On the Robustness of LLMs' Internal Representation of Code Correctness

Francisco Ribeiro, Sohaila Abdulsattar*, Renata Gonzalez*, Mahmoud Kassem*, Sarah Nadi
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates
{francisco.ribeiro, sohaila.abdulsattar, re2230, mahmoud.kassem, sarah.nadi}@nyu.edu

***Abstract*—Code generated by modern language models can look plausible, yet fail its task, and tests may be costly or incomplete when many candidates are produced. Recent work uses a model's internal representations (hidden states) to rank candidate solutions without running tests at scoring time. However, this code-correctness signal was demonstrated under a single configuration, leaving open how much its usefulness depends on the choices used to capture it. We study this question across four instruction-tuned models and two Python benchmarks, varying how the signal is constructed, how candidates are presented to the model, where their representations are read, and which examples are used to fit the signal. We also construct controlled pairs of correct programs and faulty variants to examine whether a signal captured from these pairs transfers to independently generated candidates. Our results show that no single configuration is best across the studied settings, and that separating controlled faulty variants from their source solutions does not ensure effective ranking of generated candidates.**

***Index Terms*—large language models, code generation, code correctness, linear probing**

I. Introduction

Writing source code is increasingly delegated to large language models (LLMs). They complete the next line in an editor [1], generate whole functions from a description [2], and implement entire features at the repository level [3]. As they grow more fluent, their output is increasingly accepted with little scrutiny [4], and a growing fraction of the code reaching real systems now originates from a model rather than a human [5].

However, fluency is not the same as correctness. An LLM can produce code that reads naturally yet does not implement the requested functionality [6], [7]: it may be syntactically correct and even pass some tests, yet miss the specification, with nothing apparent in it reliably indicating whether it is correct. A good test suite can check correctness, but tests are often incomplete, missing, or themselves left to the model [8], and running them is expensive when the LLM produces many candidates. Early signs of correctness would therefore help developers trust generated code and prioritise which candidates to review. While the model's own confidence (i.e., the probability it assigns to the generated tokens) or its stated confidence when asked [9] are potential signals, research showed that they are not well calibrated with correctness [10], [11].

Instead of relying on the model's outputs, another option is to look at what the LLM internally computes. A model forms intermediate numerical representations of the input as it processes it, known as *hidden states*. These states are readily available and have been shown to encode high-level properties such as truthfulness [12], [13] or latent knowledge [14]. The main idea is that contrasting the hidden states of inputs that have a property against those that lack it recovers a vector along which the concept varies [12]–[15]. This vector, a *representation* of the concept, can then be used to score new inputs by how far they point along it, indicating whether the input has the property. We use the term *linear probing* for this general approach of extracting a concept from hidden states.

Most linear probing work has targeted natural language (NL) concepts such as sentiment [15] or truthfulness [12], [13]. However, our recent work showed that it can also capture code correctness [11] where contrasting an LLM's hidden states for correct versus incorrect implementations of the same task isolates a *code correctness representation*. Figure 2 illustrates this idea, showing a model-generated implementation and the benchmark's canonical solution for one HumanEval task: returning the sum of the odd values that sit at even positions. The model's solution steps through the odd positions instead of the even ones. Our previously captured code correctness representation ranked the correct program above the incorrect one without needing test execution, while the model's own output probability and stated confidence did not. However, the highest obtained accuracy ranged from 41–63% across models and benchmarks, showing that performance varies across settings. Furthermore, for a fixed model/benchmark combination, accuracy changed with the data subsets used to capture and validate the representation and those used for testing.

Accordingly, the current results in the literature are promising but leave open questions about the representation's quality and its robustness to data variation. Our goal is to systematically investigate the factors that influence the quality and robustness of code correctness repre-

*These authors contributed equally and are listed alphabetically.

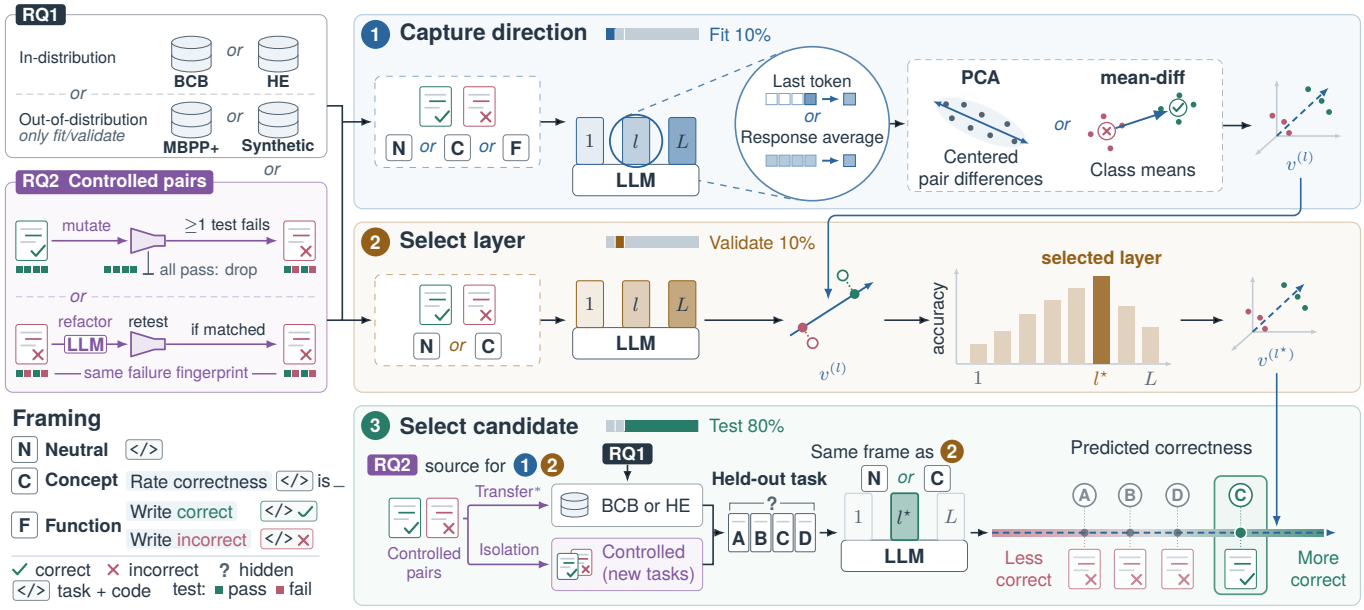

Fig. 1: Overview of the methodology. RQ2 fits and validates on test-filtered controlled pairs, then tests held-out pairs (Isolation) or benchmark candidates (Transfer). *There is a Transfer variant that selects the layer using benchmark validation data.

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

(a) Incorrect: LLM-generated attempt that iterates over *odd* positions.

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

(b) Correct: canonical solution that iterates over *even* positions.

Fig. 2: Two implementations of HumanEval/121 — *return the sum of the odd values at even positions.*

sentations. We make two key observations that motivate our study design. First, there are multiple methods to construct a concept representation in the literature [15], [16]. We hypothesize that the choice of method may affect the quality of the captured representation, and that perhaps some methods may generalize better than others across models and benchmarks. Second, the contrast between correct and incorrect programs may carry incidental differences that are not relevant to correctness, such as variable names or implementation style. Such captured incidental differences may explain why the correctness representation becomes sensitive to data variation. For example, in the example in Figure 2, the main fault is on Line 4, where the model iterates over odd positions instead of even ones. However, the two programs also differ in other ways: the correct program uses a comprehension with `sum` while the incorrect one uses a loop with an accumulator, they use different variable names, and they check for parity differently (`%2!=0` vs. `%2==1`).

