizontal axis. Conversely, a negative value, shown as a blue-shaded cell, signifies that the LLM configuration on the horizontal axis outperforms the one on the vertical axis with respect to that metric. Thus, a negative effect size does not imply no effect but a reversal in direction, with the horizontal-axis group showing larger values than the vertical-axis group. A cell value of zero denotes no difference between the two configurations.

For effect size interpretation, the magnitude, determined by its absolute value, is critical, as it reflects the strength of the effect, irrespective of direction. This is done based on established thresholds in the literature. We adopt the classification proposed by Cliff's study [77], where an effect is considered negligible if $|\delta| < 0.147$, small if $0.147 \leq |\delta| < 0.33$, medium if $0.33 \leq |\delta| < 0.474$, and large if $|\delta| \geq 0.474$. The significance and practical implications of each cell value in Figure 4, Figure 5 and Figure 6 depend on both its magnitude and statistical significance ($p < 0.05$): if an effect size is small or negligible but statistically significant, the difference between the corresponding LLM configurations is unlikely due to chance but remains too minor to be practically meaningful; if an effect size is small or negligible and not statistically significant, there is insufficient evidence of a meaningful effect, with differences likely arising from random variation; and if an effect size is medium or large and statistically significant, the difference between groups is both real (unlikely due to chance) and practically meaningful.

To illustrate how to interpret the data presented in Figure 4, Figure 5, and Figure 6, we provide a representative example

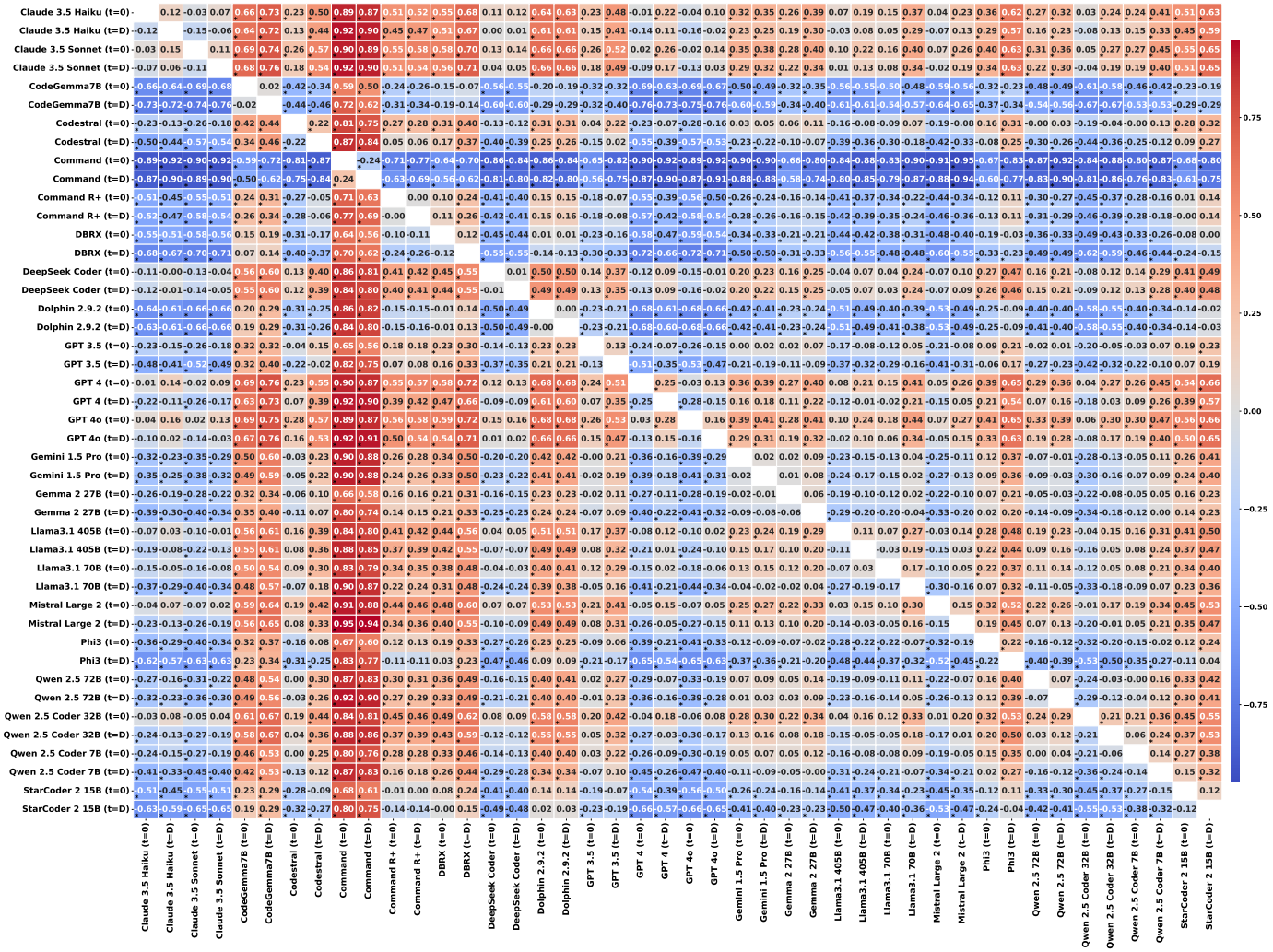

Fig. 4. Comparison of LLM configurations based on pairwise *AS* effect size and statistical significance. Note that the heatmap is symmetric.

drawn from Figure 4. The interpretation outlined here generalises to all three figures with respect to their corresponding evaluation metric. In Figure 4, the cell corresponding to *Claude 3.5 Haiku* ($t{=}D$) on the vertical axis (the second LLM configuration from the top) and *StarCoder2-15B* ($t{=}D$) on the horizontal axis (the rightmost LLM configuration) contains the value 0.59 accompanied by a star. The presence of the star indicates that the difference between the two LLM configurations is statistically significant. Given that the Cliff's δ value is $|\delta| = 0.59 > 0.474$, it qualifies as a large effect size, indicating that the difference is practically meaningful and that *Claude 3.5 Haiku* ($t{=}D$) achieved substantially higher accuracy than *StarCoder2-15B* ($t{=}D$) in our evaluation.

To identify the LLM configuration(s) that achieve the highest levels of accuracy, correctness potential, and consistent correctness—as measured by *AS*, *CPS*, and *CCS*, respectively—we refer to Figure 4, Figure 5, and Figure 6. Specifically, we look for configurations that consistently exhibit the largest effect size values when compared to others. When a group of LLM configurations each shows large values against the rest but exhibits no statistically significant differences and only negligible effect sizes in their pairwise comparisons, they may be considered to perform similarly with respect to the corresponding metric. Based on this, the lists below present the LLM configurations that show no statistically significant difference and negligible effect size in pairwise comparisons, indicating similar performance in that metric:

AS: *Claude 3.5 Haiku* ($t = 0$), *Claude 3.5 Sonnet* (at both temperatures), *Mistral Large 2* ($t{=}0$), *GPT-4* ($t{=}0$), *GPT-4o* ($t{=}0$), *Llama-3.1-405B* ($t{=}0$), *Qwen2.5-Coder-32B* ($t{=}0$).

CPS: *Claude 3.5 Haiku* ($t = D$), *Claude 3.5 Sonnet* (at both temperatures), *Mistral Large 2* ($t = D$), *GPT-4* (at both temperatures), *GPT-4o* (at both temperatures), *Qwen2.5-Coder-32B* ($t{=}D$).

CCS: *Claude 3.5 Haiku* ($t = 0$), *Claude 3.5 Sonnet* ($t = 0$), *Mistral Large 2* ($t{=}0$), *GPT-4* ($t{=}0$), *GPT-4o* ($t{=}0$), *Llama-3.1-405B* ($t{=}0$), *Qwen2.5-Coder-32B* ($t{=}0$).

We observe that, with the exception of *Claude 3.5 Sonnet* ($t{=}D$), the *AS* and *CCS* lists include an identical set of LLM configurations, all operating under $t = 0$. This alignment can be partially attributed to how the two metrics relate to repeated correctness: while *AS* measures the average correctness

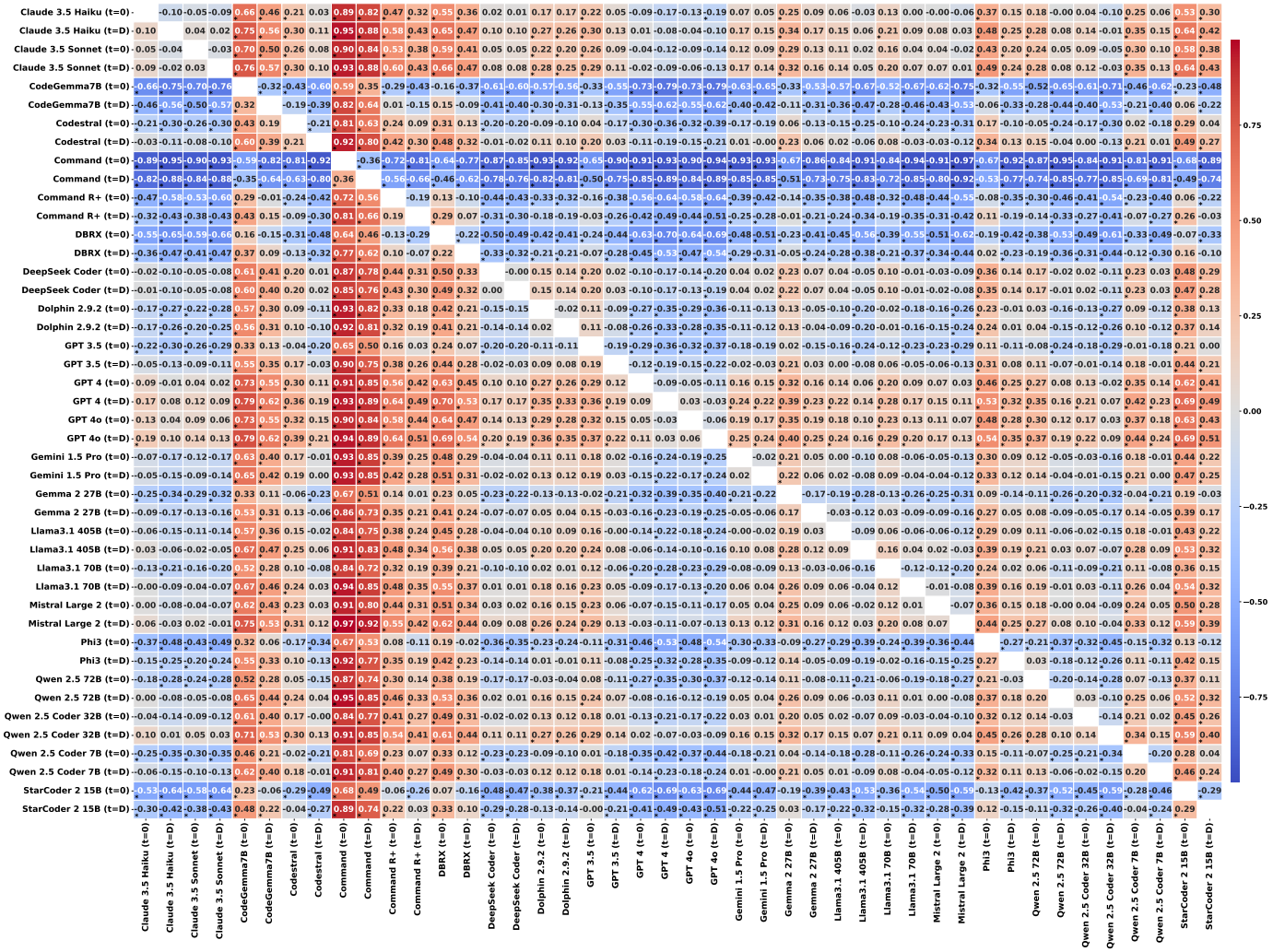

Fig. 5. Comparison of LLM configurations based on pairwise *CPS* effect size and statistical significance. Note that the heatmap is symmetric.

across all generations, *CCS* assigns credit only when all five responses for a given question instance are correct. Although setting the temperature to 0 does not make generation fully deterministic [73], it provides the most deterministic decoding context available. In such settings, models that produce a correct output once often replicate that output across repeated trials. For the specific LLM configurations identified in the top *AS* and *CCS* groups, this behaviour appears to hold—they tend to generate outputs that are both frequent and consistent, resulting in strong performance on both metrics. However, we emphasise that this alignment is not generalisable to all LLMs, as consistency under $t=0$ depends on the model architecture and decoding behaviour. Moreover, the alignment observed here does not imply equivalence between *AS* and *CCS*, as each captures correctness at a different level of granularity.

