On LLMs' Internal Representation of Code Correctness

Francisco Ribeiro
francisco.ribeiro@nyu.edu
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates

Claudio Spiess
cvspiess@ucdavis.edu
UC Davis
USA

Prem Devanbu
ptdevanbu@ucdavis.edu
UC Davis
USA

Sarah Nadi
sarah.nadi@nyu.edu
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates

ABSTRACT

Despite the effectiveness of large language models (LLMs) for code generation, they often output incorrect code. One reason is that model output probabilities are often not well-correlated with correctness, and reflect only the final output of the generation process. Inspired by findings that LLMs internally encode concepts like truthfulness, this paper explores if LLMs similarly represent code correctness. Specifically, we identify a correctness representation inside LLMs by contrasting the hidden states between pairs of correct and incorrect code for the same programming tasks. By experimenting on four LLMs, we show that exploiting this extracted correctness representation outperforms standard log-likelihood ranking, as well as verbalized model confidence. Furthermore, we explore how this internal correctness signal can be used to select higher-quality code samples, without requiring test execution. Ultimately, this work demonstrates how leveraging internal representations can enhance code generation systems and make LLMs more reliable, thus improving confidence in automatically generated code.

1 INTRODUCTION

Large language models (LLMs) are increasingly integrated into software development workflows [7], from code completion in IDEs [55, 52] to automated code generation systems [39]. As LLMs continue to improve, we can expect an increasing reliance on generated code in production systems [46].

While developers are expected to review and test generated code before deployment, this ideal scenario does not always occur in practice [83, 48]. Studies have documented cases where incorrect or vulnerable model-generated code has made it into production systems [22, 47]. Real-world incidents such as GitHub Copilot generating a fix for a .NET runtime exception that merely adds superficial bounds checking without addressing the underlying algorithmic issue [23], further demonstrate how LLMs can produce functionally inadequate solutions despite appearing reasonable.

Given this reality, it becomes essential to ensure the correctness of LLM-generated code. In this work, we define *code correctness* as adherence to a given problem specification, which can be verified through test execution—the standard measure used in code generation benchmarks and in practice.

Currently, LLMs typically output the most probable code according to their learned token distributions [70, 89, 16, 80]. However, multiple studies [68, 86, 87] show that the most probable code often fails to meet correctness, as model probabilities are not always well-correlated with correctness.

```
def solution(lst):
  sum_of_odd_elements = 0
  for i in range(1, len(lst), 2):
    if lst[i] % 2 != 0:
      sum_of_odd_elements += lst[i]
  return sum_of_odd_elements
```

(a) Incorrect: iterates odd indices

```
def solution(lst):
  return sum([x for idx, x in enumerate(lst) if idx%2==0 and x%2==1])
```

(b) Correct: sums odd elements at even indices

Figure 1: Comparison of two candidate solutions for the task *"Given a non-empty list of integers, return the sum of all of the odd elements that are in even positions."*

Consider the programming task in Figure 1, where we show an incorrect and correct solution for the task. When we compute the probability of CodeLlama 7B generating each solution (*i.e.*, scoring both implementations under the model), the incorrect implementation actually receives a higher probability. In other words, if we rank the solutions based on the probability of CodeLlama 7B producing this code, the wrong version would be ranked higher. Moreover, even if we were to ask the model for its verbalized confidence in the solutions [74] (*i.e.*, its articulated certainty about each implementation, such as "Very confident" or "Not confident"), this also failed to prioritize the correct implementation. If LLMs cannot reliably differentiate between correct and incorrect code based on what the model explicitly generates or says, then we cannot be confident in the correctness of their outputs.

But what if the true signal of correctness does not lie in what the model shows or tells us, but in what it computes internally? Looking into internal mechanisms for how LLMs interpret different concepts relates to AI interpretability and transparency, an area that has seen a lot of active research in the natural language (NL) [73, 11, 51, 12, 65, 61] and computer vision [27, 18, 56] domains. The growing evidence that LLMs internally encode rich conceptual information motivates us to look beyond the probabilistic favoring of outputs and examine the internal mechanisms at work during code generation. To this end, we turn to representation engineering (RepE) [91], a recent approach in the area of AI transparency. RepE offers a systematic approach for understanding how LLMs encode concepts by analyzing their internal representation spaces. In NL tasks, this approach showed that LLMs possess internal representations of concepts such as truthfulness [91], and importantly, that these internal representations may not always align with the probabilities assigned to the model's generated outputs. Building on the idea that software, like NL, exhibits statistical regularities that models can learn [28], it is reasonable to hypothesize that LLMs may also develop internal representations related to code correctness.

Accordingly, this paper addresses the following question: *do LLMs develop internal representations of code correctness that could provide early signals about whether generated code will pass tests, signals that are not necessarily evident from the final output probabilities?* We investigate this question by examining whether we can capture an internal representation of code correctness that can differentiate correct implementations from incorrect ones. Specifically, we aim to answer the following research questions (RQs):

RQ1 Do LLMs possess an internal representation of code correctness?

RQ2 Can leveraging internal correctness representations lead to more effective correctness ranking?

To address these RQs, we extend RepE beyond its original NL setting and adapt it to a programming language domain. We validate our efforts on *HumanEval* [16] and *BigCodeBench* [90]. Specifically, we compare this technique against standard likelihood-based [16, 41, 68] and reflective methods [74, 42, 88, 68, 33], demonstrating that RepE outperforms these baselines in identifying correct implementations. Furthermore, we introduce a ranking method based on correctness representations that allows for repeated sampling from an LLM and using RepE to get the most promising solution without running tests. In this ranking setup, we also compare to an existing learning-based code ranking method, RankEF [69]. In *HumanEval*, applying our strongest correctness-representation ranking variant improves direct pass@1 by 21.3% on average, compared to 17.7% for the strongest existing baseline (RankEF)—a roughly 20% relative gain. In *BigCodeBench*, our strongest ranking variant delivers a 51.1% average improvement over pass@1 compared to 32.5% for random selection (which for *BigCodeBench* surpasses other baselines). In both benchmarks, we close in on the pass@10 upper bound. To summarize, our contributions are: **1)** Adapting RepE for code, demonstrating its effectiveness across four state-of-the-art LLMs on both *HumanEval* and *BigCodeBench*; **2)** Empirically validating that LLMs encode correctness internally, with representation-based scores outperforming likelihood and reflective metrics in distinguishing correct from incorrect implementations; **3)** a ranking framework that leverages correctness representations that boosts pass@1 by up to 51% without test-time overhead; **4)** A public release of a replication package to facilitate future research:

https://github.com/sanadlab/code-repe

2 BACKGROUND

This section presents the foundational concepts for our work: RepE [91] for analyzing LLM internals, the question-answering framework we use, and confidence metrics for assessing LLM generations.

2.1 Representation Engineering (RepE)

RepE [91] aims to improve LLM transparency by understanding *how* concepts (*e.g.*, truthfulness) and functions (*e.g.*, honesty) are encoded within neural networks. To do this, RepE analyzes the *inner workings* of these models by examining their hidden states to isolate patterns corresponding to specific concepts or functions. RepE is a **top-down** approach [36] that starts with a high-level concept (like *truthfulness* or, in our case, *correctness*) and seeks to find *how* that concept is represented within the model's global representation space. In contrast, **bottom-up** approaches, *e.g.*, mechanistic interpretability [20], typically begin by examining the function of individual neurons or small circuits and build an understanding from there. RepE is like studying a city by looking at its neighborhoods and infrastructure (top-down) rather than reverse-engineering each brick and wire (bottom-up).

RepE has been shown to be effective at detecting and controlling issues like dishonesty and bias [91]. Crucial to RepE is Linear Artificial Tomography (LAT): a framework for analyzing hidden states in an LLM and extracting meaningful representations. At a high level, imagine LAT as a tool to scan the LLM's "brain" and identify patterns related to human-understandable concepts such as truthfulness. By revealing these human-level concepts in LLMs, RepE can be a useful tool for effectively understanding the driving processes behind LLM generations. LAT involves three key steps:

Step 1: Designing the Stimulus and Task The core of this step is to design stimuli and tasks that elicit distinct neural activity within the LLM. To extract a concept, we want to present stimuli that vary in the amount of that concept and inquire about it. For example, when extracting the concept of truthfulness, we would use scenarios that are definitively true or false to represent contrasting levels. Listing 1 shows the template underlying this step:

```
Consider the amount of {concept} in the following:
{stimulus}
The amount of {concept} is␣
```

Listing 1: Eliciting neural activity – prompt template

Consider Listing 2 which shows an instantiation of the template for the concept of truthfulness:

```
Consider the amount of truthfulness in the following:
Question: What is the capital of France?
Answer: Paris.
The amount of truthfulness is␣
```

$\mathbf{P_c}$

Listing 2: Prompt for a correct response (P_c) – truthfulness

In Listing 2, the answer "Paris" is truthful, as it correctly identifies the capital of France. Conversely, Listing 3 shows an instantiation of a prompt with an incorrect response where the answer "Marseille" is untruthful since it incorrectly identifies the capital of France.

```
Consider the amount of truthfulness in the following:
Question: What is the capital of France?
Answer: Marseille.
The amount of truthfulness is␣
```

$\mathbf{P_w}$

Listing 3: Prompt for an incorrect response (P_w) – truthfulness

Step 2: Collecting Neural Activity In Transformer models [75], hidden states exist for all token positions. These hidden states may not all be relevant to the concept at hand. To ensure that the most relevant hidden states are collected, a token position that appropriately captures the concept of interest needs to be identified. For decoder models, RepE demonstrates that the most appropriate positions are those corresponding to the concept's tokens or the last token position [91]. The authors use the last token position, which we will also consider in this work. This is why Listings 2 and 3 may appear incomplete or end abruptly: we only need the hidden states at the last token position, before any further output is generated. To collect these hidden states, a forward pass is performed on the prepared stimuli. Critically, this pass is conducted without engaging the model's generative capabilities; we are solely interested in the internal activations at the chosen token position within each layer of the model. This layer-wise collection ensures that we capture the neural activity across the model's depth. This step only involves separately collecting the hidden states (H) for each of the two types of stimuli: those corresponding to correct responses (H_c) and those corresponding to incorrect responses (H_w). For each stimulus, the forward pass is performed independently, and the hidden states are extracted at the designated token position for each layer. This separation is crucial, because it allows us to contrast the neural activity patterns associated with two instances of opposing extremes—P_c and P_w—in the next step.