In this paper, we systematically investigate the design choices behind capturing a code correctness representation to assess their effect. We also investigate how data variation affects the quality of the captured correctness signal. More specifically, we answer the following research questions (RQs):

RQ1 How do the key design choices involved in capturing a code correctness direction (the direction-construction method, prompt framing, and hidden-state location) affect its quality and robustness?

RQ2 Does controlling for differences between correct/incorrect programs via mutations and refactorings improve the quality of the correctness direction?

Figure 1 provides an overview of our methodology and summarizes the data sources and the fitting, validation, and testing stages for both RQs.

Our evaluation covers *in-distribution* settings, where the direction is fit and tested on disjoint subsets of the same benchmark (HumanEval [2] and BigCodeBench [17]), and *out-of-distribution* settings, where it is fit on MBPP+ [18], [19] and tested on either benchmark, across four instruction-tuned LLMs.

In the reported sweeps, we find that mean-diff has higher average accuracy than PCA on both benchmarks, but not in every configuration; the preferred read-out location, framing, and model vary. Directions fit on controlled mutations separate held-out mutations, but do not improve selection of independently generated candidates: the mutation-fit versus in-distribution accuracies are 19.6% versus 56.7% on BigCodeBench and 47.6% versus 52.8% on HumanEval. Because the fitting datasets also differ in content and size, this comparison cannot show that isolating the fault alone caused the result.

Overall, the contributions of this paper are threefold: 1) a systematic study of how the *method* of constructing the direction, the *framing* of the prompt, and the *hidden-state location* read from the model affect the accuracy

of the captured code correctness direction across four LLMs and two benchmarks. 2) a data generation pipeline that isolates bug-causing changes via controlled mutations and behavior-preserving refactorings to test if directions built from such data, rather than from model-generated failing attempts, carry a higher quality correctness signal. 3) a replication package with experiment code, data, and results:

Replication package
Code Archive https://doi.org/10.6084/m9.figshare.32849732 **Artifacts** https://doi.org/10.6084/m9.figshare.31752142 **Repository** https://github.com/sanadlab/llm-code-correctness

II. Background

This section presents foundational aspects to understand how LLMs represent data internally, how to extract a direction corresponding to a target concept, and how prior work applied this to code correctness.

A. Hidden states locations

A decoder-only transformer language model maps a token sequence to hidden states through a stack of L layers [20]. For an input x, it produces a hidden state $h_i^{(l)}(x) \in \mathbb{R}^d$ at every layer $l \in \{0, 1, \ldots, L\}$ and every token position i, where $l = 0$ is the embedding layer and d is the hidden size. These states encode high-level semantic and syntactic properties recoverable with simple probing methods [15]. As a hidden state exists for every token at every layer, it requires deciding which token to read the hidden state from. Two choices are common: reading the *last token*, a single hidden state that has attended to the entire preceding stimulus, or the *response average*, the mean of the hidden states over the response tokens [15].

B. Construction methods

Linear probing captures a direction in hidden-state space corresponding to a target concept by contrasting representations of inputs that differ in it [15]. Starting from paired *positive* and *negative* stimuli, it takes a hidden-state representation for each at a chosen location (Section II-A) and turns the differences into one concept direction $v^{(l)}$ per layer. Truthfulness, sentiment, and other abstract concepts are recoverable in this linear form and can be used to monitor model behaviour [12]–[14], [21].

Two prominent methods turn contrasting pairs into a direction: **(a) Principal component analysis (PCA):** the per-pair representation differences at layer l are collected and mean-centered, and $v^{(l)}$ is taken as their first principal component (i.e., the direction of largest variance among the contrasts) [22]. Its orientation is arbitrary, so the sign is resolved by a per-layer majority rule over labeled examples, so that higher projections correspond to inputs that exhibit the concept [15]. **(b) Mean-difference (mean-diff):** the direction is the difference between the positive and negative class means at layer l,

$$v^{(l)} = \bar{a}_+^{(l)} - \bar{a}_-^{(l)}, \tag{1}$$

where $\bar{a}_+^{(l)}$ and $\bar{a}_-^{(l)}$ are the class means. Its orientation is fixed, so no separate sign step is needed. Despite its simplicity, mean-diff has been reported to often recover directions better than PCA [16]. Regardless of how a direction is captured, a new input is scored by projecting its representation onto the direction: the further it points along it, the more it exhibits the concept.

C. Framings

We use the term *framing* for how the input is wrapped into a prompt. Framing shapes which aspects of the model's internal state the contrast captures. Figure 3 shows three common framings in this line of work: **(a) Neutral:** The model is shown only the input, with no reference to the concept [11], so any signal must arise from the model's own unguided processing. This is the most conservative framing. **(b) Concept:** The stimulus adds a meta-instruction directing the model to consider how much of the concept is present, eliciting its declarative knowledge about that concept [15]. **(c) Function:** The stimulus instructs the model to produce behaviour that exhibits the concept or its opposite, such as a correct or an incorrect implementation, contrasting the instruction–conditioned states [15].

D. Using linear probing to capture code correctness

While most linear probing work targets abstract NL concepts, e.g. truthfulness [12], [13], our previous work adapted it to capture code correctness [11], the same concept we study in this paper. In our previous work, contrasting pairs joined a task's canonical solution with a *plausible failing attempt*: a model-generated implementation that fails the task's tests. Such an attempt can differ from the solution in its algorithm, structure, or naming, as well as in what makes it fail. We reported two findings. First, a code correctness direction can be *captured* and used to distinguish correct from incorrect programs more accurately than the model's own confidence, whether read *intrinsically* from token probabilities [23] or *reflectively* from a verbalised judgement [9]. Second, it is *useful for ranking*: ordering candidate implementations by their correctness score raises the chance a correct one is selected, matching or exceeding a specialized ranker [24]. However, our previous results came from a *single* choice on each axis: the direction was built with PCA, read from the last token, and fit under a neutral framing but evaluated under a concept one. Its accuracy also varied with the data split, so the signal may be sensitive to how it is captured, both in terms of method and data. This sensitivity and variation motivates our current systematic investigation into the robustness of code correctness representations.

III. Experimental Setup

This section describes our experimental setup. We first cover the aspects shared across both RQs (Sections III-A–III-D), and then settings specific to each RQ (Sections III-E and III-F).

A. Fitting, Validation, and Testing Protocol

We split tasks into disjoint fitting, validation, and test partitions, used in the three steps of Figure 1. The per-layer direction is captured on the fitting partition (step 1). Since the most informative layer is not known in advance, the validation partition selects a single layer $l^\star$ (step 2): the one whose direction best separates correct from incorrect programs under the validation framing (Section III-B). We report accuracy on the held-out test partition at $l^\star$ under the matching test framing (step 3).

A program is scored by projecting its representation at the chosen location onto the layer's direction,

$$s^{(l)}(x) \;=\; \sigma^{(l)}\,\frac{\langle h^{(l)}(x),\, v^{(l)}\rangle}{\|v^{(l)}\|}, \tag{2}$$

where $\sigma^{(l)} \in \{+1, -1\}$ is the per-layer sign: the majority-rule value for PCA (Section II-B), and $+1$ for mean-diff, whose orientation is already fixed. A higher score is intended to indicate a more likely correct program.