To identify the lists of top-performing LLM configurations based on *AS*, *CPS*, and *CCS*, we adopted strict criteria: no statistically significant difference and a *negligible* effect size in all pairwise comparisons. One may choose to relax the latter threshold—for instance, by including configuration pairs with *small* effect sizes—which would naturally expand the sets listed previously. In any case, the full heatmaps presented in Figure 4, Figure 5, and Figure 6 provide the necessary information for readers to explore such comparisons in greater detail, based on the interpretation guidance outlined earlier.

Impact of Reducing Temperature to Zero*:* To address **RQ2**, we examined whether setting an LLM's temperature to zero impacts its performance on *AS*, *CPS*, and *CCS* compared to its default temperature configuration. To facilitate this comparison, particularly between each LLM's pair of temperature settings, we refer to the data in Figure 4, Figure 5, and Figure 6, and summarise the results in Table II. In Table II, the "Significance" column reports whether the observed difference between temperature settings is statistically significant based on the Mann–Whitney U test, while the "Importance" column presents the interpretation of the corresponding Cliff's δ effect size. Within each LLM, the first group refers to the configuration at $t{=}D$, and the second to that at $t{=}0$.

For example, consider *Claude 3.5 Haiku* in Table II. The analysis reveals no statistically significant difference in *AS* and *CPS*, but a statistically significant difference in *CCS*. The effect sizes are -0.12 and 0.10 for *AS* and *CPS*, respectively—both classified as negligible. For *CCS*, the effect size

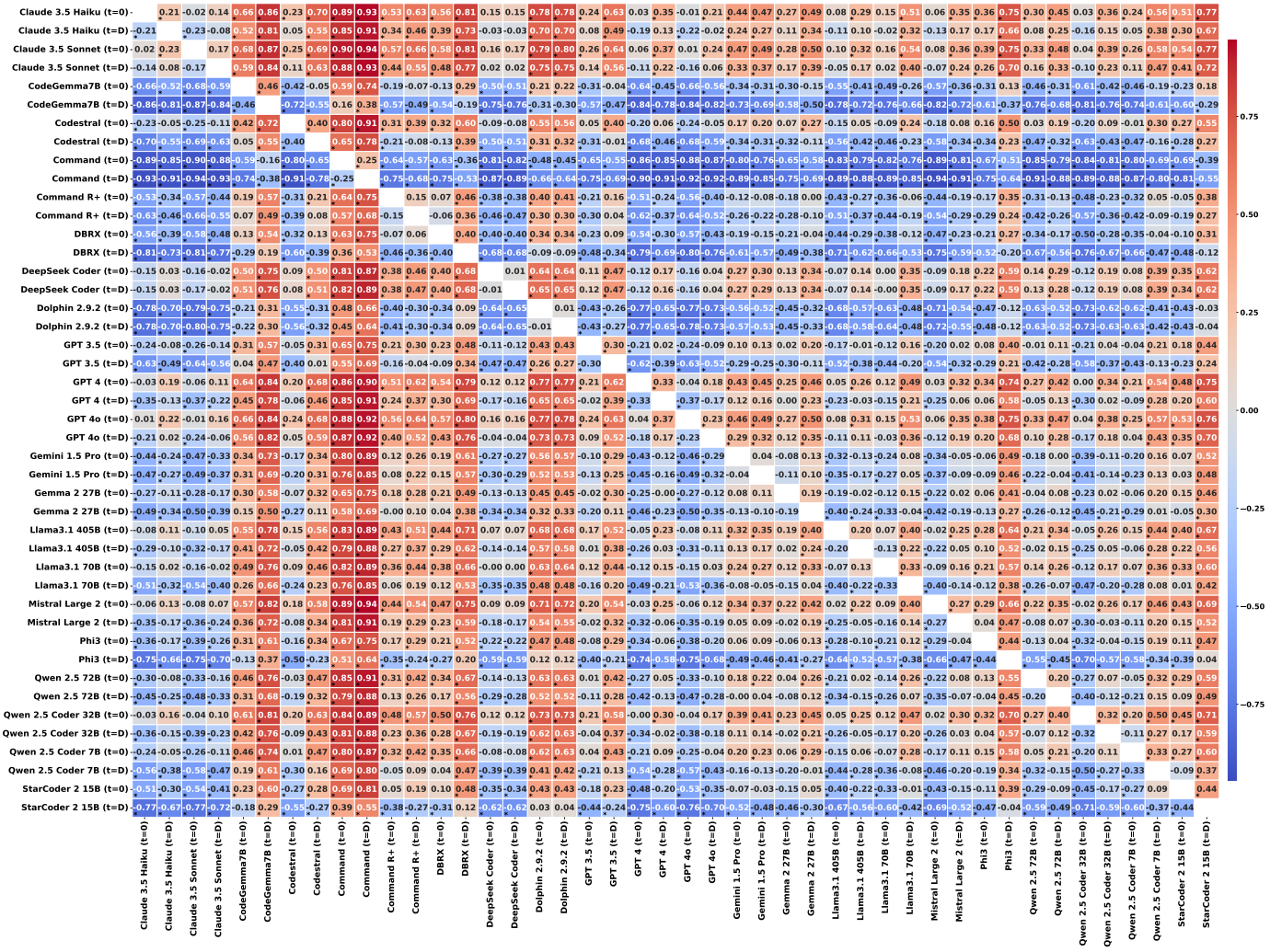

Fig. 6. Comparison of LLM configurations based on pairwise *CCS* effect size and statistical significance. Note that the heatmap is symmetric.

is -0.21, which is interpreted as small. Since negative values indicate improvement at $t = 0$ relative to $t = D$, these results suggest that *Claude 3.5 Haiku* shows slightly higher *AS* and *CCS* at $t = 0$, and slightly higher *CPS* at $t = D$. However, only the *CCS* difference is statistically significant. Yet even in that case, the effect size remains small, indicating that the observed differences are not practically meaningful. Across all models, the effect sizes for *AS* are either negligible or small, regardless of statistical significance. This suggests that the differences in *AS* between the two temperature settings are not practically meaningful. The same conclusion largely applies to *CPS*, with the exception of *Command*, where a statistically significant difference is accompanied by a positive medium effect size, indicating that the model achieves meaningfully higher correctness potential at $t = D$. With respect to *CCS*, we observe statistical significance and medium negative effect sizes for seven models—*CodeGemma-7B*, *Codestral*, *DBRX*, *Phi-3-medium*, *Llama-3.1-70B*, *Qwen2.5-Coder-7B*, and *StarCoder2-15B*—indicating notable improvement in consistent correctness under $t = 0$. For all remaining models, the effect sizes under *CCS* are either negligible or small, suggesting limited practical impact of temperature reduction.

Following all results and discussions presented in this section, we address **RQ1** and **RQ2** as follows. Regarding **RQ1**, when confronted with instances within a question neighbourhood, LLMs did not gain perfect *AS*, *CPS*, and *CCS* across all neighbourhoods. While some models achieve high scores with relatively stable performance, no model demonstrates perfect performance across all neighbourhoods. Most models exhibit some degree of variability, with fluctuations in correctness depending on the specific parameter values. This indicates that even when tasks are closely related, LLMs can fail unpredictably, highlighting limitations in their ability to generalise robustly and reliably across similar but non-equivalent inputs.

Before addressing **RQ2**, we first offer an important clarification regarding the relationship between the distributional patterns observed in Figure 3 and the statistical comparisons presented in the heatmaps in Figure 4, Figure 5, and Figure 6. An important methodological observation emerges when comparing the box plots and the corresponding statistical results visualised in the heatmaps. In Figure 3, when LLMs are evaluated under the two temperature settings, several cases show noticeable changes in the spread or shape of distributions

TABLE II
STATISTICAL TESTS AND EFFECT SIZES FOR THE IMPACT OF DECREASING TEMPERATURE TO 0 ON AS, CPS, AND CCS.

	AS			CPS			CCS		
LLM	**Significant**	**Effect Size**	**Importance**	**Significant**	**Effect Size**	**Importance**	**Significant**	**Effect Size**	**Importance**
Claude 3.5 Haiku	No	-0.12	Negligible	No	0.10	Negligible	Yes	-0.21	Small
Claude 3.5 Sonnet	No	-0.11	Negligible	No	0.03	Negligible	No	-0.17	Small
CodeGemma-7B	No	-0.02	Negligible	Yes	0.32	Small	Yes	-0.46	Medium
Gemma-2-27B	No	-0.06	Negligible	No	0.17	Small	No	-0.19	Small
Codestral	Yes	-0.22	Small	Yes	0.21	Small	Yes	-0.40	Medium
Mistral Large 2	No	-0.15	Small	No	0.07	Negligible	Yes	-0.27	Small
Command	Yes	0.24	Small	Yes	0.36	Medium	Yes	-0.25	Small
Command R+	No	-0.00	Negligible	No	0.19	Small	No	-0.15	Small
DBRX	No	-0.12	Negligible	Yes	0.22	Small	Yes	-0.40	Medium
DeepSeek-Coder-V2	No	-0.01	Negligible	No	0.00	Negligible	No	-0.01	Negligible
Dolphin 2.9.2 Mixtral 8x22b	No	-0.00	Negligible	No	0.02	Negligible	No	-0.01	Negligible
GPT-3.5-turbo	No	-0.13	Negligible	No	0.19	Small	Yes	-0.30	Small
GPT-4	Yes	-0.25	Small	No	0.09	Negligible	Yes	-0.33	Small
GPT-4o	No	-0.16	Small	No	0.06	Negligible	Yes	-0.23	Small
Gemini 1.5 Pro	No	-0.02	Negligible	No	0.02	Negligible	No	-0.04	Negligible
Phi-3-medium	Yes	-0.22	Small	Yes	0.27	Small	Yes	-0.44	Medium
Llama-3.1-70B	No	-0.17	Small	No	0.12	Negligible	Yes	-0.33	Medium
Llama-3.1-405B	No	-0.11	Negligible	No	0.09	Negligible	No	-0.20	Small
Qwen2.5-72B	No	-0.07	Negligible	No	0.20	Small	No	-0.20	Small
Qwen2.5-Coder-7B	No	-0.14	Negligible	No	0.20	Small	Yes	-0.33	Medium
Qwen2.5-Coder-32B	Yes	-0.21	Small	No	0.14	Negligible	Yes	-0.32	Small
StarCoder2-15B	No	-0.12	Negligible	Yes	0.29	Small	Yes	-0.44	Medium

across *AS*, *CPS*, and *CCS*, including shifts in interquartile ranges or outlier patterns. However, these visual differences are not always reflected in the results of statistical testing. For instance, a model may exhibit a visibly narrower or wider distribution under $t = 0$, yet the Mann–Whitney U test may indicate no statistically significant difference, and the corresponding Cliff's δ effect size may be classified as negligible. This apparent mismatch arises from the fact that box plots primarily highlight distributional characteristics, including variability, skewness, and the presence of outliers. In contrast, statistical significance testing and effect size estimation are designed to detect systematic differences in central tendency or stochastic dominance between two groups. Therefore, visual shifts in spread—such as widening IQRs under $t = 0$ in *CPS*, or compact distributions under $t = 0$ in *CCS*—may not translate into statistically significant differences if the relative ranks or central values remain similar across the compared groups.

In the case of *CPS*, for example, the box plots reveal that many models experience increased variability under $t = 0$, likely due to reduced sampling diversity. Yet, for most models, these shifts are not statistically significant and the associated effect sizes remain negligible or small, indicating that the practical impact of the observed change is limited. Similarly, for *CCS*, although $t = 0$ often leads to visibly improved consistency—as seen through higher medians and tighter distributions in Figure 3—the heatmaps show that such improvements are statistically and practically meaningful only for a subset of models with medium or large effect sizes.

These findings highlight the complementary nature of box plots and heatmaps. Box plots are well-suited for revealing qualitative patterns, distributional variation, and behavioural trends across neighbourhoods, while heatmaps provide rigorous quantitative validation of whether those patterns constitute meaningful differences. Using both tools together ensures that interpretations are visually informed and statistically justified, enabling a nuanced understanding of how LLM performance varies with temperature settings and across evaluation scores.

To answer **RQ2**, setting an LLM's temperature to zero—thereby increasing determinism—has varied effects on performance across the three evaluation scores. For *AS*, temperature reduction does not lead to practically meaningful changes; effect sizes are mostly negligible or small, even when visual differences in variability are observed in box plots. For *CPS*, in Figure 3 most models exhibit slightly reduced correctness-potential under $t = 0$, often associated with wider IQRs and explained by the loss of sampling diversity. However, according to Table II these differences are rarely statistically or practically significant. In contrast, *CCS* shows the clearest improvement under $t = 0$ with some models: 7 out of the 22 evaluated—demonstrate statistically significant and practically meaningful gains, reflected by medium effect sizes (Table II) and visibly higher medians (Figure 3). These models benefited from the reduced output variability introduced by more deterministic decoding, which increased the likelihood of generating consistently correct outputs across multiple attempts. For the remaining models, while improvements in *CCS* were often observed visually in Figure 3, the corresponding statistical differences were either insignificant or negligible in effect size, indicating limited practical impact. Thus, temperature reduction appears to offer the most consistent and measurable advantage in enhancing consistent correctness, though this benefit is model-dependent. As a practical recommendation: when reliability and reproducibility are critical—such as in safety-sensitive or testing scenarios—temperature-zero decoding should be preferred, as it increases the likelihood of consistently correct outputs.