Step 3: Extracting the Principal Direction In this final step, we use the collected neural activity to find a direction in the hidden state space that best captures the difference between the two extremes of the concept. The main idea is to compare the hidden states from pairs of opposing examples (H_c and H_w) to see how the model distinguishes between them. For each pair — such as a true versus a false scenario for truthfulness — we subtract one hidden state from the other to get a difference vector (H_{diff}), as shown in Figure 2a. Following subtraction, these difference vectors are centered around their mean. Next, Principal Component Analysis (PCA) [2] is applied to these vectors, layer by layer, to reduce dimensionality and extract the primary direction of variation. Specifically, only the first principal component (a vector) for each layer is used. Figure 2b illustrates this process.

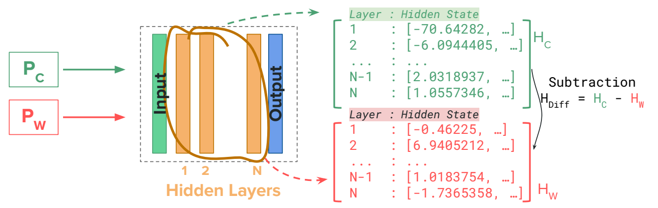
To ensure the extracted direction aligns with the target concept, the original hidden state differences are projected onto these PCA components and correlated with their corresponding concept labels (*e.g.*, correct label = 1, incorrect label = 0). Based on this correlation, LAT assigns a sign (+ or −) to each PCA component. For each pair of examples (P_c and P_w) and for each layer, the projection for the correct label ($T^l_{H_C}$) is used. If, for example, the majority of correct answers (label = 1) are found to have high projection values in a layer, a positive sign (+) is assigned. Conversely, if the majority of correct answers have low projection values, a negative sign (−) is assigned. Figure 2c illustrates the resulting set of PCA vectors and their associated signs, determined layer-wise.

These PCA vectors can then be used to calculate scores for new inputs by projecting the new inputs' hidden states on them. More precisely, the dot product between the hidden states of a new input and a basis vector results in a *representation score*.

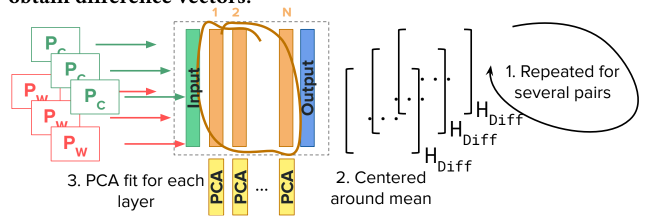
General Pipeline RepE follows a three-phase pipeline: **1) Fitting:** for each layer, collect hidden states from positive and negative stimuli, compute and normalize difference vectors, and fit a PCA to extract the principal direction that best separates the two classes **2) Validation:** project held-out data onto each layer's principal component and select the layer with the highest accuracy at distinguishing positive and negative stimuli **3) Testing:** using the chosen layer, score new instances on a test set and record accuracy.

2.2 Multiple-Choice Question Answering

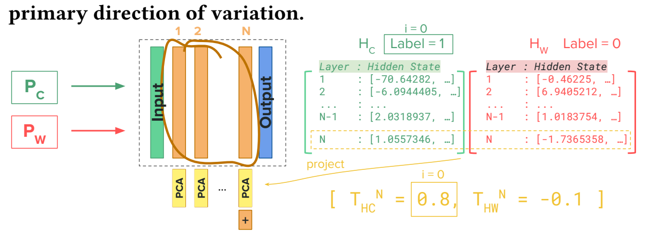
Multiple-choice question answering (MCQA) is a widely used evaluation paradigm for assessing LLM capabilities across diverse domains [35, 62, 63, 71, 50]. At its core, MCQA provides a standardized way to measure how well models can make correct selections when presented with explicit choices. [15, 6, 64, 81]



(a) Subtracting hidden states of correct and incorrect responses to obtain difference vectors.



(b) Centering the difference vectors and applying PCA to extract the primary direction of variation.



(c) Projecting the original hidden states onto the PCA components and assigning signs based on correlation with concept labels.

Figure 2: LAT overview

A typical MCQA setup [19, 43] consists of presenting models with a question alongside multiple candidate answers and requiring them to select the most appropriate response. Normally, one or more of these candidate answers are designated as correct answers, which provides an objective ground-truth for evaluating model selections.

The MCQA setup offers three key advantages: **1)** it transforms complex reasoning into discrete selection tasks, simplifying the evaluation process by reducing the problem to choosing among predefined options [53, 64]; **2)** it provides a controlled comparison environment where various selection methods can be directly evaluated on identical tasks, enabling a systematic assessment of their performance [15, 40]; **3)** it overcomes the subjectivity of generative QA by removing ambiguity in assessing model responses [43, 79].

RepE [91] leveraged this MCQA framework to evaluate the effectiveness of its technique under an NL scenario. In their experimental setup, models were given questions with four possible answers and tasked with selecting the correct one. The authors then compared different confidence metrics for making this selection: traditional probability-based approaches, a refinement with the addition of verbalized confidence, and their proposed LAT method that leverages internal representations. Section 2.3 explains the first two metrics; Section 2.1 already explained the LAT-based metric.

2.3 Confidence Metrics

Confidence metrics can be divided into two categories:

1. Intrinsic: these rely on the model's output probabilities, normally with no or only minimal additional processing:

- *Length-normalized sequence likelihood*: sums the token-level log-probabilities and normalizes by sequence length, avoiding bias toward shorter or longer outputs. The candidate with the highest score is selected. It reflects only how likely the model is to generate a sequence [16, 41, 68].

2. Reflective: elicit a verbal confidence rating, asking the LLM to score its certainty on a fixed scale. These ratings are then combined with token probabilities. We consider two variants:

- *Regular:* Seven verbalized levels ("Very low" to "Very high"), mapped to a numeric scale (evenly spaced from -1 to 1) and weighted by the model's joint probability of generating the tokens corresponding to the verbalized confidence [68, 74, 42, 88].
- *True/False (T/F):* A binary confidence asking whether the candidate is correct (`True` or `False`), scored according to the model's assigned probability to `True` [68, 33].

3 APPLYING REPE TO SOURCE CODE

Adapting RepE to source code introduces unique challenges. In NL, truthfulness is validated according to adherence and consistency with real-world facts [10, 43]. Evaluating this is notoriously complex, as models often succeed by linguistic manipulation rather than achieving genuine natural language understanding (NLU) [10, 9]. In contrast, the typical execution-based evaluation of code is more objective, benefiting from a closed-domain environment where correctness is tied to a specific set of test cases [16, 4]. Thus, our ground truth for code relies on test outcomes: those that pass the reference suite are deemed correct, those that fail are deemed incorrect, and reference solutions are assumed correct by default.

Previous research shows that contrasting two unlabeled examples, *i.e.*, without telling the model what these examples are, can uncover meaningful distinctions [21, 87, 14]. Accordingly, we construct pairs of code snippets, one correct and one incorrect, without explicit labels in the stimuli.

Recall from Section 2.1 that LAT requires stimuli in the form of structured prompts to elicit distinct neural activity about a concept—*correctness* in our case. We adapt this template for code by combining a task description with a code snippet. Listing 4 shows the prompt for an incorrect implementation:

~~~
Task: Given a non-empty list of integers, return the sum of all of
     the odd elements that are in even positions.
Code:
```python
def solution(lst):
    sum_of_odd_elements = 0
    for i in range(1, len(lst), 2):
        if lst[i] % 2 != 0:
            sum_of_odd_elements += lst[i]
    return sum_of_odd_elements
```
~~~

Listing 4: Stimulus for incorrect code

And Listing 5 shows the prompt for a correct implementation.

~~~
Task: Given a non-empty list of integers, return the sum of all of
     the odd elements that are in even positions.
Code:
```python
def solution(lst):
    return sum([x for idx, x in enumerate(lst) if idx%2==0 and x
        %2==1])
```
~~~

Listing 5: Stimulus for correct code

After preparing the stimuli, we extract the model's hidden states at the final token for each layer (Step 2 of LAT, Section 2.1). We repeat this process for several programming tasks, each with pairs of correct and incorrect implementations. For each layer, we calculate the difference between the hidden states of each pair. Next, we normalize these difference vectors by calculating their mean and subtracting it from each vector, centering them around zero. Finally, we apply PCA to these normalized difference vectors to create principal components that capture the greatest variation between correct and incorrect implementations (Step 3, Section 2.1).

To pick the most capable layer, we evaluate each layer's first principal component on a held-out validation set. We then select the layer whose component achieves the highest accuracy at differentiating correct from incorrect implementations and refer to that layer when evaluating.

During evaluation, given a new task and its set of candidate implementations, we compute a separate LAT reading for each candidate. However, this time we use the original LAT template shown in Listing 1, where we explicitly mention the concept of correctness. Listing 6 shows the evaluation prompt for one choice candidate, the incorrect implementation of the task in this case:

~~~
Consider the amount of correctness in the following:
Task: Given a non-empty list of integers, return the sum of all of
     the odd elements that are in even positions.
Code:
```python
def solution(lst):
    sum_of_odd_elements = 0
    for i in range(1, len(lst), 2):
        if lst[i] % 2 != 0:
            sum_of_odd_elements += lst[i]
    return sum_of_odd_elements
```
The amount of correctness is␣
~~~

Listing 6: Evaluation — prompt with a task and a candidate implementation (incorrect implementation in this case)

For each candidate implementation, we construct an equivalent prompt to Listing 6, with the only difference being that the `Code:` block would contain the task implementation in question. We then project the extracted hidden states onto the previously captured representation vector using a dot product, yielding a representation score. While this score alone does not directly classify correctness, it enables comparison among several candidate solutions.

4 RQ1: DO LLMS POSSESS AN INTERNAL REPRESENTATION OF CODE CORRECTNESS?

We now investigate if LLMs maintain an internal representation of code correctness. First, we describe the adaptation to code of the MCQA setup (Section 2.2). Then, we present our results and compare to existing metrics (Section 2.3).

4.1 Setup

We design our experimental setup to extract and evaluate correctness representations using LAT.

4.1.1 Data Preparation. In the original RepE work, effectiveness is evaluated using an MCQA setting where models need to select the correct answer from four choices (Section 2.2). In a code setting, this corresponds to a model selecting the correct code implementation for a given task from a given set of code snippets. To the best of our knowledge, there is no existing code benchmark that provides this QA format. Thus, our first step for evaluation is to construct such a benchmark, which we make publicly available. [5]

We focus on two datasets, *HumanEval* [16] and *BigCodeBench* [90] —summarized in Table 1—with varying levels of difficulty[1]. Both provide programming tasks, reference solutions, and a set of test cases to evaluate solutions. The reference solution serves as the correct choice for the task. For the incorrect choices, we select three additional *incorrect* implementations for each task to match the original RepE methodology. For the evaluation scenario to be challenging and realistic, these incorrect choices should be *plausible attempts* to the given programming task (rather than random code snippets that are obviously wrong). Thus, for each of the two datasets, we leverage generated implementation attempts from established LLMs that failed the task's given tests (*i.e.*, these are incorrect solutions for the task). We use each dataset's leaderboard [45, 90] to select LLMs that are reputable, *i.e.*, trusted by the community and regularly used for research and practical applications. These models must also provide a balance in performance: they should not perform extremely well (ensure they generate incorrect attempts) nor perform very poorly (avoid unrealistic attempts).