We use two metrics. **Pairwise accuracy** is the fraction of same-task correct/incorrect pairs in which the correct program scores higher (a 1-of-2 choice). During validation, we select $l^\star$ as the layer with the highest pairwise accuracy, breaking ties toward the deepest layer. **Selection accuracy** is the fraction of tasks for which the correct program is selected from four candidates (a 1-of-4 choice). We report it on the benchmark test sets and on RQ2's held-out controlled data, matching the intended use and prior work [11], [15]. For selection accuracy, $\mathrm{Acc}_{\mathrm{Val}}$ denotes accuracy at the layer selected on validation data without test labels; **best-layer accuracy** uses the layer with the highest test accuracy, an oracle upper bound useful for examining layer-selection cost.

Prior work established that the direction can outperform the model's own confidence [11], so we do not repeat that comparison. Instead, we focus on how construction and data affect the direction itself.

B. Design Space

A configuration pairs a construction method (PCA or mean-diff), a framing (neutral, concept, or function), and a hidden-state location (last token or response average). Both methods can capture incidental differences such as length or formatting: PCA follows the largest variation among pair differences, whether or not it reflects correctness, while mean-diff follows their average, which outliers can skew. The framing asks how explicitly, if at all, the LLM is prompted for correctness. The read-out location asks whether a single fully-informed position suffices (last token) or pooling across the response helps (response average).

A framing mode specifies one framing for fitting the direction and another for evaluating it. Validation and test use the same framing, so the layer is selected under the condition in which it is tested. Rather than crossing every fit framing with every evaluation framing, we use six modes, written *fit*→*eval*: `N`→`N`, `N`→`C`, `C`→`N`, `C`→`C`, `F`→`N`, and `F`→`C`, where `N`, `C`, and `F` denote the neutral, concept, and function framings. The function framing is restricted to fitting, since it pairs a positive instruction with the correct program and a negative one with the incorrect program, so the instruction itself reveals the label and would leak it at validation or test time; it therefore never appears as an evaluation framing.

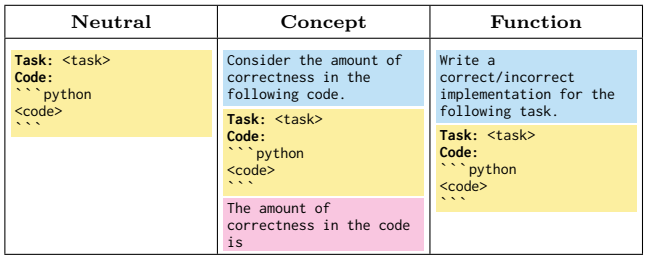

Fig. 3: The three framings' core is a task and candidate program (yellow). *Neutral* is the core only. *Concept* adds a meta-instruction (blue) and an answer cue (pink). *Function* prepends a correct/incorrect implementation instruction (blue).

RQ1 varies these three axes with the data fixed to the correct/incorrect pairs of prior work [11]. RQ2 fixes the configuration and varies the data, using pairs we construct instead (Sections III-E and III-F).

C. Models and Benchmarks

We use the same four instruction-tuned LLMs as prior work [11], with their short names in bold: **Mistral**-7B-Instruct-v0.3, **Qwen**-2.5-Coder-7B-Instruct, **OpenCoder**-8B-Instruct, and **CodeLlama**-7B-Instruct. All are open-weights (required to read hidden states) and lie in the 7–8B range, so scale does not confound cross-model comparison. Mistral is the general-purpose model from the original representation engineering work [15], and the other three are code-specialized. We evaluate on the same two benchmarks: HumanEval [2], with 164 self-contained problems over core Python and foundational algorithms, and BigCodeBench [17], with 1,140 tasks that call external libraries such as `numpy` and `pandas`. Each task provides a correct solution and an executable test suite, which we use as a proxy for correctness. As a methodological check, we reproduce the single configuration of prior work [11], a PCA direction read from the last-token hidden state fit under a neutral framing and evaluated under a concept one, and confirm reproduction of its published accuracies.

D. Stimulus Construction

A *stimulus* is the input from which we collect hidden states: a task and candidate program presented under one of the three framings (Section II-C), as shown in Figure 3. Our previous work assembled it as a plain concatenated string [11]; here, we build it structurally, e.g. using the model's `chat_template` for the neutral framing.

E. RQ1: Data Settings

We use the same candidate data from our previous work [11] where RQ1 data includes 457 BigCodeBench

tasks and 151 HumanEval tasks, rather than all 1,140 and 164 tasks in the full benchmarks. The subset is due to requiring each task to have three failing implementations. In the fitting and validation partitions, each task record pairs one correct reference implementation with one pre-generated incorrect implementation. At test time, each task has the reference solution and three pre-generated implementations that fail the benchmark's tests. The same candidate sets are used for every probed model; the failing implementations are not generated by that model.

In-distribution: the supplied data contains ten task-level splits for each benchmark. Within a split, fitting, validation, and test tasks are disjoint. Each BigCodeBench split has 45, 45, and 367 tasks in these partitions, respectively; each HumanEval split has 15, 15, and 121. These counts approximate a 10/10/80 division of the 457 BigCodeBench and 151 HumanEval tasks.

Out-of-distribution: RQ1 also fits and validates on two out-of-distribution sources: MBPP+ [18], [19] and the synthetic set inherited from prior work [11]. These are separate from RQ2's controlled mutations and refactorings. Each of the four MBPP+ splits contains 24 fitting tasks and 24 validation tasks. Across the four splits, each of the 96 source tasks appears once in fitting and once in validation. The synthetic source follows the same arrangement, with five tasks in each role per split and 20 source tasks overall. Within each split, fitting and validation tasks are disjoint. Each inner fit is crossed with the ten test splits of the target benchmark (HumanEval or BigCodeBench).

Therefore, the full RQ1's configuration space is $2\times2\times3\times2\times3\times2\times4 = 576$ configurations, corresponding to method, read-out location, fit framing, validation/test framing, fit source, test dataset, and model, respectively.

F. RQ2: Data Settings

RQ2 keeps the configuration fixed and changes the data the direction is fit and evaluated on. Starting from the HumanEval and BigCodeBench tasks, we construct correct/incorrect pairs in four steps:

Step 1: Mutant Generation. We introduce a bug into a correct solution with a rule-based mutant generator that applies one of 20 operators, each a small and targeted edit (Table I). The generator parses the program into an Abstract Syntax Tree (AST) with type inference via `astroid` [25], keeping mutations syntactically valid and applied only to semantically compatible types. For example, it blocks arithmetic mutations (swapping `+` for `-`) on string or list operands, which would fail with a trivial `TypeError` rather than a plausible logical bug; since Step 2 retains mutants that fail the test suite, such edits would otherwise pass that filter alongside logical faults. This step mutated 1,135 BigCodeBench tasks (20 operators applied; statement deletion (SD) most abundant at ∼31%) and 143 HumanEval tasks (19 operators applied; constant replacement (CR) most abundant at ∼26.8%), dropping only tasks with no applicable mutant.

TABLE I: The 20 mutation operators used to build the controlled pairs.