However, because the effect is model-dependent, practitioners should validate temperature sweeps during model selection and deployment. Where exploratory diversity or creativity is important, higher temperatures may remain preferable despite the loss of consistency. Thus, our results recommend a dual strategy: (1) deploy $t = 0$ for applications demanding stable, reproducible behaviour, and (2) use higher temperatures when diversity of solutions is more valuable. For researchers, these findings further suggest that evaluation pipelines should include multiple temperatures and report consistency-oriented metrics such as *CCS*, ensuring that robustness to decoding settings is systematically assessed.

C. Results Based on Distinct Categories

Recall that each question template T is instantiated to generate a neighbourhood of 100 instances, each with distinct parameters, and each instance is given to the LLM across 5 rounds. Let $A_{q_i}^{r_j}$ denote the LLM's answer to question instance q_i in round r_j, where $1 \leq i \leq 100$ and $1 \leq j \leq 5$. We then categorise LLM performance for a question neighbourhood $\mathcal{N}_T$ into four distinct categories as follows.

Perfect Success., This category refers to when the LLM always returns a correct result ($AS = CPS = CCS = 1.0$, i.e., the LLM can solve this neighbourhood effortlessly), that is:

$$\forall q_i \in \mathcal{N}_T,\ \forall r_j, \quad A_{q_i}^{r_j} = \text{Correct}$$

Perfect Failure. This category corresponds to when the LLM does not ever return a correct result ($AS = CPS = CCS = 0.0$, i.e., the LLM cannot solve this neighbourhood at all), that is:

$$\forall q_i \in \mathcal{N}_T,\ \forall r_j, \quad A_{q_i}^{r_j} = \text{Incorrect}$$

Stochastic Failure. This category is defined by the simultaneous presence of both conditions: (i) the LLM returns at least one incorrect result, (ii) there is no question instance for which the LLM returns an incorrect result across all 5 rounds ($0 < AS < 1$, $0 \leq CCS < 1$, and $CPS = 1$, i.e, the LLM is not completely blocked on any question instance), hence:

$$\text{(i) } \exists q_i \in \mathcal{N}_T,\ \exists r_j, \quad A_{q_i}^{r_j} = \text{Incorrect}$$

$$\text{(ii) } \forall q_i \in \mathcal{N}_T,\ \exists r_j, \quad A_{q_i}^{r_j} = \text{Correct}$$

This category captures models that are potentially capable of solving every instance, but whose performance is inconsistent. It reflects high *CPS* (all instances solved at least once), but lower *CCS* (not solved consistently), and a moderate *AS* depending on the overall correctness rate. This pattern suggests that 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.

Inconsistent Generalisation. This category is characterised by the coexistence of both of the following conditions: (i) the LLM returns at least one correct result, (ii) there is at least one question instance for which the LLM returns an incorrect result across all 5 rounds ($0 < AS,\ CPS < 1$ and $0 \leq CCS < 1$, i.e., the LLM appears to be completely blocked on at least one question instance), hence:

$$\text{(i) } \exists q_i \in \mathcal{N}_T,\ \exists r_j, \quad A_{q_i}^{r_j} = \text{Correct}$$

$$\text{(ii) } \exists q_{i'} \in \mathcal{N}_T,\ \forall r_j, \quad A_{q_{i'}}^{r_j} = \text{Incorrect}$$

where $1 \leq i' \leq 100$.

This category is particularly diagnostic, capturing cases where the LLM succeeds on some instances but consistently fails on others within the same neighbourhood. Such behaviour *may* indicate a breakdown in generalisation, despite the shared structure across question instances. It typically results in non-zero *AS*, lower *CPS*, and low *CCS*, reflecting inconsistencies in the model's reasoning. Since all instances in a neighbourhood differ only in parameter values and are designed to be similarly challenging, persistent failure on a subset *can* suggest generalisation gaps or reasoning discontinuities.

Figure 7 presents the results for each LLM configuration, categorised into the four predefined groups. As indicated by the Perfect Success bars in each subplot—and as previously noted in Section IV-B—no LLM configuration achieves perfect success across all 60 question neighbourhoods, underscoring that even the most capable models exhibit occasional failures or inconsistencies. Some LLM configurations exhibit neighbourhoods that fall into the Perfect Failure category, where the model fails to produce a correct output for any instance across all five rounds. This behaviour is most pronounced in the *Command* model. These patterns suggest that, while most configurations demonstrate at least minimal capacity to solve some instances, certain models remain fundamentally incapable of handling specific subsets of tasks. For the Stochastic Failure category, the distribution of question neighbourhoods varies across LLM configurations. Some configurations exhibit no or very few question neighbourhoods in this category, whereas others display a considerable number. An example of this category was observed when *GPT-4o* at $t = 0$ was presented with a question instance in which the parameter value was set to -15 (Figure 8). Within the corresponding neighbourhood, the LLM generated at least one correct response for each question instance across five trials. However, for a small subset of question instances, including the one in Figure 8, some incorrect responses were also produced, indicating that at least one incorrect answer was generated overall. Figure 8b presents a correct code response for the corresponding question instance, while Figure 8c displays the incorrect code generated by the same LLM configuration. The incorrect response exhibits improper circular-list handling, an incorrect termination range, and faulty sublist construction.

Concerning the Inconsistent Generalisation category, as shown in Figure 7, all LLM configurations contain a substantial number of question neighbourhoods falling into this category. Among the LLM configurations, *CodeGemma-7B* at $t = D$ (52 question neighbourhoods), *StarCoder2-15B* at $t = 0$ and $t = D$ (50 and 49 neighbourhoods, respectively), *Command R+* at both temperatures (48 neighbourhoods each), *CodeGemma-7B* at $t = 0$ (47 neighbourhoods), and *DBRX* at both temperatures (44 neighbourhoods each) exhibit a higher susceptibility to inconsistent generalisation.

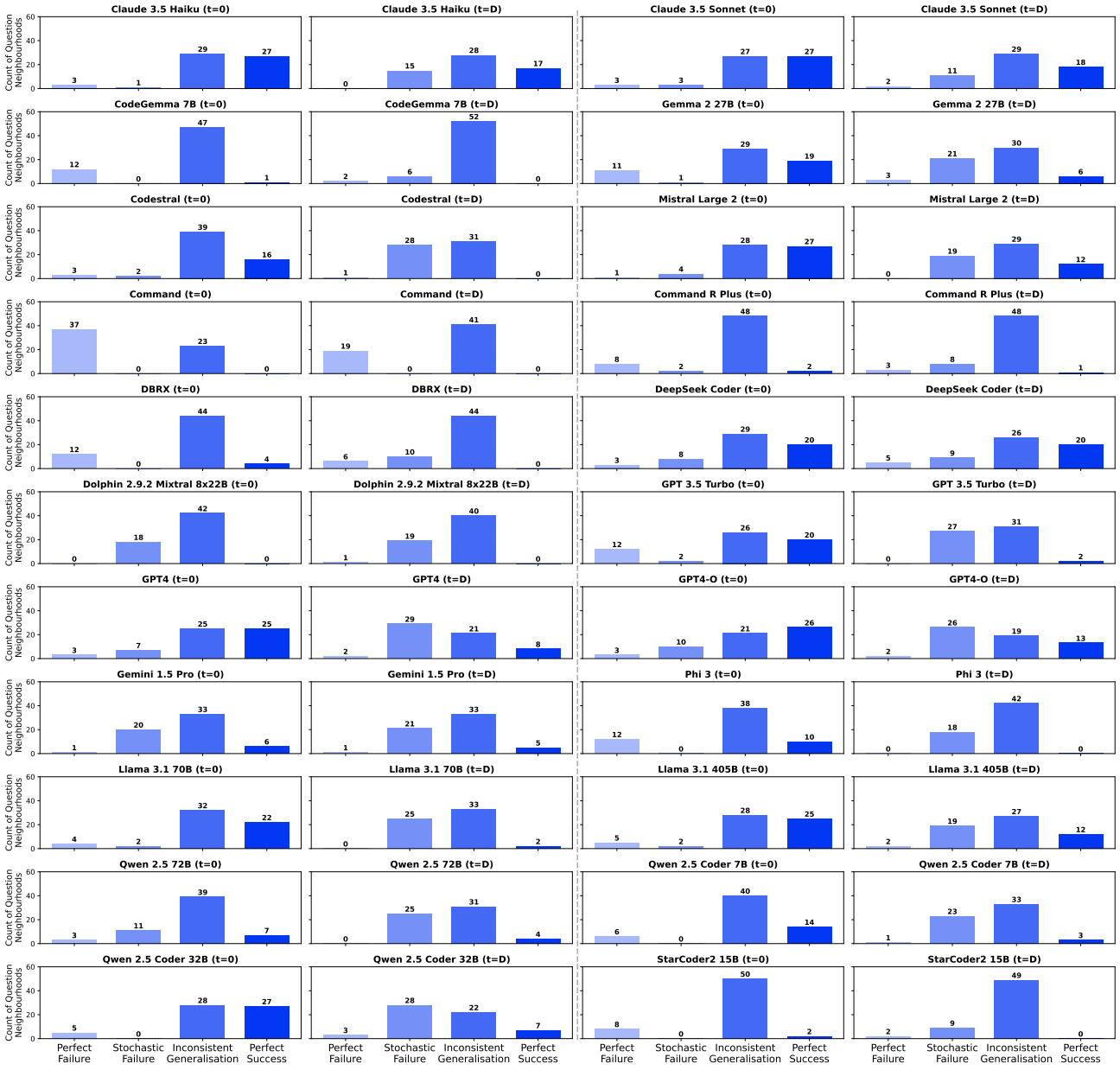

Fig. 7. LLM performance categorised into four types, with bars showing the number of question neighbourhoods per category.

An example is illustrated in Figure 9. When the question template in Figure 9a was instantiated with $p = 107$, across five attempts, the LLM generated five semantically distinct yet correct responses, indicating that at least one correct answer was generated. One such correct response is shown in Figure 9b. However, when the same template was instantiated with $p = 111$, the LLM produced five semantically distinct responses across five attempts, all of which were incorrect, demonstrating a case in which all outputs were erroneous. Figure 9c presents one such incorrect response. The generated code counts only the non-zero entries in the submatrix, rather than the total number of elements.

It is important to note that comparisons between LLM configurations—or assessments of the impact of reducing the temperature to 0—should be based on the analysis presented in Section IV-B, which considers the full distributions of *AS*, *CPS*, and *CCS*. In contrast, Figure 7 is not intended to serve as a basis for statistical comparison. Rather, it is intended to illustrate how the performance of each LLM configuration maps onto the four predefined categories. Nonetheless, it is noteworthy when an LLM successfully solves some, but not all, instances derived from a single question template. Our question neighbourhood approach reveals such discontinuities by examining LLM behaviour at a fine-grained level. In this context, the Inconsistent Generalisation category captures the phenomenon by grouping together those neighbourhoods in which the model exhibits inconsistent success across question instances. This fine-grained perspective offers valuable insights into the limitations of an LLM's reasoning capabilities.

We now present the results of our behavioural categorisation

Write a function called ‘lists_with_product_equal_n’ that takes an argument, a circular list of integers. In a circular list, the head and the tail are adjacent to each other, forming a closed loop. The function should return a list of sublists, each containing contiguous integers from the given list such that their product equals **-15**. Each sublist in the returned list can be of any size, provided it does not exceed the length of the given list. If no such sublist exists, the function should return an empty list. If multiple identical sublists satisfy the condition, all occurrences should be included in the output. The order of sublists in the returned list is not relevant.

(a) Question instance with parameter value -15.

```
def lists_with_product_equal_n(circular_list):
    n = len(circular_list)
    result = []
    for start in range(n):
        product = 1
        for end in range(start, start + n):
            product *= circular_list[end % n]
            if product == -15:
                result.append([circular_list[i % n]
                    for i in range(start, end + 1)])
            if product == 0:
                break
    return result
```

(b) Correct code returned by *GPT-4o* ($t=0$) 4 times.

```
def lists_with_product_equal_n(circular_list):
    def product_of_list(lst):
        product = 1
        for num in lst:
            product *= num
        return product
    n = len(circular_list)
    result = []
    for start in range(n):
        for end in range(1, n + 1):
            if end > n:
                sublist = circular_list[start:] +
                    circular_list[:end % n]
            else:
                sublist = circular_list[start:end]
            if product_of_list(sublist) == -15:
                result.append(sublist)
    return result
```

(c) Incorrect code returned by *GPT-4o* ($t=0$) once.