For *BigCodeBench*, we select incorrect pre-generated solutions [60] from Llama-3.3-70B, GPT-4o, and Gemini 2.0 Flash. *BigCodeBench*'s more complex tasks generally require larger, resource-intensive models (often via paid APIs) to achieve satisfactory accuracy, so we leverage its publicly available pre-generated outputs. Note that there is only one pre-generated solution for each model in *BigCodeBench*, while we need three incorrect implementations per task. Thus, for *BigCodeBench*, we select tasks on which all three models failed. Out of the 1,140 tasks in *BigCodeBench*, there are 457 tasks on which all three models failed—we refer to these as QA_{BCB}.

For *HumanEval*, we sample the incorrect solutions from Llama-3.2-3B, Gemma-3-1B, and Granite-3.3-2B. *HumanEval*'s problems are easier than *BigCodeBench*'s, so smaller models achieve reasonable accuracy. Consequently, it is also more difficult to get failing attempts. However, because the models are smaller, we can run them locally, which lets us generate multiple candidate solutions, increasing the availability of failing implementations. We use each model to generate 10 implementations for each of the 164 tasks at temperature 1. However, because *HumanEval* is smaller than *BigCodeBench*, we relax the constraint that all three models need to fail. Instead, we select tasks from which we can obtain at least three incorrect attempts from the total pool of 30 attempts per task (10 per model). We end up with 151 tasks with three failing implementations each— we refer to these as QA_{HE}.

4.1.2 *LAT Setup.*

To use LAT, we first have to capture the representation reading using a set of stimuli. We evaluate LAT's generalizability via nested cross-validation (CV) under two sources of data for the stimuli (summarized in Table 2):

[1]Note that while *BigCodeBench* is considered a more complex benchmark, its average C.C. is slightly lower than *HumanEval*'s. This is because C.C. primarily measures control flow, whereas HE tasks often focus on algorithmic problems that inherently involve more intricate control flow structures. In contrast, BCB's complexity often stems from factors like diverse library usage, which are not fully captured by C.C.

Table 1: Summary of *BigCodeBench*, *HumanEval*, and *MBPP+*—number of tasks, average number of test cases, average prompt and solution lengths, and average solution cyclomatic complexity (C.C.).

Benchmark	Nature	# Tasks	Tests (Avg.)		Prompt (Avg.)		Solution (Avg.)		
			#	Cov.	Char	Line	Char.	Line	C.C.
MBPP+ v0.2.0	Beginner-friendly	378	3.1	99%	91.6	1.0	121.1	6.1	2.2
HumanEval	Interview-level	164	7.8	98%	450.6	13.7	180.9	6.8	3.6
BigCodeBench	Real-world challenges	1,140	5.6	99%	663.2	11.7	426.0	10.0	3.1

Table 2: Data splits for different setups. Each *Fit*/*Val* split contains one correct and one incorrect implementation; each *Test* split contains one correct and three incorrect implementations.

Metric	Setup Name	Fit	Val	Test
HumanEval (HE) – QA_{HE} = 151 tasks				
Intrinsic	Standard	-	-	80% of QA_{HE} (121 tasks)
Reflective	Regular	-	-	
	T/F	-	-	
LAT ID	Fit_{HE}	10% of QA_{HE} (15 tasks)	10% of QA_{HE} (15 tasks)	
LAT OOD	Fit_{MBPP+}	25% of *MBPP+* (24 tasks)	25% of *MBPP+* (24 tasks)	
	Fit_{Syn}	25% of Synthetic (5 tasks)	25% of Synthetic (5 tasks)	
BigCodeBench (BCB) – QA_{BCB} = 457 tasks				
Intrinsic	Standard	-	-	80% of QA_{BCB} (367 tasks)
Reflective	Regular	-	-	
	T/F	-	-	
LAT ID	Fit_{BCB}	10% of QA_{BCB} (46 tasks)	10% of QA_{BCB} (46 tasks)	
LAT OOD	Fit_{MBPP+}	25% of *MBPP+* (24 tasks)	25% of *MBPP+* (24 tasks)	
	Fit_{Syn}	25% of Synthetic (5 tasks)	25% of Synthetic (5 tasks)	

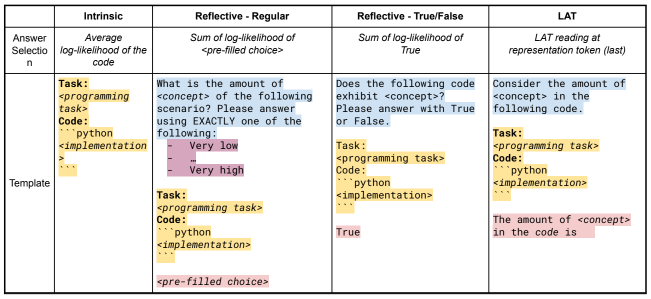
In-distribution (ID). The source of the stimuli data for fitting (*Fit*) and validation (*Val*) is the same as the test data source, but the data points are disjoint. On each benchmark (*BigCodeBench* and *HumanEval*) we run standard 10-fold CV:

- In each fold, we split tasks into **(i)** 10% for fitting ($Fit_{BCB/HE}$), **(ii)** 10% for validation ($Val_{BCB/HE}$), **(iii)** 80% for testing ($Test_{BCB/HE}$).
- For *Fit*/*Val*, we pair each reference solution with one randomly sampled incorrect implementation.
- We fit a layer-wise PCA on *Fit*, select the best layer on *Val*, and measure accuracy on *Test*.
- We report the mean ± std dev of accuracy over the 10 folds.

Out-of-distribution. In this case, the data we use for *Fit* and *Val* comes from a different data source than that used for *Test*. Accordingly, we keep the 10 (outer-)folds defined by the 80% *Test* splits above, but fit and validate on external stimuli:

- We draw the rest of the 20% from two sources: **(1) Synthetic:** prompting Llama-3.2-11B-Vision-Instruct [3] to generate 20 programming tasks, along with a correct and incorrect solution for each. **(2) *MBPP+*:** a code generation benchmark with reference solutions [4, 45] (Table 1). We use GPT-4 failed attempts from EvalPlus [72] as incorrect solutions, resulting in 97 tasks with both correct and incorrect solutions.
- For each source, we run a 4-fold inner CV: **(1)** Each inner-fold allocates 25% of stimuli for fitting, 25% for validation. **(2)** Fit layer-wise PCA on the inner *Fit* split, pick the best layer on inner *Val*, measure accuracy on *Test*.
- For each outer-fold and OOD source, average the 4 inner accuracies (mean ± std dev), then report the mean ± std dev across all 10 outer-folds.

For evaluation purposes, we report two kinds of LAT accuracy: **LAT (*Val*)** uses the layer selected by the validation sets (realistic



	Intrinsic	Reflective - Regular	Reflective - True/False	LAT
Answer Selection	*Average log-likelihood of the code*	*Sum of log-likelihood of <pre-filled choice>*	*Sum of log-likelihood of True*	*LAT reading at representation token (last)*

Template	**Task:**
	<programming task>
	Code:
	```python
	*<implementation>*
	```
		What is the amount of *<concept>* of the following scenario? Please answer using EXACTLY one of the following:
		- Very low
		- …
		- Very high
		
		Task:
		<programming task>
		Code:
		```python
		*<implementation>*
		```
		
		<pre-filled choice>
			Does the following code exhibit <concept>? Please answer with True or False.
			
			Task:
			<programming task>
			Code:
			```python
			<implementation>
			```
			
			True
				Consider the amount of <concept> in the following code.
				
				Task:
				<programming task>
				Code:
				```python
				*<implementation>*
				```
				
				The amount of *<concept>* in the *code* is

Figure 3: Prompt template for correctness evaluation.

performance), while **LAT (*Best*)** selects the optimal layer based on the *Test* set (theoretical upper bound).

4.1.3 Accuracy and Baselines. We assess how well different confidence metrics identify the correct implementation among multiple candidates. We measure this through *accuracy*, defined as the proportion of test tasks for which the correct solution was selected. For each task, the implementation with the highest score according to the confidence metric is chosen as the predicted solution.

As comparison baselines for LAT, we (1) use *Random* to **randomly** select implementations uniformly and (2) use the **intrinsic** and **reflective** confidence metrics described in Section 2.3. Figure 3 shows the prompt templates used to feed tasks and implementations to the model during the evaluation phase (not fitting).

4.1.4 Target Models. Mistral-7B-Instruct-v0.3, Qwen2.5-Coder-7B-Instruct, OpenCoder-8B-Instruct, and CodeLlama-7B-Instruct. From now on, we refer to these using the short forms **Mistral**, **Qwen**, **OpenCoder**, and **CodeLlama**, respectively. Our selection criteria were: (1) **Open-source:** required for extracting internal activations. (2) **Consistency:** Mistral was used in the original RepE work. (3) **Comparable:** models have 7-8B parameters for fair comparison without confounding effects from large size differences. (4) **Code:** Besides Mistral, we used three code-specialized models to evaluate LAT on different domains: general *vs.* code.

4.2 Results

Tables 3 and 4 compare the results of our LAT-based correctness representation extraction against the baseline confidence metrics on *BigCodeBench* and *HumanEval*, respectively. We first explain what we would hope to see in these tables. If LLMs have an inherent notion of correctness, then the verbalized confidence metrics (*i.e.*, reflective) should have an accuracy higher than standard purely probabilistic metrics (*i.e.*, intrinsic). However, even reflective metrics rely on a form of probability rather than reflect any inherent internal characteristics of the model. Therefore, if LAT can successfully capture internal representations of correctness, then we expect that LAT has an accuracy even higher than reflective metrics.