Code	Operator	Example (orig. → mut.)
Arithmetic & Relational		
AOR	Arithmetic Op. Replace	`a + b` → `a - b`
AOD	Arithmetic Op. Delete	`a + b` → `a`
AOI	Arithmetic Op. Insert	`a` → `a + 1`
ROR	Relational Op. Replace	`a == b` → `a != b`
BOR	Bitwise Op. Replace	`a | b` → `a & b`
NEG	Negation	`x` → `-x`
Variables & Constants		
CR	Constant Replace	`x = 5` → `x = 6`
VCR	Variable to Constant	`return x` → `return 0`
CVR	Constant to Variable	`return 0` → `return x`
VVR	Variable to Variable	`a = b` → `a = c`
CRN	Constructor to None	`Node()` → `None`
Control Flow & Logic		
SD	Statement Delete	`x = 1; y = 2` → `y = 2`
COR	Conditional Op. Replace	`a and b` → `a or b`
COD	Conditional Op. Delete	`a and b` → `a`
UOI	Unary Op. Insert	`if x:` → `if not x:`
UOD	Unary Op. Delete	`not x` → `x`
RC	Remove Conditional	`if a:` → `if True:`
RVM	Return Value Mutate	`return x` → `return None`
TR	True Return	`return x` → `return True`
FR	False Return	`return x` → `return False`

Step 2: Mutant Validation. We execute every mutant against the benchmark's native test suite and keep only those that fail: *killed* mutants in mutation-testing terms [26]. We discard mutants that pass the tests. This yielded 27,928 killed mutants for BigCodeBench and 4,566 for HumanEval.

Step 3: Confounding. A model may exploit surface shortcuts, e.g. keying on variable naming or code length as the "bug" signal rather than the semantics. To counter this, we add confounds: structurally refactored versions of a mutant that fail the same way. We prompt `Qwen2.5-Coder-32B-Instruct` [27] to rewrite each killed mutant with behaviour-preserving edits, giving a distinct version of the same error.

Step 4: Confound Validation. The rewrite may introduce a different error or accidentally fix the bug, so we re-run each confound and keep it only if it **fails with the same failure fingerprint** as its source mutant, discarding any that pass or fail differently. This leaves 9,585 validated confounds for BigCodeBench (593 tasks) and 1,262 for HumanEval (66 tasks).

Data Splitting. Mirroring RQ1, we split tasks into 10/10/80 fitting/validation/test partitions in each of ten folds. From the validated pairs, we form two variants of *controlled pairs*: **M-Only**, pairing each reference solution with its mutants, and **MC**, which also adds the confounds. As shown in Figure 1, both variants are fit and validated on controlled pairs, then tested in two settings: **Isolation** tests on controlled pairs of held-out tasks, and **Transfer** tests on RQ1's 1-of-4 benchmark candidate selection task. A Transfer variant (Transfer*) instead selects the layer on the benchmark's validation split (Section V-6). Since a task yields many mutants and confounds, we sample pairs in two ways: *weighted* keeps one mutant per task (probability proportional to its operator's frequency) and, for MC, one uniformly random confound; *paired* uses every mutant and confound, giving more but uneven pairs. Fitting and validation use weighted sampling by default

and paired sampling as a robustness check (Section V); the Isolation test split always uses weighted sampling.

IV. RQ1: How do the key design choices involved in capturing a code correctness direction affect its quality and robustness?

Tables II and III show the full sweep of results: every valid combination of construction method, read-out location, and framing (Section III-B) across four models. Each cell is Acc_{Val}, the validation-selected accuracy (Section III-A); the first row of each table is the configuration used by prior work [11]. Across the full sweep, no configuration is best for every benchmark, fitting source, and model. Therefore, we ask how much each choice, on its own or combined with others, changes Acc_{Val}.

We measure this with an analysis of variance (ANOVA), reported in Table IV. Each choice is a *factor* — construction method, read-out location, fitting framing, testing framing, benchmark, and model. ANOVA gives every factor, and every combination of factors, a share of the variance in Acc_{Val}: its η^2. A single factor's share is its *main effect*; a combination's share is an *interaction*, describing variation associated with factors together. Each main effect and interaction is a *term*; the six factors and their combinations give 63 terms, whose η^2 sum to 100%. Table IV lists the largest terms.

1) General View: We start with in-distribution fitting and pool BigCodeBench and HumanEval to summarize the variance associated with each choice (Table IV, panel 1). Here the fitting source is tied to the benchmark: BigCodeBench data for BigCodeBench and HumanEval data for HumanEval. We therefore do not treat fitting source as a separate factor in this pooled analysis; we compare fitting sources within each benchmark later (panel 3).

Panel 1 shows that the observed variance is spread across many terms rather than concentrated in one. The two largest shares, 15% and 14%, are (1) the combination of fitting and testing framing and (2) the model. The benchmark on its own is negligible (under 1%) but it appears in several interaction terms: with the read-out location (12%), with the location and testing framing together (11%), and with the model (7%). Summed together, all terms involving the benchmark account for about 47% of the variance. In these results, the benchmark is associated with accuracy mainly in combination with other choices, rather than on its own. The read-out location is the clearest example: on its own it explains under 1%, but paired with the benchmark it explains 12%, because the location that performs best on one benchmark is worse on the other (in panel 2 its *best* level flips between the response average and the last token). A pooled ranking can therefore obscure benchmark-specific preferences. We next examine each benchmark separately (Table IV, panel 2).

2) Within each benchmark: Examined separately, each benchmark ranks the choices differently (Table IV, panel 2). On BigCodeBench the largest share is not a single choice but the fitting and testing framing acting together (26%), followed by the read-out location (20%) and that same location combined with the testing framing (15%). The location matters, but it cannot be set on its own: the framing it is read under changes its effect. Concept fitting shows this in the cells — a mean-diff direction read at the last token of BigCodeBench scores 63–69% across the four models when the testing prompt is also concept, but 37–55% under a neutral prompt (Table II).

In these sweeps, construction is the most consistent choice: mean-diff has higher average accuracy than PCA on both benchmarks, with a 14% main effect on BigCodeBench and a 3% main effect on HumanEval. This aggregate result does not mean mean-diff wins in every configuration; the individual results still vary across models, framings, and read-out locations.

Every other choice depends on the benchmark, and the read-out location is the clearest case. On BigCodeBench the response average wins: a neutral mean-diff direction reaches 71–77% across the four models when averaged over the response, against 48–54% at the last token (Table II). On HumanEval the last token wins, 65–75% against 50–64% for the same direction (Table III). This is the cancellation that made the location look negligible when the benchmarks were pooled (under 1% in panel 1) now seen from inside each one, and it is why the location's *best* level flips between the two in panel 2.

The models divide the same way. On BigCodeBench they agree: model identity accounts for only 11% of the variance, and the best configuration lands within a few points across the four (panel 2). On HumanEval they disagree sharply: model identity accounts for 51%, the largest single share on either benchmark, and the best per-model configuration ranges from 42% (Mistral) to 62% (Qwen) (Table III). A further 20% on HumanEval is the testing framing on its own, which favours the neutral prompt (panel 2). On HumanEval, a configuration that is best for one model need not be best for another.