Fig. 8. Example of Stochastic Failure category.

Write a function called ‘submatrix_with_n_numbers’ that takes one argument which is a two-dimensional numpy matrix of integers. The function should return the count of all submatrices of the given matrix that contain **p** integers each. If no such submatrix is found, the function should return 0.

(a) Question template.

```
import numpy as np
def submatrix_with_n_numbers(matrix):
    rows, cols = matrix.shape
    count = 0
    for row in range(rows):
        for col in range(cols):
            for r in range(row, rows):
                for c in range(col, cols):
                    submatrix =
                        matrix[row:r+1,col:c+1]
                    if np.size(submatrix) == 107:
                        count += 1
    return count
```

(b) One of the five correct code answers returned by *Llama-3.1-405B* ($t=D$) for the question instance derived from the template in (a) with parameter value 107.

```
import numpy as np
def submatrix_with_n_numbers(matrix):
    count = 0
    rows = matrix.shape[0]
    cols = matrix.shape[1]
    n = 111
    for row in range(rows):
        for col in range(cols):
            for r in range(row, rows):
                for c in range(col, cols):
                    submatrix =
                        matrix[row:r+1,col:c+1]
                    if np.count_nonzero(submatrix)==n:
                        count += 1
    return count
```

(c) One of the five incorrect code answers returned by *Llama-3.1-405B* ($t=D$) for the question instance derived from the template in (a) with parameter value 111.

Fig. 9. Example of Inconsistent Generalisation category.

analysis, beginning with aggregated group-level trends (Table III) before moving to template-level details (Table IV).

To facilitate a structured analysis of model behaviour across diverse question types, Table III reports, for each group and subgroup, the proportion of the 44 LLM configurations whose performance falls into one of four distinct categories defined earlier: Perfect Failure (PF), Stochastic Failure (SF), Inconsistent Generalisation (IG), and Perfect Success (PS). Percentages are calculated relative to the total number of LLM configurations. Each entry in Table III reports the mean percentage of configurations falling into the specified category. For example, the value of 6.71 under %PF in List Manipulation, Total indicates that, on average, 6.71% of the 44 LLM configurations produced no correct answers in the question neighbourhoods involving list manipulation. Likewise, the value of 6.57 under %PF in the Slicing row shows that, on average, 6.57% of the configurations failed entirely in neighbourhoods specifically concerned with slicing tasks. Consistent with Table I, the sum of subgroup entries within a problem group may exceed the corresponding group total, as certain questions are associated with multiple subgroups.

This mapping—from each problem group and subgroup to the distribution of outcomes across the four categories—enables systematic cross-analysis of performance by problem type, supporting diagnostic insights, prompt design, and fine-tuning strategies. In particular, the proportion of configurations in the PF category is especially informative: it indicates which subgroups correspond to tasks that models consistently fail to solve, independent of specific parameter

TABLE III
PROPORTION OF 44 LLM CONFIGURATIONS ACROSS DISTINCT CATEGORIES PER PROBLEM GROUP AND SUBGROUP. VALUES ARE SHOWN AS PERCENTAGES.

Problem Group	Problem Subgroup	%PF	%SF	%IG	%PS
List Manipulation	Total	6.71	19.99	53.36	19.94
	Slicing	6.57	18.27	55.56	19.60
	Indexing	6.47	16.74	60.74	16.05
	Filtering	6.06	20.93	52.84	20.17
	Element-based Operations	4.37	20.02	52.97	22.64
	Summation	2.65	27.65	35.61	34.09
	Sorting/Order-based Operations	8.74	19.06	55.24	16.96
	Element Insertion	1.14	9.09	84.09	5.68
	Count elements	15.91	0.00	84.09	0.00
	Circular Lists	22.73	9.09	65.91	2.27
String Manipulation	Total	11.51	15.77	62.93	9.79
	Character Insertion	0.00	25.00	51.14	23.86
	Indexing	10.06	7.14	78.57	4.23
	Character Removal	14.39	6.06	78.79	0.76
	Substring/Character Extraction	14.58	11.74	65.91	7.77
	Palindrome Operations	22.73	0.00	77.27	0.00
	Anagram Detection	4.55	30.68	56.82	7.95
	Sorting	8.74	19.06	55.24	16.96
Set Manipulation	Total	15.91	11.11	65 .40	7.58
	Add Elements	23.48	4.92	67.05	4.55
	Subset/Superset Operation	0.00	40.91	36.36	22.73
	Counting Subsets	2.27	9.09	84.09	4.55
	Intersection	0.00	20.45	65.91	13.64
	Union	0.00	40.91	36.36	22.73
Searching	Total	8.65	18.92	53.80	18.63
	Linear Search	5.82	17.98	55.55	20.65
	Index-based Search	9.30	16.53	57.44	16.73
	String Search	11.51	15.77	62.93	9.79
Copying	Total	6.82	16.93	59.43	16.82
	Shallow Copy	1.14	9.09	84.09	5.68
	Copy Sublist	7.45	17.80	56.69	18.06
Mathematical Problems	Total	7.44	21.59	53.00	17.97
	Arithmetic Operations	4.55	20.45	62.12	12.88
	Factorial Calculations	6.82	0.00	93.18	0.00
	Prime Checking	19.55	18.18	52.73	9.54
	Composite Checking	6.82	29.55	36.36	27.27
	Factorisation	2.27	32.95	23.86	40.92
	Special Sequences	15.91	20.83	52.65	10.61
	Combinatorial Problems	2.27	4.55	90.91	2.27

instantiations. Conversely, high proportions of PS highlight areas of strength, whereas intermediate distributions across SF and IG indicate instability in generalisation. Taken together, Table III allows the community to interpret subgroup-level PF proportions as a measure of intrinsic task difficulty, providing a benchmark for identifying problem types where LLMs face fundamental limitations.

Building on this aggregated view, we next shift from groups and subgroups to the diagnostic lens of individual templates. Table IV complements Table III by drilling down into template-level results, enabling a more fine-grained characterisation of model behaviour. It includes all 60 question templates (available in our GitHub repository [31]) and, for each template, reports the template ID, its problem group (e.g., List, String, Searching), and one or more subgroups that specify functional characteristics (e.g., Slicing, Character Insertion, Index-based Search), following the taxonomy in Table I. For every template, the percentages indicate the distribution of LLM configurations across the same four behavioural categories introduced earlier. At this finer granularity, Table IV highlights local discontinuities at the level of individual question templates. In some cases, templates within groups that appear strong in Table III still exhibit high PF proportions, pointing to specific problem instances where models consistently fail. In this way, Table IV complements the broader subgroup trends of Table III by exposing template-level weaknesses and variations in robustness. Together, the two tables ground our three evaluation metrics—accuracy, correctness potential, and consistent correctness—within systematic patterns of robustness and failure.

V. EXPLORING REASONS FOR FAILURE

In this section, we address **RQ3** by analysing the primary errors in the LLM-generated code responses that led to incorrect outputs. Table V categorises these errors into nine failure types with each entry representing the percentage of responses from a given LLM configuration that fall into each category. The complete dataset supporting these results is publicly available at https://github.com/ShahinHonarvar/Turbulence-Benchmark-v2. The first three columns in Table V—*No Function*, *Wrong Function Name*, and *Wrong Count of Arguments*—correspond to the *well-formedness check* in the Turbulence test oracle (Figure 2). Recall that each question instance explicitly instructs the LLM to generate a *Python function* with a *specified name* and a *specified number of arguments* (Figure 1b). The remaining six failure categories are identified using the Pylint linter [78], which automatically detects various code issues. The *Total* column reports, for each LLM configuration, the aggregate percentage across all failure categories. This serves as a row-wise summary, indicating the overall magnitude of failures observed for that configuration and enabling direct comparison of overall failure levels across LLMs. The *Average* row reports, for each category, the mean proportion across all LLM configurations. This provides a column-wise summary, quantifying the expected frequency of each failure type when averaged over all evaluated models, thereby characterising the distribution of errors across the benchmark as a whole.

No function. This category includes instances where the LLM failed to generate a Python function, a failure observed in 16 LLM configurations on specific question instances from different neighbourhoods. Among these, *Codestral* at $t{=}D$ exhibited the highest proportion of such failures, with 3.51% of its responses falling into this category. As indicated in Table V, for LLMs affected by this issue, reducing the temperature to 0 resulted in a decrease in the number of responses classified under this failure category.

Wrong function name. This failure arises when the LLM generates a function with a name differing from the specified one, leading to errors, as each Turbulence test oracle requires functions to have the exact predefined names. There is no systematic or reliable method to correct function names, particularly when responses contain multiple functions. A total of 17 LLM configurations exhibited this issue, with *DBRX* and *Phi-3-medium* both at $t{=}D$ demonstrating the highest proportion (0.41%). With the exception of *Command* and *Dolphin 2.9.2 Mixtral 8x22b*, reducing the temperature to 0 generally led to

TABLE IV
PROPORTION OF 44 LLM CONFIGURATIONS ACROSS DISTINCT CATEGORIES PER QUESTION TEMPLATE, WITH THE ASSOCIATED PROBLEM GROUP AND SUBGROUPS. VALUES ARE SHOWN AS PERCENTAGES.