4.2.1 Overall Performance. We observe three patterns in Tables 3 and 4. First, the intrinsic (standard) metric on *BigCodeBench* hovers around 5–6% for all models, whereas on *HumanEval* it ranges from 21%-37%, reflecting the latter's lower difficulty. Second, reflective variants are inconsistent. For example, on *BigCodeBench*, the true/false variant outperforms the regular variant for Qwen (41.1% vs. 38.0%) but underperforms for Mistral (23.7% vs. 28.9%). In contrast, on *HumanEval*, Mistral's reflective regular accuracy (40.6%) exceeds its true/false accuracy (15.3%), while Qwen scores around 7% on both. Third and most importantly, LAT consistently outperforms all baselines. On *BigCodeBench*, it improves accuracy by +13.3 to +29.0 pp over the best reflective accuracy; on *HumanEval*, gains range from +8.7 to +26.2 pp over the strongest baseline. These results show that capturing correctness from neural activity yields a more stable correctness signal than intrinsic and reflective confidence. Throughout the paper, we use Generalized Estimating Equations (GEE) logistic regression with Benjamini-Hochberg correction ($p < 0.05$) for significance testing.

4.2.2 Out-of-Distribution Generalization. We now analyze how well correctness representations transfer across datasets. On *BigCodeBench*, fitting on *MBPP+* yields between 25.8% to 47.8% accuracy with 3 out of 4 models above random but below in-distribution LAT (50.3%–56.3%). Synthetic fitting is erratic: two models drop below random (Qwen at 12.2%, CodeLlama at 22.2%), while Mistral and OpenCoder exceed it. On *HumanEval*, *MBPP+* fitting stays above random (32.2%–39.9%), but below in-distribution LAT (41.4–60.7%). Synthetic fitting can match or exceed in-distribution LAT for some models (e.g., Qwen at 63.0% vs. 60.7%; OpenCoder at 47.8% vs. 44.0%), yet fails for others. A different, yet related, observation is the standard deviation's pattern. In-distribution fitting shows higher variance (e.g., ±10.6% under Mistral in *BigCodeBench*) compared to the lower variance observed in out-of-distribution fitting (e.g., ±0.8% under Mistral in *BigCodeBench* using Fit_{MBPP+}). This suggests a specialization versus generalization trade-off: fitting on in-distribution data achieves higher accuracy by specializing in data nuances (low bias), but this specialization leads to higher instability (high variance) when those nuances are missed across folds. On the other hand, fitting on out-of-distribution data captures more general features, despite leading to lower accuracy.

4.2.3 Model-Specific Insights. The magnitude of LAT's improvement compared to each model's strongest baseline reveals model differences. On *BigCodeBench*, OpenCoder gains the most (+28.8 pp from 27.5% to 56.3%), followed by CodeLlama (+22.8 pp), Mistral (+22.2 pp), and Qwen (+13.3 pp). On *HumanEval*, Qwen shows the largest increase (+26.2 pp from 36.8% to 63.0%), then OpenCoder (+11.3 pp), CodeLlama (+10.8 pp), and Mistral (+8.7 pp). These patterns suggest that while all models embed a correctness signal, its strength varies by training regime and task complexity. This is consistent with research highlighting that LLMs' ability to generalize, rather than merely memorize solutions, varies across models and different types of evolved coding tasks. [17].

4.2.4 Validation vs. Best Layer Selection. We compare LAT (*Val*), which selects layers via held-out validation, against LAT (*Best*), the theoretical upper bound. Across models and datasets, the gap rarely exceeds 7pp for in-distribution fitting. On *BigCodeBench*, Qwen scores 54.4% (*Val*) *vs.* 57.1% (*Best*) and, on *HumanEval*, Mistral scores 49.3% *vs.* 53.8%. The greatest gap is OpenCoder with a difference of 12.5 pp between 44% (*Val*) and 56.5% (*Best*). Still, the proximity indicates that validation-driven layer choice reliably approximates the optimal layer without test-set leakage.

Table 3: LAT vs. baseline confidence metrics on *$Test_{BCB}$* — * indicates statistically significant improvement over all baselines ($p < 0.05$).

Model	Random	Intrinsic	Reflective		LAT (Val) – Ours			LAT (Best)		
		Standard	Regular	True/False	Fit_{BCB}	Fit_{MBPP+}	Fit_{Syn}	Fit_{BCB}	Fit_{MBPP+}	Fit_{Syn}
Mistral	27.5%	5.7% ± 0.4	28.9% ± 0.8	23.7% ± 0.0	**51.1% ± 10.6***	47.8% ± 0.8*	33.0% ± 0.1*	57.9% ± 4.5	59.3% ± 0.9	48.4% ± 0.5
Qwen		5.0% ± 0.3	38.0% ± 0.7	41.1% ± 0.8	**54.4% ± 3.8***	40.2% ± 0.9	12.2% ± 0.8	57.1% ± 2.7	55.7% ± 0.7	39.4% ± 0.5
CodeLlama		4.8% ± 0.4	24.7% ± 0.4	23.7% ± 0.0	**50.3% ± 6.2***	25.8% ± 0.3	22.2% ± 0.7	55.0% ± 3.0	48.2% ± 0.9	39.0% ± 0.7
OpenCoder		6.4% ± 0.4	24.5% ± 1.3	27.3% ± 0.5	**56.3% ± 5.1***	29.5% ± 0.3	30.5% ± 0.7	60.0% ± 2.8	44.2% ± 0.7	44.2% ± 1.2

Table 4: LAT vs. baseline confidence metrics on *$Test_{HE}$* — * indicates statistically significant improvement over all baselines ($p < 0.05$).

Model	Random	Intrinsic	Reflective		LAT (Val) – Ours			LAT (Best)		
		Standard	Regular	True/False	Fit_{HE}	Fit_{MBPP+}	Fit_{Syn}	Fit_{HE}	Fit_{MBPP+}	Fit_{Syn}
Mistral	30.6%	21.3% ± 3.3	40.6% ± 2.7	15.3% ± 2.7	**49.3% ± 7.0***	39.9% ± 2.2	38.1% ± 1.4	53.8% ± 7.2	58.4% ± 1.2	47.6% ± 1.7
Qwen		36.8% ± 3.1	7.0% ± 1.1	6.7% ± 0.9	60.7% ± 8.3*	39.6% ± 1.3	**63.0% ± 2.0***	67.8% ± 4.6	65.6% ± 1.8	65.2% ± 1.9
CodeLlama		21.3% ± 2.5	28.9% ± 2.1	26.5% ± 0.0	**41.4% ± 6.4***	32.2% ± 1.0	25.5% ± 1.3	52.7% ± 4.5	49.6% ± 2.1	46.7% ± 1.5
OpenCoder		36.5% ± 3.9	31.0% ± 2.5	9.9% ± 1.6	44.0% ± 8.3*	35.1% ± 1.8	**47.8% ± 3.0***	56.5% ± 3.1	52.4% ± 2.8	56.6% ± 3.9

RQ1: Key takeaway

LAT successfully extracts meaningful correctness representations from neural activations, significantly outperforming random chance and surpassing intrinsic and reflective confidence metrics across both benchmarks and all models. On *BigCodeBench*, LAT improves accuracy by +13.3 to +28.8 pp over the best baseline; on *HumanEval*, gains range from +8.7 to +26.2 pp.

5 RQ2: CAN LEVERAGING INTERNAL CORRECTNESS REPRESENTATIONS LEAD TO MORE EFFECTIVE CORRECTNESS RANKING?

Based on RQ1, we know that LAT can capture an internal notion of correctness that distinguishes correct from incorrect solutions. In RQ2, we explore using LAT to rank multiple generated solutions by estimated correctness. Traditionally, developers prompt LLMs for code, then evaluate outputs manually or with tests, often requiring several attempts. LAT can shortcut this by ranking candidates so the most correct are near the top, making it easier to select a correct implementation from a small set.

5.1 Setup

To evaluate LAT's ranking ability, we test whether it can effectively order multiple generations by correctness.

5.1.1 Data Preparation. We focus on both *HumanEval* and *BigCodeBench*, using the same tasks and reference solutions as RQ1 for fitting and validation in both in-distribution (Fit_{HE}/Val_{HE} and Fit_{BCB}/Val_{BCB}) and out-of-distribution (Fit_{syn}/Val_{syn} and Fit_{MBPP+}/Val_{MBPP+}). For testing, we use the same 121 tasks from $Test_{HE}$ and 367 tasks from $Test_{BCB}$ established in RQ1. For each task and model, we generate 10 diverse code samples (temperature = 1.0), yielding 1,210 samples per model for *HumanEval* and 3,670 for *BigCodeBench*. The goal is to assess each model's ability to rank its own outputs, similar to self-correction.

5.1.2 LAT Setup. We use the fit, validation, and test splits from the first fold of RQ1 to ensure consistency across models and tasks while simplifying the evaluation. The ranking process involves fitting correctness representations on the designated *Fit* set, selecting the optimal layer using the *Val* set, and applying the learned vector to score and rank the generated implementations in the *Test* set.

5.1.3 Accuracy and Baselines. We evaluate our ranking approach using pass@rank-k metric, which measures the percentage of programming problems where at least one functionally correct solution (verified by test execution) appears within the top k positions. Formally, for a set of problems P and rank threshold k:

$$\text{pass@rank-}k = \frac{1}{|P|} \sum_{p \in P} \mathbb{I}[\exists i \leq k : \text{correct}(\text{rank}_i(p))]$$

where $\text{rank}_i(p)$ denotes the i-th ranked implementation for problem p, $\text{correct}(\cdot)$ indicates whether the implementation passes all unit tests, and $\mathbb{I}[\cdot]$ is the indicator function that returns 1 if the condition is true and 0 otherwise.

We report pass@rank-k for $k \in \{1, \ldots, 5\}$ on *HumanEval* and *BigCodeBench* to quantify the trade-off between the number of candidates and the probability that at least one in the top k is correct. Note that unit tests are only used to evaluate the ranked outputs, not during the ranking process. In addition to Random, Intrinsic, and Reflective metrics, we compare against these baselines:

- **pass@1 (baseline):** Fraction of problems where a single implementation passes all tests, representing single-attempt performance. We use temperature = 0.2 to balance greedy decoding (temperature = 0, which always selects the most likely token at each step) and diversity, helping avoid local minima while keeping solutions focused—in line with previous research [16]. Note that this pass@1 baseline measures success using a single generation ($N = 1$), while pass@rank-1 evaluates the top-ranked solution selected from the set of 10 diverse generations ($N = 10$).
- **pass@10 (ceiling):** Fraction of problems where at least one of 10 implementations (same set used for pass@rank-k) passes all tests ($N = 10$). We sample with temperature = 1, as this value should raise with the number of samples [41], to encourage diversity and maximize the chance of at least one correct solution, reflecting exploration rather than single-attempt success.