3) Out-of-distribution fitting: We now return to the factor held out of the general view (Section IV-1): the source of the fitting data. So far, every direction was fit on its own benchmark's fitting data. Now, we fit it on data from MBPP+ or the synthetic set, while still testing on the same benchmark (the right two blocks of Table II and Table III). Panel 3 of Table IV reports the mean Acc_{Val} for each source. In-distribution fitting is best on both benchmarks, but both decrease when fitting out-of-distribution. On HumanEval it is a small decrease: mean accuracy goes from 50% to 43% on MBPP+ and 42% on the synthetic set, a decrease of 7 and 8 points, respectively. On BigCodeBench the decrease is larger: from 52% to 35% on MBPP+ and to 22% on the synthetic set, a decrease of 17 and 30 points, respectively. The fitting source therefore matters on both benchmarks, but considerably more on BigCodeBench than on HumanEval. At 22%, the synthetic-fit mean on BigCodeBench has

TABLE II: RQ1 configuration sweep on BigCodeBench. Per-model Acc_{Val} (mean and std-dev (subscript) over folds) for every construction method (PCA/mean-diff (MD)), read-out location (last token (LAST)/response average (AVG)), fitting framing, and validation=testing framing (NEUT/CONC/FUNC). First row ≈ prior work [11]; directions fit in- or out-of-distribution (MBPP+/synthetic); per-column maxima in **bold**. All numbers are percentages.

				in-distribution				OOD: MBPP+				OOD: synthetic			
method	loc	fit	val/test	Mistral	Qwen	CodeLlama	OpenCoder	Mistral	Qwen	CodeLlama	OpenCoder	Mistral	Qwen	CodeLlama	OpenCoder
Configuration of prior work [11]															
PCA	LAST	NEUT	CONC	$42.7_{\pm 4.5}$	$40.8_{\pm 5.8}$	$40.5_{\pm 8.5}$	$39.6_{\pm 5.1}$	$32.2_{\pm 3.6}$	$20.5_{\pm 7.6}$	$30.1_{\pm 6.8}$	$28.4_{\pm 4.3}$	$27.6_{\pm 12.2}$	$10.3_{\pm 1.9}$	$22.8_{\pm 4.1}$	$\mathbf{36.0}_{\pm 5.8}$
Full configuration sweep															
PCA	LAST	NEUT	NEUT	$38.8_{\pm 4.5}$	$34.4_{\pm 3.3}$	$38.3_{\pm 3.6}$	$39.5_{\pm 5.1}$	$23.5_{\pm 1.9}$	$22.7_{\pm 1.3}$	$20.3_{\pm 1.4}$	$19.1_{\pm 1.1}$	$32.0_{\pm 1.8}$	$22.7_{\pm 2.8}$	$26.3_{\pm 3.6}$	$24.1_{\pm 3.8}$
PCA	LAST	CONC	NEUT	$37.0_{\pm 4.1}$	$47.2_{\pm 5.4}$	$36.5_{\pm 3.3}$	$47.7_{\pm 3.6}$	$24.7_{\pm 7.3}$	$43.0_{\pm 8.3}$	$34.0_{\pm 12.7}$	$32.4_{\pm 6.6}$	$25.6_{\pm 2.1}$	$27.7_{\pm 4.2}$	$17.1_{\pm 2.3}$	$26.4_{\pm 4.6}$
PCA	LAST	CONC	CONC	$62.6_{\pm 6.1}$	$60.1_{\pm 3.5}$	$56.2_{\pm 3.9}$	$62.6_{\pm 2.5}$	$50.9_{\pm 8.6}$	$52.4_{\pm 2.0}$	$41.3_{\pm 7.2}$	$49.1_{\pm 5.1}$	$13.7_{\pm 1.3}$	$14.3_{\pm 0.6}$	$18.7_{\pm 2.7}$	$32.9_{\pm 1.9}$
PCA	LAST	FUNC	NEUT	$38.3_{\pm 5.9}$	$35.5_{\pm 3.7}$	$38.1_{\pm 3.7}$	$38.8_{\pm 5.9}$	$25.2_{\pm 1.7}$	$20.3_{\pm 1.2}$	$17.5_{\pm 1.0}$	$19.4_{\pm 1.2}$	$26.8_{\pm 1.4}$	$20.7_{\pm 2.3}$	$30.5_{\pm 8.5}$	$30.2_{\pm 3.9}$
PCA	LAST	FUNC	CONC	$42.7_{\pm 5.5}$	$39.7_{\pm 7.8}$	$36.1_{\pm 9.0}$	$36.7_{\pm 5.5}$	$30.5_{\pm 2.6}$	$26.4_{\pm 9.6}$	$30.3_{\pm 7.8}$	$32.3_{\pm 7.1}$	$16.3_{\pm 4.3}$	$9.6_{\pm 1.5}$	$13.8_{\pm 3.8}$	$17.0_{\pm 2.3}$
PCA	AVG	NEUT	NEUT	$64.5_{\pm 5.4}$	$65.7_{\pm 11.5}$	$63.4_{\pm 7.8}$	$61.8_{\pm 6.3}$	$34.6_{\pm 12.7}$	$45.8_{\pm 13.8}$	$32.9_{\pm 16.2}$	$49.5_{\pm 10.4}$	$25.1_{\pm 3.5}$	$21.2_{\pm 5.6}$	$22.3_{\pm 3.9}$	$32.2_{\pm 8.9}$
PCA	AVG	NEUT	CONC	$49.2_{\pm 6.8}$	$40.9_{\pm 6.0}$	$42.4_{\pm 4.8}$	$48.8_{\pm 8.0}$	$49.5_{\pm 2.3}$	$41.2_{\pm 2.5}$	$32.1_{\pm 6.6}$	$29.4_{\pm 5.5}$	$23.0_{\pm 7.3}$	$24.2_{\pm 10.9}$	$20.2_{\pm 4.7}$	$25.7_{\pm 3.0}$