ID	Group(Subgroups)	% in PF	% in SF	% in IG	% in PS
1	List(Indexing, Slicing); Searching(Linear Search); Copying(Copy Sublist)	0.00	20.45	25.00	54.55
2	List(Indexing, Slicing); Searching(Linear Search); Copying(Copy Sublist)	25.00	2.27	70.45	2.28
3	List(Element-based Operation, Filtering, Indexing, Slicing); Searching(Linear Search); Copying(Copy Sublist)	2.27	31.82	45.45	20.46
4	List(Element-based Operation, Filtering, Indexing, Slicing); Searching(Linear Search); Copying(Copy Sublist)	22.73	0.00	77.27	0.00
5	List(Element-based Operation, Filtering, Indexing, Slicing); Searching(Linear Search); Copying(Copy Sublist)	0.00	27.27	31.82	40.91
6	List(Element-based Operation, Filtering, Indexing, Slicing); Searching(Linear Search); Copying(Copy Sublist)	18.18	0.00	81.82	0.00
7	List(Element-based Operation, Filtering, Indexing, Slicing); Searching(Linear Search); Copying(Copy Sublist)	2.27	25.00	34.09	38.64
8	List(Element-based Operation, Filtering, Indexing, Slicing); Searching(Linear Search); Copying(Copy Sublist)	18.18	0.00	81.82	0.00
9	List(Element-based Operation, Filtering, Indexing, Slicing); Searching(Linear Search); Copying(Copy Sublist)	2.27	25.00	45.45	27.28
10	List(Element-based Operation, Filtering, Indexing, Slicing); Searching(Linear Search); Copying(Copy Sublist)	9.09	0.00	90.91	0.00
11	List(Element-based Operation, Slicing, Sorting/Order-based Operations); Searching(Index-based Search, Linear Search)	0.00	15.91	31.82	52.27
12	List(Element-based Operation, Slicing, Sorting/Order-based Operations); Searching(Index-based Search, Linear Search)	0.00	13.64	34.09	52.27
13	List(Element-based Operation, Slicing, Sorting/Order-based Operations); Searching(Index-based Search, Linear Search)	2.27	20.45	72.73	4.55
14	List(Element-based Operation, Slicing, Sorting/Order-based Operations); Searching(Index-based Search, Linear Search)	2.27	11.36	84.09	2.28
15	List(Element-based Operation, Filtering, Indexing, Slicing, Summation); Searching(Linear Search)	2.27	34.09	29.55	34.09
16	List(Element-based Operation, Filtering, Indexing, Slicing, Summation); Searching(Linear Search)	0.00	45.45	27.27	27.28
17	List(Element-based Operation, Filtering, Indexing, Slicing); Searching(Linear Search); Copying(Copy Sublist); Math(Arithmetic Operations)	0.00	38.64	38.64	22.72
18	List(Element-based Operation, Filtering, Indexing, Slicing, Summation); Searching(Linear Search); Math(Arithmetic Operations)	0.00	29.55	36.36	34.09
19	List(Element-based Operation, Filtering, Indexing, Slicing); Searching(Linear Search); Copying(Copy Sublist); Math(Arithmetic Operations)	0.00	0.00	100.00	0.00
20	List(Element-based Operation, Slicing, Sorting/Order-based Operations); Searching(Index-based Search, Linear Search)	6.82	11.36	72.73	9.09
21	List(Element-based Operation, Filtering, Indexing, Slicing); Searching(Linear Search); Copying(Copy Sublist); Math(Arithmetic Operations)	0.00	31.82	47.73	20.45
22	Math(Arithmetic Operations, Special Sequences)	2.27	15.91	81.82	0.00
23	List(Filtering); Searching(Linear Search); Math(Factorisation)	0.00	36.36	13.64	50.00
24	Searching(Linear Search); Math(Factorisation)	2.27	36.36	15.91	45.46
25	List(Element Insertion, Indexing); Copying(Shallow Copy)	0.00	18.18	70.45	11.37
26	List(Element-based Operation, Filtering, Summation); Searching(Linear Search)	0.00	20.45	6.82	72.73
27	List(Element Insertion, Indexing); Copying(Shallow Copy)	2.27	0.00	97.73	0.00
28	List(Sorting/Order-based Operations); String(Sorting, String Indexing, Substring/Character Extraction); Searching(Index-based Search, Linear Search, String Search)	2.27	29.55	40.91	27.27
29	List(Element-based Operation, Indexing); Math(Factorisation)	2.27	25.00	40.91	31.82
30	String(Character Insertion); Searching(String Search)	0.00	22.73	54.55	22.72
31	List(Element-based Operation, Indexing); Math(Special Sequences)	4.55	29.55	29.55	36.35
32	String(Character Insertion); Searching(String Search)	0.00	27.27	47.73	25.00
33	String(String Indexing, Substring/Character Extraction); Searching(Linear Search, String Search); Copying(Copy Sublist)	0.00	2.27	97.73	0.00
34	Set(Subset/Superset Operation, Union)	0.00	40.91	36.36	22.73
35	String(Character Removal, String Indexing, Substring/Character Extraction); Searching(Linear Search, String Search)	18.18	0.00	81.82	0.00
36	String(Character Removal, String Indexing, Substring/Character Extraction); Searching(Linear Search, String Search)	20.45	4.55	75.00	0.00
37	String(Character Removal, String Indexing, Substring/Character Extraction); Searching(Linear Search, String Search)	4.55	13.64	79.55	2.26
38	Set(Counting Subsets); Math(Combinatorial Problems)	2.27	9.09	84.09	4.55
39	List(Sorting/Order-based Operations); String(Sorting, Substring/Character Extraction); Searching(String Search)	0.00	31.82	52.27	15.91
40	List(Sorting/Order-based Operations); String(Sorting, Substring/Character Extraction); Searching(Index-based Search, String Search)	0.00	22.73	68.18	9.09
41	List(Element-based Operation, Filtering, Indexing); Set(Intersection); Searching(Linear Search)	0.00	20.45	65.91	13.64
42	List(Element-based Operation, Indexing); Set(Add Elements); Math(Factorial Calculations, Prime Checking)	6.82	0.00	93.18	0.00
43	List(Element-based Operation, Indexing, Slicing, Sorting/Order-based Operations); Searching(Linear Search); Copying(Copy Sublist); Math(Prime Checking)	4.55	34.09	40.91	20.45
44	List(Element-based Operation, Filtering, Indexing, Slicing); Set(Add Elements); Searching(Linear Search); Math(Composite Checking)	6.82	29.55	36.36	27.27
45	String(Palindrome Operations, String Indexing, Substring/Character Extraction); Set(Add Elements); Searching(Index-based Search, Linear Search, String Search); Math(Combinatorial Problems)	2.27	0.00	97.73	0.00
46	List(Indexing); Math(Factorisation)	4.55	34.09	25.00	36.36
47	String(Palindrome Operations, String Indexing, Substring/Character Extraction); Set(Add Elements); Searching(Linear Search, String Search)	22.73	0.00	77.27	0.00
48	List(Filtering, Indexing, Slicing, Sorting/Order-based Operations, Summation); Math(Arithmetic Operations)	9.09	0.00	90.91	0.00
49	Math(Arithmetic Operations, Special Sequences)	2.27	22.73	75.00	0.00
50	List(Filtering); String(Anagram Detection); Searching(String Search)	4.55	27.27	56.82	11.36
51	List(Filtering); String(Anagram Detection); Searching(String Search)	4.55	34.09	56.82	4.54
52	String(Palindrome Operations, Substring/Character Extraction); Set(Add Elements); Searching(String Search)	43.18	0.00	56.82	0.00
53	String(Substring/Character Extraction); Set(Add Elements); Searching(String Search)	59.09	0.00	40.91	0.00
54	List(Sorting/Order-based Operations); Searching(Index-based Search); Math(Prime Checking, Special Sequences)	4.55	31.82	43.18	20.45
55	List(Circular Lists, Filtering, Slicing); Copying(Copy Sublist); Math(Arithmetic Operations)	22.73	9.09	65.91	2.27
56	String(Substring/Character Extraction); Searching(String Search); Copying(Copy Sublist)	2.27	36.36	22.73	38.64
57	List(Filtering, Slicing, Summation); Copying(Copy Sublist); Math(Arithmetic Operations)	4.55	36.36	22.73	36.36
58	List(Count elements, Filtering, Slicing)	15.91	0.00	84.09	0.00
59	List(Sorting/Order-based Operations); Searching(Index-based Search); Math(Prime Checking, Special Sequences)	15.91	25.00	52.27	6.82
60	List(Sorting/Order-based Operations); Searching(Index-based Search); Math(Prime Checking, Special Sequences)	65.91	0.00	34.09	0.00

TABLE V
PERCENTAGE OF RESPONSES BASED ON THE TEST FAILURE CATEGORIES.

LLMs	No Function	Wrong Function Name	Wrong Count of Arguments	Syntax Error	Static Type Error	Resource Exhaustion	Runtime Error	Assertion Error	Fuzzing Failure	Total
Claude 3.5 Haiku ($t=0$)	0.00	0.00	0.00	0.03	0.00	1.58	0.03	7.03	9.03	17.70
Claude 3.5 Haiku ($t=D$)	0.00	0.00	0.00	0.02	0.02	1.52	0.07	8.19	8.89	18.71
Claude 3.5 Sonnet ($t=0$)	0.00	0.00	0.00	0.00	0.00	0.42	0.00	11.97	4.23	16.62
Claude 3.5 Sonnet ($t=D$)	0.00	0.00	0.00	0.01	0.02	0.35	0.00	11.51	4.21	16.10
CodeGemma-7B ($t=0$)	0.00	0.00	0.10	2.18	4.82	0.07	0.16	49.26	2.95	59.54
CodeGemma-7B ($t=D$)	0.71	0.34	0.49	5.08	3.65	0.57	0.48	48.55	4.13	64.00
Gemma-2-27B ($t=0$)	0.00	0.00	0.00	0.20	0.44	0.34	0.26	31.00	6.07	38.31
Gemma-2-27B ($t=D$)	0.00	0.00	0.00	0.29	0.98	0.48	0.20	30.61	5.32	37.88
Codestral ($t=0$)	0.00	0.00	1.23	0.02	0.02	1.60	0.00	22.72	9.84	35.43
Codestral ($t=D$)	3.51	0.01	0.74	0.07	0.04	1.34	0.02	23.56	7.91	37.20
Mistral Large 2 ($t=0$)	0.00	0.00	0.00	0.00	0.00	1.60	0.00	14.37	7.61	23.58
Mistral Large 2 ($t=D$)	0.00	0.00	0.13	0.18	0.02	1.59	0.03	15.46	7.52	24.93
Command ($t=0$)	0.00	0.04	0.00	5.18	2.91	0.00	0.63	78.37	7.32	94.45
Command ($t=D$)	0.46	0.04	0.02	2.41	4.86	0.15	0.73	78.46	6.57	93.70
Command R+ ($t=0$)	0.00	0.00	0.02	0.04	0.24	1.05	0.29	31.75	8.50	41.89
Command R+ ($t=D$)	0.00	0.01	0.02	0.07	0.31	1.01	0.24	32.36	7.62	41.64
DBRX ($t=0$)	0.05	0.01	0.27	2.16	2.62	0.06	0.29	36.86	6.26	48.58
DBRX ($t=D$)	0.34	0.41	1.12	7.51	2.30	0.44	0.32	37.59	4.82	54.85
DeepSeek-Coder-V2 ($t=0$)	0.00	0.00	0.17	0.03	0.02	0.05	0.00	15.44	8.99	24.70
DeepSeek-Coder-V2 ($t=D$)	0.00	0.00	0.24	0.01	0.04	0.03	0.00	15.62	9.09	25.03
Dolphin 2.9.2 Mixtral 8x22b ($t=0$)	0.54	0.11	0.46	1.80	2.17	0.04	0.14	36.61	6.30	48.17
Dolphin 2.9.2 Mixtral 8x22b ($t=D$)	0.90	0.10	0.46	1.89	2.22	0.04	0.19	36.42	6.14	48.36
GPT-3.5-turbo ($t=0$)	0.00	0.00	0.00	0.42	0.79	0.32	0.02	31.31	6.01	38.87
GPT-3.5-turbo ($t=D$)	0.13	0.07	0.00	0.51	0.86	0.44	0.11	30.68	6.45	39.25
GPT-4 ($t=0$)	0.00	0.00	0.00	0.00	0.03	0.80	0.00	10.38	4.37	15.58
GPT-4 ($t=D$)	0.00	0.24	0.00	0.11	0.04	0.79	0.01	12.38	4.62	18.19
GPT-4o ($t=0$)	0.00	0.00	0.00	0.01	0.00	1.03	0.00	9.23	5.41	15.68
GPT-4o ($t=D$)	0.01	0.01	0.01	0.07	0.01	0.92	0.00	10.71	4.49	16.23
Gemini 1.5 Pro ($t=0$)	0.07	0.00	0.00	0.46	1.49	0.00	0.00	19.75	6.01	27.78
Gemini 1.5 Pro ($t=D$)	0.17	0.00	0.00	0.53	1.53	0.01	0.01	19.64	6.20	28.09
Phi-3-medium ($t=0$)	0.00	0.00	0.05	0.07	0.91	0.87	0.05	25.75	9.22	36.92
Phi-3-medium ($t=D$)	0.05	0.41	0.60	0.82	0.83	0.70	0.19	33.55	6.85	44.00
Llama-3.1-70B ($t=0$)	0.00	0.00	0.02	0.00	0.12	0.00	0.05	17.68	11.11	28.98
Llama-3.1-70B ($t=D$)	0.01	0.01	0.06	0.17	0.24	0.02	0.04	19.06	9.72	29.33
Llama-3.1-405B ($t=0$)	0.00	0.00	0.00	0.00	0.01	0.00	0.05	16.17	7.95	24.18
Llama-3.1-405B ($t=D$)	0.00	0.00	0.00	0.06	0.03	0.01	0.04	16.51	8.45	25.10
Qwen2.5-72B ($t=0$)	0.08	0.00	0.01	0.03	0.01	1.60	0.02	20.43	7.80	29.98
Qwen2.5-72B ($t=D$)	0.11	0.01	0.04	0.21	0.08	1.55	0.06	21.08	7.30	30.44
Qwen2.5-Coder-7B ($t=0$)	0.00	0.00	0.02	1.20	1.43	0.08	0.17	18.25	8.48	29.63
Qwen2.5-Coder-7B ($t=D$)	0.00	0.02	0.25	0.62	1.33	0.18	0.08	21.51	7.61	31.60
Qwen2.5-Coder-32B ($t=0$)	0.00	0.00	0.00	0.12	0.02	0.00	0.03	12.58	7.74	20.49
Qwen2.5-Coder-32B ($t=D$)	0.00	0.00	0.00	0.98	0.01	0.07	0.04	13.26	7.42	21.78
StarCoder2-15B ($t=0$)	0.00	0.00	0.70	0.43	3.85	0.00	0.00	32.51	5.65	43.14
StarCoder2-15B ($t=D$)	0.06	0.10	0.70	0.62	1.61	0.35	0.14	37.91	6.45	47.94
Average	0.16	0.04	0.18	0.83	0.98	0.55	0.12	25.55	6.92	35.33

a decrease in the number of responses classified under this failure category for the affected LLMs.

Wrong number of arguments. In this failure category, the LLM generates a function with the correct name but an incorrect number of arguments, rendering it incompatible with the test oracle. This issue was observed in 25 LLM configurations, with *Codestral* at $t = 0$ exhibiting the highest proportion (1.23%). Similar to the previously discussed failure categories, reducing the temperature to 0 generally decreased the occurrence of this issue among impacted LLMs, with the exception of *Codestral*, *Command R+*, *Dolphin 2.9.2 Mixtral 8x22b*, and *StarCoder2-15B*.

Syntax errors. In this category, the LLM output was unparseable due to Python syntax errors, including invalid characters, assignment issues, expression errors, generator/comprehension issues, unmatched brackets and parentheses, indentation and block structure errors, string and Unicode errors, as well as linting and variable-related issues. This failure was observed in 39 LLM configurations, with *DBRX* at $t = D$ exhibiting the highest proportion (7.51%). Consistent with the

trends in the previous categories, reducing the temperature to 0 generally decreased the occurrence of this issue across affected LLMs; however, this trend was reversed for *Claude 3.5 Haiku*, *Command*, *DeepSeek-Coder-V2*, and *Qwen2.5-Coder-7B*, where the number of unparseable responses increased when the temperature was set to 0.