- **RankEF:** We also compare to RankEF [69], a multi-task learning approach leveraging execution feedback to improve code ranking. During training, RankEF integrates classification labels and execution feedback to better distinguish correct from incorrect candidates. We report pass@rank-k for RankEF on the same *BigCodeBench* and *HumanEval* test sets. Since no ready-to-use RankEF ranker was available, we reproduced the training process from the RankEF GitHub repository. Following the original paper, we used CodeT5+ [77] as the base model, further trained on 5,000 APPS [25] training tasks. For each task, we generated 100 samples with CodeT5+ and collected execution feedback for training. The model was trained using the described multi-task learning framework with hard parameter sharing. The authors report that training a CodeT5+-based RankEF model this way generalizes well and can rank generations from other models and datasets. Thus, we train a single RankEF model and use it to rank generations from all models in our experiments, where each model produces 10 candidate implementations per task. To validate our reproduction, we compared our results to the authors' reported scores on CodeLlama, the LLM both works have in common. Our reproduced ranker achieves slightly higher accuracy, likely because our evaluation ranks 10 candidate implementations per task, while the original work ranks 100.

5.2 Results

Figures 4-5 and Tables 5-6 summarize the results of RQ2.

5.2.1 Overall Performance. Our LAT-based ranking approach demonstrates significant improvements over the pass@1 baseline across both benchmarks, particularly as more candidates are considered ($k > 1$). While simpler rankers like `Random` or `Intrinsic` show some gains, our LAT-based fittings (Fit_{HE}/Fit_{BCB}, Fit_{MBPP+}, Fit_{Syn}) consistently provide a more robust path to higher accuracy. On the *HumanEval* benchmark, this benefit is often immediate. For instance, Mistral's accuracy with Fit_{HE} is 41.3% at Rank-1, well above its 34.7% pass@1 baseline and the 38.0% achieved by `Random` selection—Figure 4. On the more challenging *BigCodeBench* benchmark, where all models have lower baseline accuracies, the advantage of LAT-based ranking becomes even clearer at higher ranks, consistently outperforming the baselines and narrowing the gap to the pass@10 performance ceiling (See Figure 5). Compared to RankEF, an approach that needs a full fine-tune procedure on execution feedback, our LAT-based technique achieves competitive or superior accuracy without requiring expensive training or test execution, demonstrating the value of our lightweight ranking (see Section 6.2). Numerically, LAT shows superior or comparable accuracy to all baselines, including RankEF, across most settings. Only a few comparisons reached statistical significance.

5.2.2 Performance on HumanEval. Figure 4 details the results for *HumanEval*. The four panels highlight that while different models benefit from ranking to varying degrees, the LAT-based fittings are consistently effective. For **Mistral**, the Fit_{HE} and Fit_{Syn} fittings provide the strongest performance, starting at 41.3% and reaching 64.5% and 62.0% respectively at $k = 5$, significantly outperforming all alternative methods and nearing the 67.8% ceiling. Notably, both LAT fittings surpass RankEF, which in comparison achieved 35.5% at $k = 1$ and 59.5% at $k = 5$. **Qwen**, with a very high 85.1% pass@1 baseline, still benefits from LAT ranking. Its Fit_{HE} and Fit_{Syn} fittings match the baseline at $k = 1$ and climb to 95.0% and 93.4% respectively by $k = 5$, almost reaching the 95.9% ceiling. This shows that even for highly capable models, LAT can effectively identify the best among several good candidates. **CodeLlama** presents an interesting case where the `Intrinsic` metric is surprisingly effective, outperforming other methods at several ranks. This is likely due to the self-ranking context, where the model's generation probabilities are a strong ranking method, even though it depends on how well a particular model's output probabilities reflect actual correctness. However, our LAT-based fittings still show competitive performance, with Fit_{HE} and Fit_{Syn} reaching 57.0% at $k = 5$. Here, RankEF (34.7% at $k = 1$, 55.4% at $k = 5$) also shows competitive performance, though our LAT fittings remain higher. For **OpenCoder**, Fit_{Syn} and Fit_{MBPP+} provide the most substantial gains over its 78.5% baseline, reaching 88.4% and 86.0% at $k = 3$, respectively, demonstrating the power of out-of-distribution signals. Across all models, the `Reflective` ranking methods consistently underperform, often doing worse than `Random`. This suggests that simple self-correction signals are unreliable, reinforcing the need for the more sophisticated capturing of correctness provided by LAT.

For easier reference, Table 5 summarizes the pass@rank-1 accuracy for all models and ranking methods on the *HumanEval* benchmark. LAT-based fittings achieve the best rank-1 performance for Mistral (41.3%), Qwen (85.1%), and OpenCoder (80.2%), while CodeLlama benefits most from its Intrinsic metric (39.7%). Notably, our approach outperforms RankEF across all models.

Table 5: Comparison of Random, Intrinsic, Reflective, LAT, and RankEF pass@rank-1 accuracy (%) on *HumanEval* using *Test$_{HE}$*.

Model	Random	Intrinsic	Reflective		LAT (Val) – Ours			RankEF
		Standard	Regular	T/F	Fit_{HE}	Fit_{MBPP}	Fit_{Syn}	
Mistral	38.0%	33.9%	26.4%	33.9%	**41.3%**	35.5%	**41.3%**	35.5%
Qwen	64.5%	80.2%	24.8%	26.4%	**85.1%**	28.1%	**85.1%**	81.0%
CodeLlama	35.5%	**39.7%**	31.4%	30.6%	35.5%	30.6%	31.4%	34.7%
OpenCoder	76.9%	76.0%	64.5%	63.6%	63.6%	70.2%	**80.2%**	75.2%

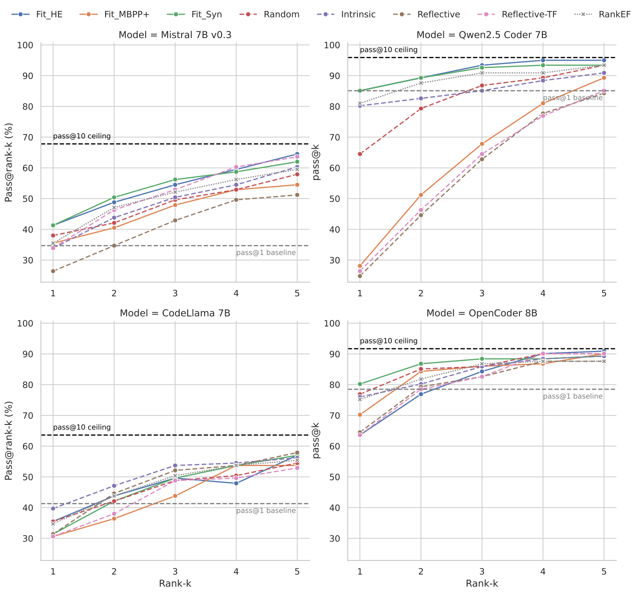


Figure 4: Accuracy of different ranking methods for *HumanEval* compared to the pass@1 baseline and pass@10 ceiling.

Table 6: Comparison of Random, Intrinsic, Reflective, LAT, and RankEF pass@rank-1 accuracy (%) on *BigCodeBench* using $Test_{BCB}$.

Model	Random	Intrinsic	Reflective		LAT (Val) – Ours			RankEF
		Standard	Regular	T/F	Fit_{BCB}	Fit_{MBPP}	Fit_{Syn}	
Mistral	3.5%	3.0%	3.5%	2.5%	1.4%	3.8%	**5.7%**	4.1%
Qwen	5.7%	5.2%	3.0%	4.6%	4.9%	**7.1%**	5.2%	6.0%
CodeLlama	1.6%	2.7%	2.5%	1.9%	0.8%	1.6%	2.7%	**3.3%**
OpenCoder	7.6%	6.8%	6.8%	8.2%	6.8%	8.2%	**8.4%**	8.2%

5.2.3 Performance on BigCodeBench. Figure 5 shows the results on the more difficult *BigCodeBench* benchmark. For **Mistral**, the Fit_{Syn} fitting is the most effective, improving from 5.7% at $k = 1$ to 10.4% at $k = 5$, consistently staying above all other methods. Fit_{Syn} already outperforms RankEF at rank-1 (5.7% vs. 4.1%) and maintains a clear lead across all ranks, reaching 10.4% at rank-5 compared to RankEF's 8.2%. In contrast, we find that by rank-2, RankEF does not outperform the Random baseline. **Qwen** sees all three LAT-based fittings surpass the baselines and alternative methods by $k = 2$, with Fit_{BCB} and Fit_{Syn} reaching 16.9% and 16.6% at $k = 5$. **CodeLlama** struggles on this benchmark, only able to surpass the pass@1 baseline and non-LAT methods by rank $k = 3$. Here, RankEF performs best at $k = 1$ (3.3%), suggesting that execution feedback may be particularly valuable when models struggle with task difficulty. However, despite the lower start at rank-1 (2.7%) and tying RankEF at rank-2, Fit_{Syn} provides the highest accuracy for the remaining ranks, achieving 9.5% at $k = 5$. **OpenCoder** shows significant gains with out-of-distribution fittings. The Fit_{Syn} curve rises sharply with Fit_{MBPP+} catching on by rank $k = 4$, eventually reaching 21.0% and 20.2% at $k = 5$, surpassing the 11.4% baseline and demonstrating that LAT-based ranking is more effective than non-LAT methods. On this benchmark, the `Intrinsic` metric is far less effective than on *HumanEval*, highlighting its unreliability as a general strategy. In contrast, the LAT-based fittings, particularly Fit_{Syn}, prove to be a consistently effective method for improving performance across all models on *BigCodeBench*.

Table 6 summarizes the pass@rank-1 on the more challenging *BigCodeBench* benchmark. Overall, LAT's Fit_{Syn} fitting achieves top performance for Mistral (5.7%) and OpenCoder (8.4%) while LAT's Fit_{MBPP} achieves the highest score for Qwen (7.1%). With the exception of CodeLlama, LAT-based ranking achieves higher accuracy than RankEF for all models at rank-1.

5.2.4 Out-of-distribution Generalization. A key strength of our approach is its ability to generalize. On *HumanEval*, although the in-distribution Fit_{HE} fitting often leads, the out-of-distribution Fit_{Syn} fitting matches or exceeds it at low k for Mistral and Qwen. This contrasts with our MCQA setup (RQ1), where incorrect options came from other models and out-of-distribution fittings underperformed. In RQ2, all candidates are self-generated, so a model's internal correctness signal may align more consistently. For OpenCoder, Fit_{Syn} achieves 80.2% at $k = 1$, versus 63.6% for Fit_{HE}. Notably, the out-of-distribution Fit_{Syn}'s generalization outperforms RankEF for all ranks despite RankEF's more specialized training, highlighting the transferability of LAT correctness patterns.