PCA	AVG	CONC	NEUT	$41.3_{\pm 7.5}$	$52.0_{\pm 4.4}$	$41.3_{\pm 5.9}$	$56.9_{\pm 4.4}$	$44.5_{\pm 8.4}$	$49.3_{\pm 10.4}$	$44.9_{\pm 16.3}$	$41.0_{\pm 18.3}$	$26.4_{\pm 1.2}$	$27.0_{\pm 3.5}$	$28.0_{\pm 11.5}$	$29.3_{\pm 6.7}$
PCA	AVG	CONC	CONC	$61.2_{\pm 5.1}$	$59.9_{\pm 3.9}$	$58.7_{\pm 6.7}$	$63.1_{\pm 1.8}$	$57.0_{\pm 7.1}$	$\mathbf{54.2}_{\pm 7.1}$	$44.7_{\pm 5.4}$	$39.7_{\pm 8.8}$	$13.0_{\pm 0.9}$	$12.9_{\pm 1.1}$	$16.7_{\pm 6.0}$	$23.6_{\pm 1.9}$
PCA	AVG	FUNC	NEUT	$61.4_{\pm 5.3}$	$66.0_{\pm 2.6}$	$63.3_{\pm 5.9}$	$65.2_{\pm 6.2}$	$35.5_{\pm 14.0}$	$38.3_{\pm 11.2}$	$31.2_{\pm 11.6}$	$\mathbf{53.8}_{\pm 12.7}$	$\mathbf{33.3}_{\pm 3.7}$	$24.2_{\pm 5.5}$	$22.0_{\pm 2.9}$	$29.5_{\pm 7.3}$
PCA	AVG	FUNC	CONC	$26.0_{\pm 5.3}$	$41.8_{\pm 6.1}$	$36.9_{\pm 6.5}$	$48.2_{\pm 4.6}$	$27.1_{\pm 10.1}$	$38.9_{\pm 7.2}$	$26.9_{\pm 8.6}$	$27.1_{\pm 7.2}$	$10.7_{\pm 2.2}$	$13.7_{\pm 2.7}$	$14.6_{\pm 4.1}$	$17.2_{\pm 2.9}$
MD	LAST	NEUT	NEUT	$48.5_{\pm 3.9}$	$54.1_{\pm 10.0}$	$48.6_{\pm 6.1}$	$48.6_{\pm 7.2}$	$23.1_{\pm 1.8}$	$23.2_{\pm 1.4}$	$20.4_{\pm 1.5}$	$19.1_{\pm 1.2}$	$31.5_{\pm 1.1}$	$24.3_{\pm 3.1}$	$25.5_{\pm 4.4}$	$22.9_{\pm 2.2}$
MD	LAST	NEUT	CONC	$50.2_{\pm 3.6}$	$58.4_{\pm 8.5}$	$51.7_{\pm 8.2}$	$49.6_{\pm 5.4}$	$32.5_{\pm 2.8}$	$19.9_{\pm 8.3}$	$27.8_{\pm 6.9}$	$28.6_{\pm 7.3}$	$27.4_{\pm 7.5}$	$11.0_{\pm 4.4}$	$14.1_{\pm 3.9}$	$28.9_{\pm 4.9}$
MD	LAST	CONC	NEUT	$37.3_{\pm 4.1}$	$54.8_{\pm 6.1}$	$48.0_{\pm 7.3}$	$50.6_{\pm 3.8}$	$22.0_{\pm 6.2}$	$36.1_{\pm 10.7}$	$38.0_{\pm 7.7}$	$28.0_{\pm 8.2}$	$27.5_{\pm 3.5}$	$\mathbf{30.1}_{\pm 4.5}$	$20.2_{\pm 6.2}$	$26.1_{\pm 5.1}$
MD	LAST	CONC	CONC	$66.0_{\pm 6.3}$	$68.7_{\pm 4.8}$	$63.0_{\pm 3.1}$	$65.1_{\pm 2.0}$	$50.7_{\pm 12.3}$	$52.1_{\pm 2.0}$	$43.9_{\pm 2.3}$	$41.2_{\pm 8.3}$	$14.9_{\pm 2.0}$	$14.1_{\pm 0.9}$	$18.4_{\pm 3.2}$	$32.8_{\pm 2.0}$
MD	LAST	FUNC	NEUT	$46.1_{\pm 4.9}$	$49.0_{\pm 7.5}$	$47.6_{\pm 4.3}$	$47.6_{\pm 7.6}$	$25.6_{\pm 1.8}$	$20.7_{\pm 1.2}$	$17.7_{\pm 0.9}$	$19.7_{\pm 1.4}$	$27.8_{\pm 2.2}$	$19.4_{\pm 1.7}$	$24.6_{\pm 4.3}$	$27.6_{\pm 4.3}$
MD	LAST	FUNC	CONC	$39.7_{\pm 8.2}$	$46.1_{\pm 10.2}$	$48.4_{\pm 6.9}$	$54.3_{\pm 2.6}$	$31.6_{\pm 3.2}$	$28.6_{\pm 9.9}$	$29.6_{\pm 9.4}$	$31.1_{\pm 7.4}$	$14.3_{\pm 3.4}$	$9.5_{\pm 1.2}$	$9.5_{\pm 0.9}$	$17.3_{\pm 2.1}$
MD	AVG	NEUT	NEUT	$\mathbf{73.5}_{\pm 4.9}$	$\mathbf{77.4}_{\pm 2.6}$	$\mathbf{71.2}_{\pm 4.2}$	$\mathbf{74.8}_{\pm 5.1}$	$36.3_{\pm 2.8}$	$52.2_{\pm 7.0}$	$41.3_{\pm 7.7}$	$41.1_{\pm 4.3}$	$24.6_{\pm 4.2}$	$16.3_{\pm 3.9}$	$18.7_{\pm 2.3}$	$28.4_{\pm 3.8}$
MD	AVG	NEUT	CONC	$50.4_{\pm 6.7}$	$49.8_{\pm 7.8}$	$57.9_{\pm 8.3}$	$50.6_{\pm 8.3}$	$29.8_{\pm 5.3}$	$45.6_{\pm 4.2}$	$30.2_{\pm 3.1}$	$31.0_{\pm 2.1}$	$20.6_{\pm 9.4}$	$15.8_{\pm 1.7}$	$20.9_{\pm 10.8}$	$28.3_{\pm 3.6}$
MD	AVG	CONC	NEUT	$50.7_{\pm 7.7}$	$54.0_{\pm 4.2}$	$62.1_{\pm 7.7}$	$56.5_{\pm 6.6}$	$44.5_{\pm 10.9}$	$42.3_{\pm 7.0}$	$\mathbf{57.9}_{\pm 6.9}$	$47.0_{\pm 5.3}$	$28.1_{\pm 3.8}$	$29.2_{\pm 4.1}$	$\mathbf{31.1}_{\pm 9.6}$	$28.9_{\pm 7.1}$
MD	AVG	CONC	CONC	$64.6_{\pm 4.8}$	$66.8_{\pm 4.5}$	$64.7_{\pm 3.8}$	$67.0_{\pm 1.9}$	$\mathbf{60.0}_{\pm 4.8}$	$51.0_{\pm 11.1}$	$42.0_{\pm 3.9}$	$34.2_{\pm 3.3}$	$13.4_{\pm 1.2}$	$13.1_{\pm 1.2}$	$12.8_{\pm 3.6}$	$22.7_{\pm 2.7}$
MD	AVG	FUNC	NEUT	$68.9_{\pm 3.8}$	$72.4_{\pm 4.1}$	$69.6_{\pm 3.9}$	$70.6_{\pm 4.5}$	$29.0_{\pm 11.7}$	$40.1_{\pm 4.0}$	$35.6_{\pm 5.9}$	$50.9_{\pm 13.8}$	$30.8_{\pm 1.9}$	$21.5_{\pm 6.1}$	$21.7_{\pm 4.3}$	$31.0_{\pm 5.8}$
MD	AVG	FUNC	CONC	$46.0_{\pm 4.4}$	$46.7_{\pm 5.9}$	$57.4_{\pm 7.0}$	$55.9_{\pm 8.0}$	$28.5_{\pm 4.6}$	$39.0_{\pm 12.0}$	$32.6_{\pm 6.3}$	$34.6_{\pm 7.8}$	$12.7_{\pm 2.3}$	$13.8_{\pm 2.7}$	$11.3_{\pm 1.1}$	$16.0_{\pm 0.8}$