Static type errors. The integrated Pylint linter [78] detected static type errors in the generated code, encompassing issues such as undefined variables, unsubscriptable objects, incorrect module or attribute usage, type errors, duplicate function definitions, set and dictionary issues, invalid format or encoding errors, and other miscellaneous errors. This failure was observed in 40 LLM configurations, with *Command* at $t{=}D$ exhibiting the highest proportion (4.58%). Reducing the temperature to 0 led to a decrease in the number of responses classified under this category for 16 LLMs; however, it resulted in an increase for 6 LLMs.

Resource exhaustion error. This category encompasses cases where the generated code exceeded computational resources, such as time or memory, during execution, despite the availability of more efficient solutions. This issue was identified in 38 LLM configurations, with *Codestral*, *Mistral Large 2*, and *Qwen2.5-72B*, all at $t = 0$, exhibiting the highest proportion (1.6%). For example, given the question template: *Write a function called 'find_subset_of_length_n' that takes one argument, a set of elements, and returns the number of all its subsets of size* ***p***, most LLM configurations generated resource-intensive functions when the template was instantiated with $p > 60$. One such case involved *GPT-4o* ($t = D$), which was given a question instance of the template with $p = 90$ and generated a function to return the number of subsets of size 90. The model's answer included `len(list(combinations(elements,90)))`, where "combinations" is a function from Python's itertools module that generates all possible subsets of a given size. By converting the iterator to a list, this implementation constructs and stores all such subsets in memory before counting them, which makes it highly inefficient for large sets. A more scalable alternative is `math.comb(len(elements),90)`, which performs efficiently regardless of set size. Reducing the temperature mitigated this issue in 10 LLMs, had no effect in one LLM, and led to a reversal of the trend in 10 LLMs.

Runtime errors. This category includes instances where executing the generated code resulted in runtime errors, such as operations between incompatible types, access to uninitialized local variables, indexing and slicing errors, unpacking errors, use of non-iterable objects in iteration, and invalid function return types. These errors were observed in 33 LLM configurations, with *Command* at $t{=}D$ exhibiting the highest proportion (0.73%). Reducing the temperature generally decreased the frequency of such errors across impacted LLMs, except for *Gemma-2-27B*, *Command R+*, *Llama-3.1-70B*, *Llama-3.1-405B*, and *Qwen2.5-Coder-7B*.

Assertion errors and fuzzing failures. These two categories encompass cases where the code executed but was functionally incorrect, without aligning with the errors in the previous seven categories. The following examples illustrate this category. One common source of functional errors involved inaccuracies in handling index or numerical ranges, or misinterpretation of the given question instance. We identified a pattern involving 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: $(0, x), [0, x], (x, x), [x, x], (x, x + 1)$ and $[x, x + 1]$, with parentheses indicating exclusivity and square brackets indicating inclusivity. For example, when *Claude 3.5 Haiku* ($t{=}D$) was asked to return all positive integers from index 1 to 7, both exclusive, it incorrectly returned integers from $[1 : 7]$ instead of the correct range $[2 : 7]$. Another issue arose with *Claude 3.5 Sonnet* ($t = 0$), as illustrated in Figure 10a. In this case, instead of slicing the given list according to the specified indices, the LLM-generated code returned all integers between and including the values at those indices. Given an input list of $[4, 6, 8]$, the expected output was $[4, 6]$, whereas the generated code erroneously produced $[4, 5, 6]$. Another similar edge case occurs with *Qwen2.5-Coder-32B* ($t = D$), as illustrated in Figure 10b. The issue with the generated code is that it always returns 0, regardless of the value of n passed to the function. A counterexample is when $n = 3$, where the expected output should be 3; however, the code incorrectly returns 0. This suggests that the model has confused the range $[3, 4]$, which includes the boundaries, with the range $(3, 4)$, which excludes them.

Another type of functional error involves partially correct responses that overlook specific inputs. In Figure 11a, code generated by *GPT-3.5-turbo* ($t{=}D$) fails to return 2 when 2 occurs at index 85, due to using `if number > 2` instead of `if number >= 2`, thereby excluding 2 as a prime.

Logical miscalculations also arise. In Figure 11b, code from *Gemini 1.5 Pro* ($t{=}0$) underestimates the number of anagram pairs by counting only groups with at least two anagrams, rather than computing the total number of pairs.

In the context of string manipulation, an example of erroneous substitution logic occurred when inserting a character before another in a string if either was a space character. For example, in Figure 11c, the corresponding question instance and the generated code by *Codestral* ($t = 0$) are presented. The generated code incorrectly places `W` immediately before each space rather than immediately after it, resulting in an unintended modification of the string structure.

VI. Threats to Validity

Our assessment of LLM correctness depends on (a) unambiguous questions, (b) accurate test oracles that do not misclassify correct answers as incorrect, and (c) strong test oracles capable of detecting errors. For (a) and (b), although skilled Python programmers reviewed our questions and test oracles (see Section III), some ambiguities or oracle errors may remain. For (c), we combine regression testing with random differential testing to robustly assess LLM responses, but acknowledge that no testing approach can be fully exhaustive.

We evaluated 22 LLMs, so our findings are limited to this set. However, Turbulence is designed to accommodate additional models in future work. Ideally, we would have included

Write a function called ‘all_pos_ints_inclusive’ that takes one argument, a list of integers, and returns the list of all positive integers from index **0** to index **1**, both inclusive. If no positive integers exist in the specified range, the function should return an empty list.

```
def all_pos_ints_inclusive(numbers):
   if len(numbers) < 2:
      return []
   start = numbers[0]
   end = numbers[1]
   if start > end:
      return []
   return [num for num in range(start, end+1)
       if num > 0]
```

(a) Incorrect code generated by *Claude 3.5 Sonnet* ($t=0$).

Write a function called ‘sum_of_divisors_in_range’ that takes a positive integer as its argument, and returns the sum of all positive divisors of the given integer that are between **3** and **4**, both inclusive. If no such divisors exist in the specified range, the function should return 0.

```
def sum_of_divisors_in_range(n):
   return 0
```

(b) Incorrect code generated by *Qwen2.5-Coder-32B* ($t=D$).

Fig. 10. Examples of functional errors in LLM-generated code.

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
```

(a) Incorrect code generated by *GPT-3.5-turbo* ($t=D$).

Write a function called ‘if_contains_anagrams’ that takes one argument, a list of strings. The function should return true if there are at most **5** pairs of anagrams in the given list; otherwise, it should return false. The function should only consider the anagrams of the English letters. The function should only consider the anagrams of the length of at least three. The function should compare the strings in a case-insensitive manner.

```
from collections import defaultdict
def if_contains_anagrams(list_of_strings):
   anagram_dict = defaultdict(list)
   for word in list_of_strings:
      if len(word) >= 3:
         sorted_word =
             ''.join(sorted(word.lower()))
         anagram_dict[sorted_word].append(word)
   count = 0
   for key, value in anagram_dict.items():
      if len(value) >= 2:
         count += 1
   if count <= 5:
      return True
   else:
      return False
```

(b) Incorrect code generated by *Gemini 1.5 Pro* ($t=0$).

Write a function called ‘insert_after_character’ that takes one argument, a string, and returns a string that is identical except that character **‘W’** has been inserted immediately succeeding any occurrences of character **‘ ’**.