On *BigCodeBench*, out-of-distribution fittings (Fit_{Syn}, Fit_{MBPP+}) frequently outperform Fit_{BCB}. For Mistral and OpenCoder, Fit_{Syn} is top across almost all ranks. This robust transfer suggests a single well-fitted LAT can rank solutions on new tasks without any in-distribution data and often better than an in-distribution fit.

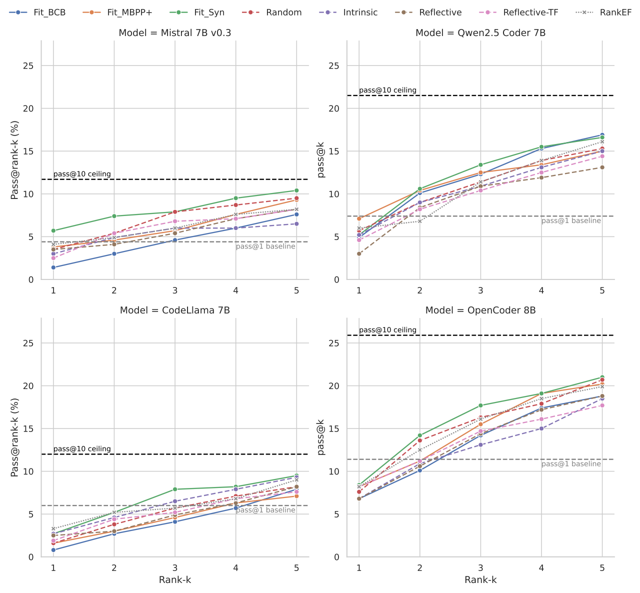


Figure 5: Accuracy of different ranking methods for *BigCodeBench* compared to the pass@1 baseline and pass@10 ceiling.

5.2.5 Rank Threshold Analysis. Across both benchmarks, accuracy consistently improves as the rank threshold k increases from 1 to 5, demonstrating that the LAT-based method effectively favors correct solutions from the candidate pool. While the top-ranked solution ($k = 1$) is not always correct, expanding consideration to the top 3 or 5 candidates captures most of the achievable performance, rapidly approaching the pass@10 ceiling. For example, on *HumanEval*, Mistral improves from its 34.7% pass@1 baseline to 64.5% at $k = 5$ with Fit_{HE}, closing most of the gap to its 67.8% ceiling. A similar pattern holds on *BigCodeBench*, where OpenCoder jumps from an 11.4% baseline to 21.0% at $k = 5$ with Fit_{Syn}. Comparing growth trajectories between LAT and RankEF reveals model-dependent patterns. For Mistral, LAT's Fit_{Syn} starts higher at rank-1 (5.7% vs. RankEF's 4.1%) and maintains its lead through rank-5 (10.4% vs. 8.2%). For Qwen, Fit_{BCB} and Fit_{Syn} start lower at rank-1 (4.9% and 5.2% vs. RankEF's 6.0%) but overtake RankEF by rank-2 (10.1% and 10.6% vs. RankEF's 6.8%), showing steeper improvement. For CodeLlama, LAT's Fit_{Syn} ties RankEF at rank-2 (5.2%) and then surpasses it at rank-3 onwards, reaching 9.5% at rank-5 versus RankEF's 9.0%. For OpenCoder, LAT's Fit_{Syn} shows competitive or superior performance across all ranks. Overall, LAT fittings demonstrate a pattern of rapid early improvement (rank-1 to rank-3), effectively catching up to and surpassing RankEF even when off to a slightly lower start for CodeLlama.

5.2.6 Model-Specific Insights. On *HumanEval*, models with lower baselines like Mistral see the largest relative gains (an 85.9% relative improvement from pass@1 to $k = 5$ with Fit_{HE}). For high-performing models like Qwen, LAT is still able to push it towards its ceiling. RankEF also shows consistent improvements over the pass@1 baseline, but LAT's best fittings achieve comparable or superior gains. For Mistral, LAT's Fit_{HE} achieves +29.8 points over pass@1 (vs. RankEF's +24.8), while for OpenCoder, Fit_{HE} achieves +12.4 (vs. RankEF's +9.1). For CodeLlama, LAT's gains with Fit_{Syn} are +15.7 points, compared to RankEF with +14.1 points.

On the more challenging *BigCodeBench* benchmark, these model-dependent patterns highlight the robustness of our ranking approach. Qwen improves its accuracy by 9.5% absolute points (from 7.4% to 16.9% at $k = 5$ with Fit_{BCB}), while OpenCoder achieves a 9.6% absolute gain (from 11.4% to 21.0% with Fit_{Syn}). The consistent success of the Fit_{Syn} fitting, especially on *BigCodeBench*, suggests that training on a diverse, synthetic dataset equips the LAT model with a strong and generalizable understanding of code correctness.

RQ2: Key takeaway

LAT-based ranking outperforms pass@1 baseline, intrinsic, and reflective methods on *HumanEval* and *BigCodeBench* across all models. LAT showed significant gains by rank $k = 3$ and further closed most of the gap to the pass@10 ceiling by rank $k = 5$. Except for CodeLlama, LAT achieved higher accuracy than RankEF for all models at rank-1. A key finding is the effectiveness of out-of-distribution fittings (Fit_{Syn} and Fit_{MBPP+}), which highlights LAT's potential to reduce costly test executions and generalize to new tasks without requiring in-distribution data.

6 DISCUSSION

We now discuss the implications of our findings in terms of practical utility, differences between our approach and training-based ones, as well as directions for future research.

6.1 Practical Utility

The ability to capture internal correctness representations has practical applications for developers. Our ranking method in RQ2 shows that these representations can be used to select promising code solutions without running tests [31]. In essence, when an LLM generates multiple code solutions, our method can filter out incorrect candidates and highlight promising ones. Thus, developers can focus their verification efforts on a smaller set of top-ranked solutions, rather than having to examine every candidate. It could even be used "behind the scenes" where the LLM-based generation system internally generates multiple candidates and returns only the highest ranked candidate to the developer, effectively increasing the accuracy of the single generation the developer sees. Beyond this, our technique can be integrated into the software development life cycle. Since our approach captures relative correctness, it is best suited for comparing code changes. For example, in a CI/CD pipeline, it can flag changes where the new code appears less correct than the previous version, and prioritize test cases based on which code is deemed less correct. [82] In IDEs, it can provide confidence scores for code suggestions relative to the current implementation, or compare multiple suggestions against each other. [58].

6.2 Fitted Representations vs. Trained Rankers

Our accuracy comparisons with RankEF showed superior or comparable results. We discuss some additional practical advantages that our LAT-based ranking approach offers over trained rankers.
Setup cost and data efficiency. RankEF requires training a separate ranker model on thousands of code samples, which is memory- and compute-intensive. Moreover, gathering execution feedback for RankEF's training dataset requires running test suites on thousands of generated samples, adding substantial overhead to the setup. Reproducing RankEF involved 72 GPU-hours for sample generation, 10 CPU-hours for execution feedback collection, and 90 GPU-hours for model training on an NVIDIA A100 (peak memory: 60GB). Our technique, by contrast, is *fitted* rather than *trained*—requiring only PCA fitting on a small dataset. Average fitting time (PCA + Validation) is 3.75 seconds. This is computationally inexpensive and data-efficient, which is especially evident from our synthetic fitting set of only of 5 triplets of tasks with correct and incorrect implementations that fits in around 1 second.
Flexibility and adaptability. The lightweight fitting process makes our approach more adaptable. Should correctness criteria evolve (e.g., from 'correctness' to 'readability' or 'adherence to new style guidelines'), our technique can be re-fitted on a small set of relevant examples. Training-heavy approaches like RankEF would require expensive re-training with large datasets.
Inference and generalizability trade-offs. Our technique requires extracting hidden states during generation, resulting in higher per-task inference time (0.399s vs. RankEF's 0.202s). However, this ~0.2s overhead per task is negligible compared to the amortized setup cost: RankEF's 172 GPU+CPU-hours would need to rank roughly 3 million tasks to break even on compute cost alone. For practical scenarios, LAT's lightweight setup far outweighs its inference overhead. Additionally, while our approach is model-specific (representations captured from one LLM cannot be applied to another's hidden states), a model and its fitting can rank code solutions from any source. RQ1 demonstrates this, where a choice was made on candidates from different sources: reference solution and other LLMs (see Section 4.1.1).

6.3 The Nature of Internal Correctness

Our findings strongly indicate that LLMs develop an internal representation of code correctness—in line with similar observations in reasoning models [84]. More concretely, the core finding is that correctness representations extracted via RepE outperform baseline confidence metrics [66]. In RQ1, LAT improves accuracy over reflective baselines on both *BigCodeBench* and *HumanEval*. In RQ2, LAT-based ranking improves direct pass@1 performance and approaches the pass@10 ceiling. Overall, this indicates that a correctness signal captured from the model's internal states provides a more stable indicator of correctness than the model's output probabilities, which are poorly calibrated with correctness [68]. In this work, we showed that this signal *exists and can be leveraged* but not *why or how* it exists. Is the model learning a deep semantic understanding of the code's logic? Or is it learning more superficial pattern recognition based on syntactic structures and token sequences that correlate with correctness in the training data? It would be interesting to explore this distinction in future work.

6.4 Beyond Functional Correctness

In this paper, we focused on functional correctness, which can be objectively verified using tests. Since the represented concept depends on the contrasting stimuli presented to the LLM, future work could explore whether LAT can also capture other non-functional properties such as maintainability or efficiency [76, 1].

7 THREATS TO VALIDITY

Construct Validity. We use passing unit tests as a proxy for correctness, while tests may fall short when it comes to non-functional aspects (*e.g.*, performance and readability). There are two factors that can affect the correctness representation captured by LAT: the stimuli used and the model layer chosen to capture the correctness signal. Since our primary goal in this paper is to rigorously explore if RepE can be used for code correctness, we carefully adhered to the same stimulus setup and layer selection used in the original work to ensure methodological consistency. Specifically, we employ a robust selection method that verifies which layer has the highest accuracy on a validation split. We also evaluate on two different benchmarks (*HumanEval* and *BigCodeBench*), combining ID and OOD stimuli, and doing CV.