TABLE III: RQ1 configuration sweep on HumanEval; columns and conventions as in Table II. All numbers are percentages.

				in-distribution				OOD: MBPP+				OOD: synthetic			
method	loc	fit	val/test	Mistral	Qwen	CodeLlama	OpenCoder	Mistral	Qwen	CodeLlama	OpenCoder	Mistral	Qwen	CodeLlama	OpenCoder
Configuration of prior work [11]															
PCA	LAST	NEUT	CONC	$32.3_{\pm 3.8}$	$56.0_{\pm 14.3}$	$34.5_{\pm 9.1}$	$42.2_{\pm 8.8}$	$30.7_{\pm 3.2}$	$27.3_{\pm 7.9}$	$33.3_{\pm 10.3}$	$22.9_{\pm 4.3}$	$34.2_{\pm 6.0}$	$53.4_{\pm 4.9}$	$36.6_{\pm 11.0}$	$32.3_{\pm 17.3}$
Full configuration sweep															
PCA	LAST	NEUT	NEUT	$64.0_{\pm 4.0}$	$68.9_{\pm 8.0}$	$66.5_{\pm 4.7}$	$65.5_{\pm 6.8}$	$54.7_{\pm 4.5}$	$78.1_{\pm 3.0}$	$\mathbf{67.6}_{\pm 3.1}$	$71.0_{\pm 2.7}$	$\mathbf{64.4}_{\pm 5.8}$	$57.3_{\pm 6.9}$	$64.5_{\pm 4.1}$	$69.7_{\pm 3.8}$
PCA	LAST	CONC	NEUT	$43.6_{\pm 10.8}$	$70.8_{\pm 7.8}$	$39.9_{\pm 12.6}$	$59.5_{\pm 12.9}$	$25.3_{\pm 7.9}$	$29.7_{\pm 8.4}$	$29.7_{\pm 11.2}$	$37.8_{\pm 6.2}$	$34.3_{\pm 7.9}$	$31.9_{\pm 17.5}$	$52.7_{\pm 6.7}$	$60.8_{\pm 15.5}$
PCA	LAST	CONC	CONC	$33.5_{\pm 5.2}$	$56.4_{\pm 9.3}$	$42.7_{\pm 9.3}$	$52.3_{\pm 6.6}$	$34.5_{\pm 10.4}$	$31.9_{\pm 15.8}$	$30.1_{\pm 7.6}$	$32.5_{\pm 14.5}$	$33.0_{\pm 3.2}$	$62.4_{\pm 1.9}$	$35.8_{\pm 4.2}$	$57.3_{\pm 2.4}$
PCA	LAST	FUNC	NEUT	$63.5_{\pm 3.3}$	$69.0_{\pm 6.1}$	$66.0_{\pm 3.7}$	$66.0_{\pm 6.2}$	$59.3_{\pm 5.5}$	$79.0_{\pm 3.6}$	$63.3_{\pm 4.2}$	$71.3_{\pm 2.4}$	$54.2_{\pm 10.6}$	$68.8_{\pm 5.1}$	$56.0_{\pm 17.0}$	$26.6_{\pm 24.7}$
PCA	LAST	FUNC	CONC	$30.5_{\pm 3.0}$	$55.2_{\pm 13.9}$	$33.0_{\pm 7.0}$	$40.4_{\pm 9.2}$	$32.0_{\pm 2.4}$	$27.9_{\pm 7.8}$	$32.2_{\pm 7.2}$	$33.7_{\pm 8.4}$	$30.7_{\pm 4.0}$	$61.8_{\pm 3.6}$	$31.0_{\pm 5.5}$	$40.0_{\pm 8.4}$
PCA	AVG	NEUT	NEUT	$35.6_{\pm 5.5}$	$42.1_{\pm 12.7}$	$36.4_{\pm 6.7}$	$51.6_{\pm 9.3}$	$38.3_{\pm 11.9}$	$46.5_{\pm 8.0}$	$41.8_{\pm 6.8}$	$53.7_{\pm 6.8}$	$32.6_{\pm 8.9}$	$30.1_{\pm 11.3}$	$33.1_{\pm 9.8}$	$29.3_{\pm 15.1}$
PCA	AVG	NEUT	CONC	$34.5_{\pm 8.3}$	$49.3_{\pm 9.9}$	$30.0_{\pm 4.7}$	$50.2_{\pm 11.1}$	$28.2_{\pm 4.7}$	$41.5_{\pm 11.9}$	$33.3_{\pm 4.2}$	$33.0_{\pm 8.7}$	$31.6_{\pm 10.0}$	$50.2_{\pm 11.9}$	$32.3_{\pm 8.2}$	$36.4_{\pm 16.7}$
PCA	AVG	CONC	NEUT	$32.4_{\pm 9.7}$	$49.5_{\pm 8.7}$	$40.7_{\pm 9.0}$	$49.4_{\pm 7.5}$	$26.8_{\pm 16.0}$	$40.5_{\pm 4.2}$	$40.8_{\pm 4.3}$	$51.8_{\pm 8.5}$	$34.1_{\pm 3.5}$	$29.6_{\pm 4.6}$	$28.4_{\pm 3.4}$	$35.2_{\pm 5.8}$
PCA	AVG	CONC	CONC	$34.4_{\pm 3.4}$	$58.8_{\pm 8.3}$	$35.5_{\pm 10.0}$	$42.7_{\pm 4.7}$	$30.5_{\pm 7.6}$	$29.8_{\pm 19.6}$	$33.2_{\pm 11.2}$	$34.6_{\pm 16.6}$	$33.2_{\pm 3.2}$	$63.6_{\pm 2.1}$	$30.7_{\pm 3.4}$	$47.4_{\pm 3.4}$
PCA	AVG	FUNC	NEUT	$44.3_{\pm 8.9}$	$59.3_{\pm 14.4}$	$44.7_{\pm 11.0}$	$62.8_{\pm 6.7}$	$41.8_{\pm 10.1}$	$64.9_{\pm 6.5}$	$52.5_{\pm 11.8}$	$56.1_{\pm 7.2}$	$35.1_{\pm 2.8}$	$40.7_{\pm 9.0}$	$36.7_{\pm 3.3}$	$27.9_{\pm 9.8}$
PCA	AVG	FUNC	CONC	$31.8_{\pm 5.4}$	$60.3_{\pm 8.5}$	$36.7_{\pm 5.7}$	$47.9_{\pm 14.1}$	$34.2_{\pm 6.8}$	$58.8_{\pm 4.7}$	$30.7_{\pm 8.8}$	$34.9_{\pm 10.7}$	$31.2_{\pm 2.7}$	$66.9_{\pm 3.1}$	$25.3_{\pm 7.7}$	$42.0_{\pm 7.6}$
MD	LAST	NEUT	NEUT	$64.6_{\pm 3.0}$	$74.5_{\pm 9.2}$	$68.0_{\pm 3.3}$	$65.4_{\pm 6.1}$	$55.1_{\pm 3.6}$	$79.1_{\pm 2.9}$	$67.2_{\pm 3.2}$	$71.0_{\pm 2.7}$	$60.9_{\pm 8.3}$	$59.5_{\pm 9.3}$	$\mathbf{65.1}_{\pm 3.6}$	$\mathbf{70.5}_{\pm 4.2}$
MD	LAST	NEUT	CONC	$34.1_{\pm 6.6}$	$67.9_{\pm 5.8}$	$36.4_{\pm 5.6}$	$41.2_{\pm 10.2}$	$31.2_{\pm 4.1}$	$34.0_{\pm 9.6}$	$31.9_{\pm 9.3}$	$23.7_{\pm 4.5}$	$35.7_{\pm 4.6}$	$54.4_{\pm 5.8}$	$28.5_{\pm 7.1}$	$49.0_{\pm 3.9}$
MD	LAST	CONC	NEUT	$45.5_{\pm 12.7}$	$\mathbf{79.4}_{\pm 10.5}$	$50.7_{\pm 12.5}$	$58.5_{\pm 12.0}$	$29.8_{\pm 9.1}$	$47.4_{\pm 10.8}$	$45.1_{\pm 12.5}$	$45.5_{\pm 14.2}$	$22.6_{\pm 8.9}$	$23.0_{\pm 5.5}$	$44.4_{\pm 19.3}$	$60.6_{\pm 14.7}$
MD	LAST	CONC	CONC	$39.4_{\pm 6.3}$	$61.3_{\pm 8.2}$	$49.4_{\pm 6.0}$	$54.0_{\pm 4.5}$	$44.9_{\pm 4.2}$	$51.4_{\pm 3.8}$	$40.8_{\pm 9.6}$	$37.0_{\pm 16.1}$	$33.1_{\pm 3.0}$	$62.5_{\pm 2.0}$	$35.5_{\pm 4.7}$	$57.5_{\pm 2.5}$
MD	LAST	FUNC	NEUT	$\mathbf{64.7}_{\pm 4.5}$	$68.7_{\pm 8.3}$	$\mathbf{69.8}_{\pm 1.6}$	$65.0_{\pm 5.2}$	$\mathbf{59.9}_{\pm 5.0}$	$\mathbf{79.1}_{\pm 3.2}$	$63.0_{\pm 4.0}$	$\mathbf{71.3}_{\pm 2.4}$	$53.6_{\pm 5.6}$	$\mathbf{71.7}_{\pm 3.1}$	$63.8_{\pm 3.1}$	$36.3_{\pm 25.7}$
MD	LAST	FUNC	CONC	$32.4_{\pm 7.1}$	$59.8_{\pm 10.6}$	$35.4_{\pm 6.0}$	$41.8_{\pm 7.6}$	$32.6_{\pm 3.6}$	$30.7_{\pm 6.7}$	$38.9_{\pm 4.9}$	$32.1_{\pm 8.5}$	$31.3_{\pm 3.3}$	$62.9_{\pm 2.7}$	$24.5_{\pm 3.4}$	$40.0_{\pm 5.9}$
MD	AVG	NEUT	NEUT	$49.8_{\pm 5.5}$	$58.2_{\pm 9.7}$	$51.6_{\pm 7.3}$	$63.7_{\pm 8.8}$	$32.3_{\pm 7.8}$	$50.4_{\pm 4.3}$	$49.5_{\pm 10.2}$	$47.0_{\pm 3.3}$	$35.1_{\pm 5.8}$	$33.4_{\pm 11.4}$	$33.3_{\pm 4.2}$	$29.0_{\pm 13.9}$
MD	AVG	NEUT	CONC	$29.9_{\pm 3.3}$	$68.4_{\pm 13.4}$	$32.1_{\pm 8.0}$	$49.3_{\pm 10.0}$	$26.9_{\pm 6.5}$	$32.6_{\pm 12.6}$	$37.9_{\pm 8.0}$	$33.2_{\pm 5.8}$	$34.8_{\pm 4.4}$	$57.2_{\pm 6.4}$	$32.6_{\pm 6.5}$	$25.3_{\pm 14.5}$
MD	AVG	CONC	NEUT	$34.5_{\pm 7.9}$	$56.8_{\pm 5.1}$	$40.9_{\pm 7.4}$	$49.3_{\pm 10.8}$	$36.4_{\pm 11.6}$	$42.6_{\pm 3.5}$	$41.9_{\pm 8.4}$	$46.5_{\pm 15.5}$	$33.2_{\pm 5.5}$	$27.0_{\pm 4.0}$	$25.3_{\pm 5.3}$	$33.8_{\pm 6.3}$
MD	AVG	CONC	CONC	$45.0_{\pm 9.0}$	$63.6_{\pm 11.4}$	$47.8_{\pm 3.8}$	$54.3_{\pm 8.5}$	$43.6_{\pm 8.6}$	$28.4_{\pm 12.0}$	$47.5_{\pm 4.5}$	$26.5_{\pm 4.7}$	$32.7_{\pm 3.7}$	$63.7_{\pm 2.3}$	$30.9_{\pm 3.9}$	$47.1_{\pm 3.3}$
MD	AVG	FUNC	NEUT	$48.8_{\pm 5.7}$	$64.0_{\pm 12.5}$	$49.8_{\pm 6.0}$	$\mathbf{66.8}_{\pm 8.1}$	$38.2_{\pm 9.0}$	$71.5_{\pm 4.8}$	$48.2_{\pm 10.6}$	$58.5_{\pm 9.7}$	$38.0_{\pm 5.4}$	$45.3_{\pm 14.8}$	$35.9_{\pm 4.0}$	$36.4_{\pm 17.1}$
MD	AVG	FUNC	CONC	$30.7_{\pm 5.8}$	$61.2_{\pm 8.6}$	$35.0_{\pm 8.6}$	$54.0_{\pm 10.1}$	$29.8_{\pm 5.0}$	$46.0_{\pm 11.6}$	$35.7_{\pm 6.6}$	$37.7_{\pm 11.2}$	$32.1_{\pm 3.1}$	$66.9_{\pm 3.0}$	$24.1_{\pm 4.7}$	$45.0_{\pm 3.8}$