```
def insert_after_character(s):
   return s.replace(" ", "W ")
```

(c) Incorrect code generated by *Codestral* ($t=0$).

Fig. 11. Examples of functional errors in LLM-generated code.

GitHub Copilot [79], but its lack of programmatic access makes large-scale evaluation challenging, though we note that workarounds for this are available [13], [80]. We would also have liked to include results for OpenAI’s o1 model [69] which was state-of-the-art when we ran our experiments. However, access to its API was restricted to Tier 5 users, while our account was limited to Tier 3. Furthermore, even with access, the associated costs would have been prohibitive at the time, due to the model’s substantially higher per-token pricing and its tendency to produce extended intermediate chain-of-thought outputs, leading to increased token usage.

To balance financial constraints, we set M and R (as defined in Definition 2, with M also in Definition 3 and Definition 4) to 100 parameter settings per template and five repeat runs per instance to capture sampling variance. Although seemingly arbitrary, these values reflect resource limits; with greater resources, we would choose larger ones.

To minimise training data bias [28], we created original question templates rather than reusing internet-sourced problems. While some may resemble public questions, they are tailored to the question neighbourhood methodology. Unlike typical programming tasks, where parameters remain formal user inputs, our artificial questions provide two advantages: (i) they target hard-to-find edge cases, and (ii) they enable reproducible, comparable evaluation across models.

Our study is based on 60 question neighbourhoods, which may not capture the full range of real-world programming tasks. Nevertheless, the Turbulence framework is designed for extensibility and can readily incorporate additional question templates to enhance coverage and generalisability. It remains flexible for future studies using different question sets, including broader or domain-specific evaluations.

VII. Related Work

Numerous benchmarks [81] have been introduced to evaluate LLMs across domains, including code generation. Here,

we focus on benchmarks and studies relevant to assessing the code generation capabilities of LLMs. We then highlight the key distinctions between these efforts and our study.

Benchmarks for code correctness and general evaluation. A variety of benchmarks have been developed to evaluate the correctness of code generated by LLMs. Notable among them are HumanEval [3], HumanEval-X [82], APPS [4], MBPP [9], BigCodeBench [83], CodeContests [84], and CodeXGLUE [85]. These datasets primarily focus on standalone coding tasks and functional correctness. Xu et al. [10] extended these evaluations to multilingual LLMs, assessing correctness and perplexity, while Nguyen and Nadi [5] evaluated Copilot's [79] performance using LeetCode problems across four programming languages. Moradi et al. [86] compared Copilot's solutions against human-written ones, and Zeng et al. [87] introduced CoderUJB, a runnable Java benchmark built from real-world projects that enables execution-based, context-rich evaluation of 2,239 programming tasks. Yuan et al. [12] emphasised the importance of fine-tuning, while Du et al. [88] highlighted the challenges of class-level code generation using ClassEval. SWE-bench, introduced by Jimenez et al. [89], targets real GitHub issues and evaluates bug fixing and feature implementation across large codebases. Bradbury and More [28] introduced HumanEval_T, a template-based variant of HumanEval that generates distinct but semantically equivalent tasks to reduce data leakage. Thakur et al. [90] evaluated LLMs on Verilog code generation, a domain largely overlooked in mainstream benchmarks.

Existing benchmarks primarily assess correctness on isolated tasks. In contrast, our methodology examines sets of non-equivalent tasks, enabling the identification of selective model successes within a structured neighbourhood.

Evaluation frameworks and metrics. Numerous tools and metrics have been proposed to improve LLM evaluation. DeepEval [91] by Confident AI [92] is an open-source unit-testing framework for LLM outputs, supporting over 14 metrics. Manh [93] presented CodeLLM Evaluator, a framework supporting standard benchmarks, backends, and adapter-based evaluations. IdentityChain [94] assessed self-consistency in code generation and comprehension, while CCTest [95] leveraged structural consistency to improve completion. ICE-Score [96] is a reference-free, multi-dimensional metric focused on semantic alignment, and CodeJudge [97] evaluates semantic correctness without requiring test cases. CodeScore, developed by Dong et al. [98], uses a fine-tuned LLM trained via UniCE to predict executability and test pass rates across Ref-only, NL-only, and hybrid inputs. Ren et al. [99] introduced CodeBLEU, combining token overlap, syntax, and semantic similarity. RACE [100] evaluates correctness, readability, maintainability, and efficiency. Allamanis et al. [101] proposed Round-Trip Correctness, a test-free metric based on semantic consistency between forward and inverse tasks.

These frameworks propose diverse metrics, but remain tied to single-instance evaluation. We contribute a neighbourhood-based methodology with three complementary scores that capture reliability across related tasks.

Robustness and adversarial testing. LLM robustness to prompt variation and adversarial attacks has received growing attention. Shirafuji et al. [18] and Wang et al. [16] showed that small prompt changes can significantly impact model outputs. Döderlein et al. [17] observed that Copilot and Codex [102] are sensitive to minor prompt perturbations, though temperature tuning can help. Mastropaolo et al. [13] investigates whether GitHub Copilot generates different code from semantically equivalent method descriptions. RADAR [103] and Anand et al. [21] confirmed that small syntactic modifications degrade model performance. Several frameworks aim to improve robustness: CLAWSAT [104] uses contrastive adversarial training; CoTR [19] applies syntactic transformations to defend against attacks; MHM [105] and CodeBERT-Attack [106] generate semantically equivalent adversarial examples; CODA [107] and CARROT [108] focus on detecting and improving adversarial robustness. Zeng et al. [109] compared LLMs across diverse tasks, revealing their susceptibility to adversarial inputs. Techniques like DAMP [110] and obfuscation [111] produce semantically equivalent attacks. CoCoFuzzing [112] uses coverage-guided fuzzing to probe models, while ALERT [14] targets LLMs with naturalistic adversarial prompts.

Prior studies generally rely on perturbations that yield equivalent prompts. By contrast, our approach constructs neighbourhoods of non-equivalent tasks, thereby exposing systematic limitations in model generalisation that extend beyond adversarial fragility and revealing where models succeed selectively within structured variation.

Code quality, vulnerability, and security. Multiple studies have examined the quality and security of LLM-generated code. Tambon et al. [113] analysed 333 Python functions and identified 10 common bug patterns. Siddiq et al. [114] found code smells in the training data and outputs of GPT-CodeClippy and Copilot, raising concerns about maintainability and security. Asare et al. [115] showed Copilot replicates human vulnerabilities in 33% of cases and generates fixes in 25%. Khoury et al. [116] demonstrated that ChatGPT can improve code security when explicitly prompted. Pearce et al. [117] found Copilot frequently produces insecure code and evaluated its vulnerability across prompt types and domains. Wong et al. [118] formally verified Copilot's outputs for specification compliance. Lahiri et al. [119] proposed TiCoder, which integrates user feedback and validation to enhance trust. Al-Kaswan et al. [120] analysed memorisation in LLMs, highlighting legal and security risks.

These quality- and security-focused studies examine bugs and vulnerabilities in generated code. By contrast, our work analyses correctness and consistency across task families, exposing reliability gaps rather than specific defect types.

Evaluation of logical and semantic reasoning. Several works investigate reasoning in code generation. REval [121] evaluates logical consistency through program execution. COCO [20] tests semantic alignment by modifying programming instructions, while LogicAsker [122] uses logic-based test cases to assess reasoning. Rajan et al. [123] developed KONTEST, combining knowledge graphs and metamorphic oracles to detect inconsistencies. Dozono et al. [124] intro-

duced CODEGUARDIAN to identify common weaknesses and accelerate detection in IDEs. Jian et al. [125] proposed a debugging and explanation framework for LLMs, and Yang et al. [126] evaluated LLMs for Java unit test generation.

Reasoning-focused evaluations address logic or semantics within single problems. In contrast, we assess whether models reason consistently across neighbourhoods of related tasks, exposing discontinuities invisible at the instance level.

Survey and meta-evaluation studies. Multiple surveys have examined the landscape of LLM evaluation for code. Wang et al. [127] reviewed 20 studies and identified gaps in evaluating code quality and trustworthiness. Chen et al. [128] outlined common metrics and benchmarks, discussing their limitations and future directions. Chang et al. [11] highlighted the need for scalable, robust evaluation methodologies, particularly given claims that LLMs are nearing AGI-level capabilities.

Beyond all of the related work discussed above, the most closely related study to our work is the benchmark proposed by Bradbury and More [28], which introduces HumanEval_T to address data contamination by generating semantically equivalent variants of existing programming tasks using combinatorial test design. Their approach ensures benchmark integrity and fairness by constructing variants robust against potential data leakage, but the tasks remain equivalent in meaning. Our work builds on this idea of systematic variation while shifting the focus: instead of generating equivalent variants, we construct neighbourhoods of semantically similar yet distinct tasks. This minimal semantic divergence allows us to assess not just robustness to superficial reformulation, but also the stability of LLM generalisation across genuinely different problem instances. Another closely related study is that of Shirafuji et al. [18], which investigates how syntactic modifications—such as renaming variables or rephrasing prompts—affect the correctness of generated code while preserving task semantics. Their goal is to assess the sensitivity of LLMs to superficial changes in input formulation. Indeed, most prior studies assessing the robustness of LLMs in code generation adopt semantics-preserving transformations, analysing model behaviour under variations that maintain the original task intent.

In contrast, we propose Turbulence, a fully automated evaluation framework designed to assess the performance of LLMs across neighbourhoods of question instances that are semantically close but not equivalent. Each neighbourhood is generated from a parameterised template, where variations in template parameters yield distinct tasks that, while conceptually similar, differ in semantic content. By leveraging minimal semantic divergence, Turbulence provides a novel perspective on model behaviour. The framework enables systematic evaluation along three complementary dimensions—accuracy, correctness potential, and consistent correctness—capturing both task-level correctness and the stability and generalisation of LLMs across related tasks.

Our method does not rely on adversarial training or manual augmentation; instead, we systematically vary template parameters to uncover generalisation gaps. Furthermore, while existing testing frameworks primarily rely on unit tests, our tool combines unit testing with random differential testing, enabling a more scalable and nuanced assessment of code correctness. This is achieved through a hybrid of fixed test suites and fuzzing, facilitating behaviour-driven evaluation tailored to the challenges of instruction-tuned code generation.

Overall, our benchmark goes beyond task-level correctness by offering a structured methodology to reveal model weaknesses in generalising across semantically similar yet distinct problems. Using accuracy, correctness potential, and consistent correctness as core metrics, it offers a more nuanced characterisation of LLM behaviour in structured problem spaces.

VIII. Conclusions and Future Work

We introduced a new method for evaluating the accuracy, correctness potential, and consistent correctness of LLMs in code generation, grounded in the concept of *question neighbourhoods*. Assessing performance across a question neighbourhood enables the identification of not only individual problem instances that an LLM fails to solve, but also broader gaps in the model's ability to generalise within a structured problem space. We operationalised this approach through Turbulence, the first benchmark to systematically evaluate code-generating LLMs using question neighbourhoods. Our experiments with 22 models revealed that all exhibited deficiencies in at least one of the three evaluation metrics across certain neighbourhoods. Reducing the decoding temperature from the default to 0 had minimal impact on accuracy and correctness potential but led to meaningful improvements in consistent correctness for 7 out of 22 models—highlighting the model-dependent benefits of reduced output variability. Promising future directions include examining the impact of quantisation on LLM performance and extending the question neighbourhood approach to code-infilling models.

References

[1] J. D. Weisz *et al.*, "Perfection not required? Human-AI partnerships in code translation," in *IUI: College Station, TX, USA, April 13-17, 2021*, T. Hammond *et al.*, Eds. 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, Prague, Czech Republic, July 3-5, 2023, Proceedings, Part II*, ser. Lecture Notes in Computer Science, J. Mikyska *et al.*, Eds., vol. 14074. Springer, 2023, pp. 171–179. [Online]. Available: https://doi.org/10.1007/978-3-031-36021-3_15

[3] 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

[4] D. Hendrycks *et al.*, "Measuring coding challenge competence with APPS," in *NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html

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

[6] 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, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh *et al.*, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html

[7] 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, Sydney, NSW, Australia, March 27-31, 2023*. ACM, 2023, pp. 491–514. [Online]. Available: https://doi.org/10.1145/3581641.3584037

[8] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?" in *CCS, Copenhagen, Denmark, November 26-30, 2023*, 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

[9] J. Austin *et al.*, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[10] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *MAPS@PLDI: 6th ACM SIGPLAN, San Diego, CA, USA, 13 June 2022*, S. Chaudhuri and C. Sutton, Eds. ACM, 2022, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3520312.3534862

[11] Y. Chang *et al.*, "A survey on evaluation of large language models," *ACM Trans. Intell. Syst. Technol.*, vol. 15, no. 3, Mar. 2024. [Online]. Available: https://doi.org/10.1145/3641289

[12] Z. Yuan *et al.*, "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

[13] 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, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2149–2160. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00181

[14] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *ICSE, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1482–1493. [Online]. Available: https://doi.org/10.1145/3510003.3510146

[15] Z. Tian, J. Chen, and Z. Jin, "Code difference guided adversarial example generation for deep code models," in *ASE, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 850–862. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00149

[16] S. Wang *et al.*, "ReCode: Robustness evaluation of code generation models," in *ACL, Toronto, Canada, July 9-14, 2023*, A. Rogers *et al.*, Eds. ACL, 2023, pp. 13 818–13 843. [Online]. Available: https://doi.org/10.18653/v1/2023.acl-long.773

[17] 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

[18] A. Shirafuji *et al.*, "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

[19] G. Yang *et al.*, "Assessing and improving syntactic adversarial robustness of pre-trained models for code translation," *Inf. Softw. Technol.*, vol. 181, p. 107699, 2025. [Online]. Available: https://doi.org/10.1016/j.infsof.2025.107699

[20] M. Yan *et al.*, "COCO: Testing code generation systems via concretized instructions," *CoRR*, vol. abs/2308.13319, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.13319

[21] 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

[22] T. Y. Zhuo *et al.*, "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

[23] 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

[24] M. Gardner *et al.*, "Evaluating models' local decision boundaries via contrast sets," in *EMNLP, Online Event, 16-20 November 2020*, 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

[25] A. Saparov and H. He, "Language models are greedy reasoners: A systematic formal analysis of chain-of-thought," in *ICLR, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/forum?id=qFVVBzXxR2V

[26] F. Shi *et al.*, "Language models are multilingual chain-of-thought reasoners," in *ICLR, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/forum?id=fR3wGCk-IXp