Internal Validity. Prompt formatting choices or verbalized confidence levels could influence our observations. We reduce prompt bias by using prompt designs that are consistent with prior work. The sourcing of candidate solutions also differs between RQ1—where incorrect options come from other LLMs—and RQ2—where variants are self-generated—which may partly explain divergent out-of-distribution behavior. Evaluating both settings exposes how candidate origin impacts performance. Also, collecting incorrect implementations from existing LLMs could introduce bias if those models share training data or produce similar errors to the target model. We address this by collecting failing solutions from multiple reputable models and using an established evaluation harness [59] and benchmarks (*HumanEval* and *BigCodeBench*)

External Validity. We evaluate the accuracy of four 7-8B parameter LLMs on two Python benchmarks (*HumanEval* and *BigCodeBench*). To improve generalization, we include both general and code models, evaluate on benchmarks with differing degrees of complexity, and test LAT on OOD data (synthetic and *MBPP+*). However, the observed LAT effectiveness and ranking improvements may not hold for larger or smaller models, different architectures, or other programming languages. Our single-function evaluation does not cover some aspects of real-world code, *e.g.*, multi-file projects or integration tests, potentially found in repository-level benchmarks such as SWE-Bench [32]. We see our work not as a direct tool for such settings, but as a component within an LLM-based agent's strategy. We plan to explore such settings as future work.

8 RELATED WORK

LLMs have improved developer productivity and efficiency [29], but concerns remain about the quality of generated code, including incorrectness [78] and security vulnerabilities [57].

Recent work has explored training neural rankers to select code solutions without test execution. CodeRanker [31] uses fault-aware classification to predict program correctness. RankEF [69] improves upon this with multi-task learning on execution feedback, enabling it to learn failure causes.

Many studies focus on estimating LLM confidence and improving calibration. Traditional methods often use output probabilities [26], few-shot prompting [34], or verbalized confidence [74]. However, neural networks are often miscalibrated [24], and standard metrics may not reliably reflect correctness in code generation [68, 54], with added concerns in cyber threat intelligence [49].

To address these issues, recent work explores more sophisticated confidence metrics. One approach infers confidence from the consistency of multiple generated outputs [37], linking program similarity to correctness and reducing errors. Similar consistency-based methods have been applied to NL tasks [85].

LLM internal states have also been leveraged for confidence estimation, such as through contrastive learning across layers [8] or by weighting tokens using attention values [44], reinforcing the potential of internal signals for output quality.

AI interpretability and transparency are increasingly important [38, 67]. Representation engineering [91] systematically studies how concepts are encoded in hidden states and has been used to detect and control issues like dishonesty in NL.

Bui *et al.* [13] also use LLM internal states for code correctness, but classify single programs, while we rank candidates. Our method uses RepE to capture differences between correct and incorrect programs, unlike their separate classifier approach.

Huang *et al.* [30] use internal states for line-level risk assessment, flagging potentially incorrect code. Their classification-based setup (on tasks like code repair, translation, and editing) differs from our ranking approach and program synthesis benchmarks we used.

9 CONCLUSION

In this work, we demonstrated that LLMs encode meaningful internal representations of code correctness. We adapted RepE to source code, validated its effectiveness on *HumanEval* and *BigCodeBench*, and demonstrated that our representation-based selection method outperforms basic output probabilities and reflective confidence metrics. We extended our efforts by implementing a ranking setup that leverages these correctness signals to select higher-quality solutions from multiple generations, improving pass@1 performance and approaching a theoretical pass@10 ceiling. We also achieve superior or comparable performance to the state-of-the-art RankEF ranker model across most settings. The data and code necessary to replicate our experiments is publicly available. [5] We consider continued exploration of LLM internals promises further improvements in trustworthy AI-assisted programming.

REFERENCES

[1] Altaf Allah Abbassi et al. *Unveiling Inefficiencies in LLM-Generated Code: Toward a Comprehensive Taxonomy.* 2025. arXiv: 2503.06327 [cs.SE].

[2] Hervé Abdi et al. "Principal component analysis". In: *WIREs Computational Statistics* 2.4 (2010), pp. 433–459. DOI: https://doi.org/10.1002/wics.101. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wics.101.

[3] Meta AI. *Llama-3.2-11B-Vision-Instruct API.* https://api.together.ai/playground/chat/meta-llama/Llama-Vision-Free. Accessed: 2025-07-18. 2024.

[4] Jacob Austin et al. *Program Synthesis with Large Language Models.* 2021. arXiv: 2108.07732 [cs.PL].

[5] Anonynous Authors. *On LLMs' Internal Representation of Code Correctness.* https://figshare.com/s/6ded6fcc84af7a088161. Replication package. 2026.

[6] Nishant Balepur et al. *Which of These Best Describes Multiple Choice Evaluation with LLMs? A) Forced B) Flawed C) Fixable D) All of the Above.* 2025. arXiv: 2502.14127 [cs.CL].

[7] Shraddha Barke et al. "Grounded Copilot: How Programmers Interact with Code-Generating Models". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (Apr. 2023). DOI: 10.1145/3586030.

[8] Mohammad Beigi et al. "InternalInspector I^2: Robust Confidence Estimation in LLMs through Internal States". In: *Findings of the Association for Computational Linguistics: EMNLP 2024.* Ed. by Yaser Al-Onaizan et al. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 12847–12865. DOI: 10.18653/v1/2024.findings-emnlp.751.

[9] Emily M. Bender et al. "Climbing towards NLU: On Meaning, Form, and Understanding in the Age of Data". In: *Proceedings of the 58th Annual Meeting of*

the Association for Computational Linguistics. Ed. by Dan Jurafsky et al. Online: Association for Computational Linguistics, July 2020, pp. 5185–5198. DOI: 10.18653/v1/2020.acl-main.463.

[10] Emily M. Bender et al. "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?" In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency.* FAccT '21. Virtual Event, Canada: Association for Computing Machinery, 2021, pp. 610–623. ISBN: 9781450383097. DOI: 10.1145/3442188.3445922.

[11] Steven Bills et al. *Language models can explain neurons in language models.* https://openaipublic.blob.core.windows.net/neuron-explainer/paper/index.html. 2023.

[12] Tolga Bolukbasi et al. "Man is to computer programmer as woman is to homemaker? debiasing word embeddings". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems.* NIPS'16. Barcelona, Spain: Curran Associates Inc., 2016, pp. 4356–4364. ISBN: 9781510838819.

[13] Tuan-Dung Bui et al. *Correctness Assessment of Code Generated by Large Language Models Using Internal Representations.* 2025. arXiv: 2501.12934 [cs.SE].

[14] Zhuchen Cao et al. *Pragmatic Reasoning improves LLM Code Generation.* 2025. arXiv: 2502.15835 [cs.CL].

[15] Anthony Chen et al. "Evaluating Question Answering Evaluation". In: *Proceedings of the 2nd Workshop on Machine Reading for Question Answering.* Ed. by Adam Fisch et al. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 119–124. DOI: 10.18653/v1/D19-5817.

[16] Mark Chen et al. *Evaluating Large Language Models Trained on Code.* 2021. arXiv: 2107.03374 [cs.LG].

[17] Wentao Chen et al. *Memorize or Generalize? Evaluating LLM Code Generation with Evolved Questions.* 2025. arXiv: 2503.02296 [cs.AI].

[18] Yida Chen et al. *Beyond Surface Statistics: Scene Representations in a Latent Diffusion Model.* 2023. arXiv: 2306.05720 [cs.CV].

[19] Peter Clark et al. "Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge". In: *CoRR* abs/1803.05457 (2018). arXiv: 1803.05457.

[20] Arthur Conmy et al. "Towards automated circuit discovery for mechanistic interpretability". In: *Proceedings of the 37th International Conference on Neural Information Processing Systems.* NIPS '23. New Orleans, LA, USA: Curran Associates Inc., 2024.

[21] Zhiyu Fan et al. "Oracle-Guided Program Selection from Large Language Models". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis.* ISSTA 2024. Vienna, Austria: Association for Computing Machinery, 2024, pp. 628–640. ISBN: 9798400706127. DOI: 10.1145/3650212.3680308.

[22] Yujia Fu et al. "Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study". In: *ACM Trans. Softw. Eng. Methodol.* (Feb. 2025). Just Accepted. ISSN: 1049-331X. DOI: 10.1145/3716848.

[23] Github-Copilot. *Pull Request: Fix IndexOutOfRangeException in RegexInterpreter.Backtrack method.* 2025. eprint: https://github.com/dotnet/runtime/pull/115733.

[24] Chuan Guo et al. "On calibration of modern neural networks". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70.* ICML'17. Sydney, NSW, Australia: JMLR.org, 2017, pp. 1321–1330.

[25] Dan Hendrycks et al. "Measuring Coding Challenge Competence With APPS". In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2).* 2021.

[26] Dan Hendrycks et al. "Measuring Massive Multitask Language Understanding". In: *International Conference on Learning Representations.* 2021.

[27] Evan Hernandez et al. *Natural Language Descriptions of Deep Visual Features.* 2022. arXiv: 2201.11114 [cs.CV].

[28] Abram Hindle et al. "On the naturalness of software". In: *Proceedings of the 34th International Conference on Software Engineering.* ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 837–847. ISBN: 9781467310673.

[29] Xinyi Hou et al. "Large Language Models for Software Engineering: A Systematic Literature Review". In: *ACM Trans. Softw. Eng. Methodol.* 33.8 (Dec. 2024). ISSN: 1049-331X. DOI: 10.1145/3695988.

[30] Yuheng Huang et al. "Risk Assessment Framework for Code LLMs via Leveraging Internal States". In: *arXiv e-prints*, arXiv:2504.14640 (Apr. 2025), arXiv:2504.14640. DOI: 10.48550/arXiv.2504.14640. arXiv: 2504.14640 [cs.SE].

[31] Jeevana Priya Inala et al. *Fault-Aware Neural Code Rankers.* 2022. arXiv: 2206.03865 [cs.PL].

[32] Carlos E Jimenez et al. "SWE-bench: Can Language Models Resolve Real-world Github Issues?" In: *The Twelfth International Conference on Learning Representations.* 2024.

[33] Saurav Kadavath et al. *Language Models (Mostly) Know What They Know.* 2022. arXiv: 2207.05221 [cs.CL].

[34] Yuval Kirstain et al. "A Few More Examples May Be Worth Billions of Parameters". In: *Findings of the Association for Computational Linguistics: EMNLP 2022.* Ed. by Yoav Goldberg et al. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, Dec. 2022, pp. 1017–1029. DOI: 10.18653/v1/2022.findings-emnlp.72.

[35] Tom Kwiatkowski et al. "Natural Questions: A Benchmark for Question Answering Research". In: *Transactions of the Association for Computational Linguistics* 7 (2019). Ed. by Lillian Lee et al., pp. 452–466. DOI: 10.1162/tacl_a_00276.

[36] Nancy G. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety.* The MIT Press, Jan. 2012. ISBN: 9780262298247. DOI: 10.7551/mitpress/8179.001.0001. eprint: https://direct.mit.edu/book-pdf/2280500/book_9780262298247.pdf.