[27] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *NeurIPS, New Orleans, LA, USA, November 28 - December 9, 2022*, 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

[28] J. S. Bradbury and R. More, "Addressing data leakage in HumanEval using combinatorial test design," in *ICST, Napoli, Italy, March 31 - April 4, 2025*. IEEE, 2025, pp. 587–591. [Online]. Available: https://doi.org/10.1109/ICST62969.2025.10989022

[29] S. Honarvar, M. van der Wilk, and A. F. Donaldson, "Turbulence: Systematically and automatically testing instruction-tuned large language models for code," in *ICST, Napoli, Italy, March 31 - April 4, 2025*. IEEE, 2025, pp. 80–91. [Online]. Available: https://doi.org/10.1109/ICST62969.2025.10989005

[30] 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

[31] S. Honarvar and A. Donaldson, "Turbulence-Benchmark-v2," https://github.com/ShahinHonarvar/Turbulence-Benchmark-v2, 2025.

[32] ——, "Turbulence-Benchmark-v2," https://dx.doi.org/10.21227/pmmv-nt11, 2025.

[33] L. Yuan *et al.*, "Revisiting out-of-distribution robustness in NLP: Benchmarks, analysis, and llms evaluations," in *NeurIPS, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh *et al.*, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/b6b5f50a2001ad1cbccca96e693c4ab4-Abstract-Datasets_and_Benchmarks.html

[34] F. Cassano *et al.*, "MultiPL-E," https://github.com/nuprl/MultiPL-E/blob/main/dataset_builder/terms.csv, 2023.

[35] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," in *CHI: Virtual Event / Yokohama Japan, May 8-13, 2021, Extended Abstracts*, 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

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

[37] ——, "Models," https://platform.openai.com/docs/models, 2024.

[38] Cohere, "Models Overview," https://docs.cohere.com/v2/docs/models#command, 2024.

[39] Cohere, "We're building the future of language AI," https://cohere.com/about, 2024.

[40] A. PBC, "Meet Claude," https://www.anthropic.com/claude/haiku, 2024.

[41] Anthropic PBC, "AI research and products that put safety at the frontier," https://www.anthropic.com/, 2024.

[42] A. PBC, "Claude 3.5 Sonnet," https://www.anthropic.com/news/claude-3-5-sonnet, 2024.

[43] A. PBC, "Claude 3.5 Haiku," https://www.anthropic.com/claude, 2024.

[44] M. Reid *et al.*, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," *CoRR*, vol. abs/2403.05530, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.05530

[45] Google, "Build AI responsibly to benefit humanity," https://deepmind.google/about/, 2024.

[46] T. Mesnard *et al.*, "Gemma: Open models based on Gemini research and technology," *CoRR*, vol. abs/2403.08295, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.08295

[47] deepinfra, "Fast ML inference, simple API," https://deepinfra.com/, 2024.

[48] H. Zhao *et al.*, "CodeGemma: Open code models based on Gemma," *CoRR*, vol. abs/2406.11409, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2406.11409

[49] Google Cloud, "Innovate faster with enterprise-ready AI, enhanced by Gemini models," https://cloud.google.com/vertex-ai?hl=en, 2024.

[50] DeepSeek-AI *et al.*, "DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence," *CoRR*, vol. abs/2406.11931, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2406.11931

[51] deepseek, "deepseek Into the unknown," https://www.deepseek.com, 2024.

[52] Mistral-AI, "Large Enough," https://mistral.ai/news/mistral-large-2407/, 2024.

[53] Mistral-AI, "Mistral AI team," https://mistral.ai/company/, 2024.

[54] Mistral-AI, "Codestral: Hello, World!" https://mistral.ai/news/codestral/, 2024.

[55] E. Hartford, "dolphin-mixtral-8x7b," https://erichartford.com/dolphin-25-mixtral-8x7b?source=more_articles_bottom_blogs, 2023.

[56] Mistral-AI, "Mixtral of experts," https://mistral.ai/news/mixtral-of-experts/, 2023.

[57] OpenRouter, "A unified interface for LLMs," https://openrouter.ai/, 2024.

[58] Microsoft, "Introducing Phi-3: Redefining what's possible with SLMs," https://azure.microsoft.com/en-us/blog/introducing-phi-3-redefining-whats-possible-with-slms/, 2024.

[59] Microsoft, "Tiny but mighty: The Phi-3 small language models with big potential," https://news.microsoft.com/source/features/ai/the-phi-3-small-language-models-with-big-potential/, 2024.

[60] Microsoft, "Code it possible," https://azure.microsoft.com/en-gb/, 2024.

[61] A. Dubey *et al.*, "The Llama 3 herd of models," *CoRR*, vol. abs/2407.21783, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2407.21783

[62] Alibaba Cloud, "About Qwen," https://www.alibabacloud.com/en/solutions/generative-ai/qwen?_p_lc=1, 2024.

[63] B. Hui *et al.*, "Qwen2.5-Coder technical report," *CoRR*, vol. abs/2409.12186, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2409.12186

[64] Hugging Face, "Turn AI models into APIs," https://huggingface.co/inference-endpoints/dedicated, 2024.

[65] The Mosaic Research Team, "Introducing DBRX: A new state-of-the-art open llm," https://www.databricks.com/blog/introducing-dbrx-new-state-art-open-llm, Mar. 2024.

[66] databricks, "Databricks is the data and AI company," https://www.databricks.com/company/about-us, Oct. 2024.

[67] A. Lozhkov *et al.*, "StarCoder 2 and the Stack v2: The next generation," *CoRR*, vol. abs/2402.19173, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.19173

[68] BigCode, "The Mission," https://www.bigcode-project.org/docs/about/mission/, 2024.

[69] OpenAI, "Introducing OpenAI o1-preview," https://openai.com/index/introducing-openai-o1-preview/, 2024.

[70] M. Caccia *et al.*, "Language gans falling short," in *ICLR, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=BJgza6VtPB

[71] T. B. Hashimoto, H. Zhang, and P. Liang, "Unifying human and statistical evaluation for natural language generation," in *NAACL-HLT, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. ACL, 2019, pp. 1689–1701. [Online]. Available: https://doi.org/10.18653/v1/n19-1169

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

[73] J. Gawlikowski *et al.*, "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

[74] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965. [Online]. Available: http://www.jstor.org/stable/2333709

[75] N. M. Razali and Y. B. Wah, "Power comparisons of Shapiro-Wilk , Kolmogorov-Smirnov , Lilliefors and Anderson-Darling tests," 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:18639594

[76] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947. [Online]. Available: http://www.jstor.org/stable/2236101

[77] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.

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

[79] GitHub, "Your AI pair programmer," https://github.com/features/copilot, 2023.

[80] B00TK1D, "Copilot.api," 2024, https://github.com/B00TK1D/copilot-api.

[81] F. Beeson, "LLM benchmarks," 12 2024. [Online]. Available: https://github.com/leobeeson/llm_benchmarks

[82] Q. Zheng *et al.*, "CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X," in *KDD, Long Beach, CA, USA, August 6-10, 2023*, A. K. Singh *et al.*, Eds. ACM, 2023, pp. 5673–5684. [Online]. Available: https://doi.org/10.1145/3580305.3599790

[83] T. Y. Zhuo *et al.*, "BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions," in *ICLR, Singapore, April 24-28, 2025*. OpenReview.net, 2025. [Online]. Available: https://openreview.net/forum?id=YrycTjllL0

[84] 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

[85] S. Lu *et al.*, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html

[86] A. M. Dakhel *et al.*, "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

[87] Z. Zeng, Y. Wang, R. Xie, W. Ye, and S. Zhang, "CoderUJB: An executable and unified java benchmark for practical programming scenarios," in *ISSTA, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 124–136. [Online]. Available: https://doi.org/10.1145/3650212.3652115

[88] X. Du *et al.*, "Evaluating large language models in class-level code generation," in *ICSE, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 81:1–81:13. [Online]. Available: https://doi.org/10.1145/3597503.3639219

[89] C. E. Jimenez *et al.*, "SWE-bench: Can language models resolve real-world GitHub issues?" in *ICLR, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=VTF8yNQM66

[90] S. Thakur *et al.*, "Benchmarking large language models for automated verilog RTL code generation," in *DATE, Antwerp, Belgium, April 17-19, 2023*. IEEE, 2023, pp. 1–6. [Online]. Available: https://doi.org/10.23919/DATE56975.2023.10137086

[91] C. AI, "Quick Introduction," 12 2024. [Online]. Available: https://www.deepeval.com/docs/getting-started

[92] Confident AI, "Stay Confident," 12 2024. [Online]. Available: https://www.confident-ai.com/blog

[93] D. N. Manh, "A framework for easily evaluation code generation model," 3 2024. [Online]. Available: https://github.com/FSoft-AI4Code/code-llm-evaluator

[94] M. J. Min *et al.*, "Beyond accuracy: Evaluating self-consistency of code large language models with IdentityChain," in *ICLR, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=caW7LdAALh

[95] Z. Li *et al.*, "CCTEST: Testing and repairing code completion systems," in *ICSE, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1238–1250. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00110

[96] T. Y. Zhuo, "ICE-Score: Instructing large language models to evaluate code," in *EACL, St. Julian's, Malta, March 17-22, 2024*, Y. Graham and M. Purver, Eds. Association for Computational Linguistics, 2024, pp. 2232–2242. [Online]. Available: https://aclanthology.org/2024.findings-eacl.148

[97] W. Tong and T. Zhang, "CodeJudge: Evaluating code generation with large language models," in *EMNLP, Miami, FL, USA, November 12-16, 2024*, Y. Al-Onaizan, M. Bansal, and Y. Chen, Eds. ACL, 2024, pp. 20 032–20 051. [Online]. Available: https://aclanthology.org/2024.emnlp-main.1118

[98] Y. Dong *et al.*, "CodeScore: Evaluating code generation by learning code execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 3, pp. 77:1–77:22, 2025. [Online]. Available: https://doi.org/10.1145/3695991

[99] S. Ren *et al.*, "CodeBLEU: A method for automatic evaluation of code synthesis," *CoRR*, vol. abs/2009.10297, 2020. [Online]. Available: https://arxiv.org/abs/2009.10297

[100] J. Zheng *et al.*, "Beyond correctness: Benchmarking multi-dimensional code generation for large language models," *CoRR*, vol. abs/2407.11470, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2407.11470

[101] M. Allamanis, S. Panthaplackel, and P. Yin, "Unsupervised evaluation of code llms with Round-Trip Correctness," in *ICML, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=YnFuUX08CE

[102] OpenAI, "OpenAI Codex," https://openai.com/index/openai-codex/, 2021.

[103] G. Yang *et al.*, "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
[104] J. Jia *et al.*, "ClawSAT: Towards both robust and accurate code models," in *SANER, Taipa, Macao, March 21-24, 2023*, T. Zhang, X. Xia, and N. Novielli, Eds., 2023. [Online]. Available: https://doi.org/10.1109/SANER56733.2023.00029
[105] H. Zhang *et al.*, "Generating adversarial examples for holding robustness of source code processing models," in *AAAI, IAAI, EAAI, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 1169–1176. [Online]. Available: https://doi.org/10.1609/aaai.v34i01.5469
[106] ——, "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
[107] Z. Tian, J. Chen, and Z. Jin, "Code difference guided adversarial example generation for deep code models," in *ASE 2023, Luxembourg, September 11-15, 2023*, 2023. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00149
[108] H. Zhang *et al.*, "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
[109] Z. Zeng *et al.*, "An extensive study on pre-trained models for program understanding and generation," in *ISSTA, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds., 2022. [Online]. Available: https://doi.org/10.1145/3533767.3534390
[110] 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
[111] S. Srikant *et al.*, "Generating adversarial computer programs using optimized obfuscations," in *ICLR, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=PH5PH9ZO_4
[112] 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
[113] F. Tambon *et al.*, "Bugs in large language models generated code: an empirical study," *Empir. Softw. Eng.*, vol. 30, no. 3, p. 65, 2025. [Online]. Available: https://doi.org/10.1007/s10664-025-10614-4
[114] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. S. Santos, "An empirical study of code smells in transformer-based code generation techniques," in *SCAM, Limassol, Cyprus, October 3, 2022*. IEEE, 2022, pp. 71–82. [Online]. Available: https://doi.org/10.1109/SCAM55253.2022.00014
[115] O. Asare, M. Nagappan, and N. Asokan, "Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code?" *Empir. Softw. Eng.*, vol. 28, no. 6, p. 129, 2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10380-1
[116] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by ChatGPT?" in *SMC, Honolulu, Oahu, HI, USA, October 1-4, 2023*. IEEE, 2023, pp. 2445–2451. [Online]. Available: https://doi.org/10.1109/SMC53992.2023.10394237
[117] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," *Commun. ACM*, vol. 68, no. 2, pp. 96–105, 2025. [Online]. Available: https://doi.org/10.1145/3610721
[118] 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
[119] S. K. Lahiri *et al.*, "Interactive code generation via test-driven user-intent formalization," *CoRR*, vol. abs/2208.05950, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2208.05950
[120] A. Al-Kaswan, M. Izadi, and A. van Deursen, "Traces of memorisation in large language models for code," in *ICSE, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 78:1–78:12. [Online]. Available: https://doi.org/10.1145/3597503.3639133
[121] J. Chen *et al.*, "Reasoning runtime behavior of a program with LLM: how far are we?" in *ICSE, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2025, pp. 1869–1881. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00012
[122] Y. Wan *et al.*, "A & B == B & A: Triggering logical reasoning failures in large language models," *CoRR*, vol. abs/2401.00757, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.00757
[123] S. S. Rajan, E. O. Soremekun, and S. Chattopadhyay, "Knowledge-based consistency testing of large language models," in *EMNLP, Miami, Florida, USA, November 12-16, 2024*, Y. Al-Onaizan, M. Bansal, and Y. Chen, Eds. ACL, 2024, pp. 10 185–10 196. [Online]. Available: https://doi.org/10.18653/v1/2024.findings-emnlp.596
[124] 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
[125] N. Jiang *et al.*, "LeDex: Training llms to better self-debug and explain code," in *NeurIPS, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons *et al.*, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/3ea832724870c700f0a03c665572e2a9-Abstract-Conference.html
[126] L. Yang *et al.*, "On the evaluation of large language models in unit test generation," in *ASE, Sacramento, CA, USA, October 27 - November 1, 2024*, V. Filkov, B. Ray, and M. Zhou, Eds. ACM, 2024, pp. 1607–1619. [Online]. Available: https://doi.org/10.1145/3691620.3695529
[127] J. Wang and Y. Chen, "A review on code generation with LLMs: Application and evaluation," in *MedAI*, 2023, pp. 284–289.
[128] L. Chen *et al.*, "A survey on evaluating large language models in code generation tasks," *CoRR*, vol. abs/2408.16498, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.16498

Shahin Honarvar (Member, IEEE) is a PhD student in the Department of Computing at Imperial College London. His research focuses on the safety, robustness, and evaluation of large language models for code, with an emphasis on systematic testing methodologies. He has introduced a novel benchmarking framework, Turbulence, to assess correctness and robustness in LLMs for code. He received his MSc in Data Analytics from the University of Warwick, where his thesis investigated non-uniform quantisation techniques for graph convolutional networks, demonstrating their efficiency compared to uniform approaches. He obtained his BSc in Computer Science from the University of Leicester, where his dissertation introduced QSharpCheck, the first property-based testing framework for quantum programs in Q#.

Marek Rei is a researcher in Machine Learning and Natural Language Processing. His work investigates new architectures and optimisation methods for language modelling and multimodal representations. He is an Associate Professor of Machine Learning at Imperial College London and a Visiting Researcher at the University of Cambridge. He also serves as an AI Advisor for Locai Labs and Esgrid Technologies, and provides consultancy services through Perception Labs. Previously, he completed a post-doctoral fellowship at the University of Cambridge and worked in the research team at SwiftKey, where he developed experimental technologies for language modelling and natural language processing. He received his PhD from the University of Cambridge.

Alastair F. Donaldson (Member, IEEE) received his Ph.D. from the University of Glasgow. He is a Professor at Imperial College London, where he leads the FastPL Group and serves as the Director of Research for the Department of Computing. His main research interests lie in the intersection of software testing, formal verification, and parallel computing. He was Founder and Director of GraphicsFuzz, a startup company based on his group's research on GPU compiler testing, which was acquired by Google in 2018. After the acquisition, he spent some time working as a Software Engineer with Google before returning full time to Imperial in 2021. He received the 2017 Roger Needham Award in recognition for his research achievements. He is an ACM Senior Member, a Fellow of the British Computer Society, and serves as Editor-in-Chief for ACM Transactions on Programming Languages and Systems.