[37] Jia Li et al. *Showing LLM-Generated Code Selectively Based on Confidence of LLMs.* 2024. arXiv: 2410.03234 [cs.SE].

[38] Jiliang Li et al. "Do Machines and Humans Focus on Similar Code? Exploring Explainability of Large Language Models in Code Summarization". In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension.* ICPC '24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 47–51. ISBN: 9798400705861. DOI: 10.1145/3643916.3644434.

[39] Shuang Li et al. "Assessing the Performance of AI-Generated Code: A Case Study on GitHub Copilot". In: *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE).* 2024, pp. 216–227. DOI: 10.1109/ISSRE62328.2024.00030.

[40] Wangyue Li et al. "Can Multiple-choice Questions Really Be Useful in Detecting the Abilities of LLMs?" In: *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024).* Ed. by Nicoletta Calzolari et al. Torino, Italia: ELRA and ICCL, May 2024, pp. 2819–2834.

[41] Yujia Li et al. "Competition-level code generation with AlphaCode". In: *Science* 378.6624 (2022), pp. 1092–1097. DOI: 10.1126/science.abq1158. eprint: https://www.science.org/doi/pdf/10.1126/science.abq1158.

[42] Stephanie Lin et al. "Teaching Models to Express Their Uncertainty in Words". In: *Trans. Mach. Learn. Res.* 2022 (2022).

[43] Stephanie Lin et al. "TruthfulQA: Measuring How Models Mimic Human Falsehoods". In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Ed. by Smaranda Muresan et al. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 3214–3252. DOI: 10.18653/v1/2022.acl-long.229.

[44] Zhen Lin et al. "Contextualized Sequence Likelihood: Enhanced Confidence Scores for Natural Language Generation". In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing.* Ed. by Yaser Al-Onaizan et al. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 10351–10368. DOI: 10.18653/v1/2024.emnlp-main.578.

[45] Jiawei Liu et al. "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation". In: *Thirty-seventh Conference on Neural Information Processing Systems.* 2023.

[46] David Lo. " Trustworthy and Synergistic Artificial Intelligence for Software Engineering: Vision and Roadmaps ". In: *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE).* Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 69–85. DOI: 10.1109/ICSE-FoSE59343.2023.00010.

[47] Vahid Majdinasab et al. "Assessing the Security of GitHub Copilot's Generated Code - A Targeted Replication Study". In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* 2024, pp. 435–444. DOI: 10.1109/SANER60148.2024.00051.

[48] Shane Mcintosh et al. "An empirical study of the impact of modern code review practices on software quality". In: *Empirical Softw. Engg.* 21.5 (Oct. 2016), pp. 2146–2189. ISSN: 1382-3256. DOI: 10.1007/s10664-015-9381-9.

[49] Emanuele Mezzi et al. *Large Language Models are Unreliable for Cyber Threat Intelligence.* 2025. arXiv: 2503.23175 [cs.CR].

[50] Todor Mihaylov et al. "Can a Suit of Armor Conduct Electricity? A New Dataset for Open Book Question Answering". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing.* Ed. by Ellen Riloff et al. Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 2381–2391. DOI: 10.18653/v1/D18-1260.

[51] Tomas Mikolov et al. "Linguistic Regularities in Continuous Space Word Representations". In: *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.* Ed. by Lucy Vanderwende et al. Atlanta, Georgia: Association for Computational Linguistics, June 2013, pp. 746–751.

[52] Ran Mo et al. "Assessing and Analyzing the Correctness of GitHub Copilot's Code Suggestions". In: *ACM Trans. Softw. Eng. Methodol.* (Jan. 2025). Just Accepted. ISSN: 1049-331X. DOI: 10.1145/3715108.

[53] Francesco Maria Molfese et al. *Right Answer, Wrong Score: Uncovering the Inconsistencies of LLM Evaluation in Multiple-Choice Question Answering.* 2025. arXiv: 2503.14996 [cs.CL].

[54] Jiwon Moon et al. *Don't Judge Code by Its Cover: Exploring Biases in LLM Judges for Code Evaluation.* 2025. arXiv: 2505.16222 [cs.CL].

[55] Nhan Nguyen et al. "An empirical evaluation of GitHub copilot's code suggestions". In: *Proceedings of the 19th International Conference on Mining Software Repositories.* MSR '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1–5. ISBN: 9781450393034. DOI: 10.1145/3524842.3528470.

[56] Maxime Oquab et al. "DINOv2: Learning Robust Visual Features without Supervision". In: *Trans. Mach. Learn. Res.* 2024 (2024).

[57] Hammond Pearce et al. "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions". In: *Commun. ACM* 68.2 (Jan. 2025), pp. 96–105. ISSN: 0001-0782. DOI: 10.1145/3610721.

[58] Luca Ponzanelli et al. "Prompter: A Self-Confident Recommender System". In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 577–580. DOI: 10.1109/ICSME.2014.99.

[59] BigCode Project. *BigCode Evaluation Harness*. Accessed: 2025-07-18. 2025.

[60] BigCode Project. *BigCodeBench v0.2.4*. https://github.com/bigcode-project/bigcodebench/releases/tag/v0.2.4. Accessed: 2025-07-18. 2024.

[61] Alec Radford et al. *Learning to Generate Reviews and Discovering Sentiment*. 2017. arXiv: 1704.01444 [cs.LG].

[62] Pranav Rajpurkar et al. *SQuAD: 100,000+ Questions for Machine Comprehension of Text*. 2016. arXiv: 1606.05250 [cs.CL].

[63] Siva Reddy et al. *CoQA: A Conversational Question Answering Challenge*. 2019. arXiv: 1808.07042 [cs.CL].

[64] Joshua Robinson et al. *Leveraging Large Language Models for Multiple Choice Question Answering*. 2023. arXiv: 2210.12353 [cs.CL].

[65] Patrick Schramowski et al. *BERT has a Moral Compass: Improvements of ethical and moral values of machines*. 2019. arXiv: 1912.05238 [cs.CL].

[66] Arindam Sharma et al. *Assessing Correctness in LLM-Based Code Generation via Uncertainty Estimation*. 2025. arXiv: 2502.11620 [cs.SE].

[67] Chandan Singh et al. *Rethinking Interpretability in the Era of Large Language Models*. 2024. arXiv: 2402.01761 [cs.CL].

[68] Claudio Spiess et al. *Calibration and Correctness of Language Models for Code*. 2024. arXiv: 2402.02047 [cs.SE].

[69] Zhihong Sun et al. "Sifting through the Chaff: On Utilizing Execution Feedback for Ranking the Generated Code Candidates". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 229–241. ISBN: 9798400712487. DOI: 10.1145/3691620.3695000.

[70] Alexey Svyatkovskiy et al. "IntelliCode compose: code generation using transformer". In: ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1433–1443. ISBN: 9781450370431. DOI: 10.1145/3368089.3417058.

[71] Alon Talmor et al. "CommonsenseQA: A Question Answering Challenge Targeting Commonsense Knowledge". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein et al. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4149–4158. DOI: 10.18653/v1/N19-1421.

[72] EvalPlus Team. *EvalPlus MBPP+ Release v0.2.0*. https://github.com/evalplus/mbppplus_release/releases/tag/v0.2.0. Accessed: 2025-07-18. 2024.

[73] Adly Templeton et al. "Scaling Monosemanticity: Extracting Interpretable Features from Claude 3 Sonnet". In: *Transformer Circuits Thread* (2024).

[74] Katherine Tian et al. "Just Ask for Calibration: Strategies for Eliciting Calibrated Confidence Scores from Language Models Fine-Tuned with Human Feedback". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by Houda Bouamor et al. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 5433–5442. DOI: 10.18653/v1/2023.emnlp-main.330.

[75] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017.

[76] Yanlin Wang et al. *Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models*. 2025. arXiv: 2407.00456 [cs.SE].

[77] Yue Wang et al. "CodeT5+: Open Code Large Language Models for Code Understanding and Generation". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by Houda Bouamor et al. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 1069–1088. DOI: 10.18653/v1/2023.emnlp-main.68.

[78] Zhijie Wang et al. " Towards Understanding the Characteristics of Code Generation Errors Made by Large Language Models ". In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 2587–2599. DOI: 10.1109/ICSE55347.2025.00180.

[79] Fangyun Wei et al. *Rethinking Generative Large Language Model Evaluation for Semantic Comprehension*. 2024. arXiv: 2403.07872 [cs.CL].

[80] Sean Welleck et al. *From Decoding to Meta-Generation: Inference-time Algorithms for Large Language Models*. 2024. arXiv: 2406.16838 [cs.CL].

[81] Mengge Xue et al. "Strengthened Symbol Binding Makes Large Language Models Reliable Multiple-Choice Selectors". In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Lun-Wei Ku et al. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 4331–4344. DOI: 10.18653/v1/2024.acl-long.237.

[82] S. Yoo et al. "Regression testing minimization, selection and prioritization: a survey". In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120. DOI: https://doi.org/10.1002/stvr.430. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.430.

[83] Fiorella Zampetti et al. "A Study on the Interplay between Pull Request Review and Continuous Integration Builds". In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019, pp. 38–48. DOI: 10.1109/SANER.2019.8667996.

[84] Anqi Zhang et al. *Reasoning Models Know When They're Right: Probing Hidden States for Self-Verification*. 2025. arXiv: 2504.05419 [cs.AI].

[85] Caiqi Zhang et al. "LUQ: Long-text Uncertainty Quantification for LLMs". In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Ed. by Yaser Al-Onaizan et al. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 5244–5262. DOI: 10.18653/v1/2024.emnlp-main.299.

[86] Shun Zhang et al. *Planning with Large Language Models for Code Generation*. 2023. arXiv: 2303.05510 [cs.LG].

[87] Tianyi Zhang et al. "Coder reviewer reranking for code generation". In: *Proceedings of the 40th International Conference on Machine Learning*. ICML'23. Honolulu, Hawaii, USA: JMLR.org, 2023.

[88] Kaitlyn Zhou et al. "Navigating the Grey Area: How Expressions of Uncertainty and Overconfidence Affect Language Models". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by Houda Bouamor et al. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 5506–5524. DOI: 10.18653/v1/2023.emnlp-main.335.

[89] Yuqi Zhu et al. "Hot or Cold? Adaptive Temperature Sampling for Code Generation with Large Language Models". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 38.1 (Mar. 2024), pp. 437–445. DOI: 10.1609/aaai.v38i1.27798.

[90] Terry Yue Zhuo et al. "BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions". In: *arXiv preprint arXiv:2406.15877* (2024).

[91] Andy Zou et al. *Representation Engineering: A Top-Down Approach to AI Transparency*. 2023. arXiv: 2310.01405 [cs.LG].