dropped to the 25% chance level of the 1-of-4 selection: a poorly matched fitting source does not merely weaken the correctness signal but can reduce it to chance.

4) Construction, data, and model: The decomposition so far has more terms than can be compared across the two benchmarks at a glance. Essentially, producing a direction takes three different components, and every factor in our work can be attributed to one of them: (1) how the direction is *constructed*: the method, read-out location, and framing; (2) what *data* it is fit on: in-distribution and out-of-distribution; (3) which *model* it is read from: 4 LLMs which are neither a construction choice nor part of the data. These three groups comprise everything, and panel 4 of Table IV reports them per benchmark as before. On BigCodeBench the fitting data carries 75% of the variance, construction 16%, and the model 9%. On HumanEval the order is different: construction carries 42%, the model 40%, and the fitting data 18%. The kind of choice that carries the most variance is therefore not the same on the two benchmarks: on BigCodeBench it is the data the direction is fit on, and on HumanEval it is how the direction is built and which model it is read from.

TABLE IV: ANOVA for RQ1. η^2 = variance share in Acc_{Val} from a factor (main effect) or an interaction. Panels (1)–(4) follow Sections IV-1–IV-4

(1) General View			η^2	
main effects				
2. model identity			14%	
6. testing framing (test)			7%	
7. construction method			7%	
22. benchmark			<1%	
24. read-out location (loc)			<1%	
28. fitting framing (fit)			<1%	
interactions				
1. fit × test framing			15%	
3. loc × benchmark			12%	
4. loc × test framing × benchmark			11%	
5. benchmark × model			7%	
any term involving benchmark			47%	
	BigCodeBench		HumanEval	
	η^2	best	η^2	best
(2) Within each benchmark				
read-out location	20%	avg	6%	last
construction method	14%	MD	3%	MD
fitting framing	5%	conc	1%	func
testing framing	<1%	—	20%	neut
model identity	11%	agree	51%	Qwen
fit × test framing	26%		7%	
loc × test framing	15%		8%	
(3) Out-of-distribution fitting			*mean acc.*	
in-distribution	52%		50%	
MBPP+	35%		43%	
synthetic	22%		42%	
(4) Construction, data, and model				
construction	16%		42%	
fitting data	75%		18%	
model	9%		40%	

RQ1: Key takeaway

There is no single best configuration for capturing a code correctness direction. Across the reported sweeps, mean-diff has higher average accuracy than PCA on both benchmarks, but it does not win in every configuration. Read-out location, framing, and model also vary across benchmarks, so which *kind* of choice matters most is benchmark-dependent: on BigCodeBench the fitting-data group accounts for most of the variance (75%), while on HumanEval it splits between construction (42%) and model (40%). These benchmark-specific variance shares summarize the observed results; they do not separate the effects of fitting-source content and size.

V. RQ2: Does controlling for differences between correct/incorrect programs improve the quality of the correctness direction?

Table V compares directions fit on *controlled pairs*: the incorrect program is constructed by mutating a correct solution, rather than independently generated as in the benchmark candidate sets. It reports M-Only, whose incorrect member is a single-fault mutation, and MC, which adds a behaviour-preserving rewrite (Section III-F). The upper block shows RQ1's mean-diff results on the same benchmark test sets; the lower blocks fit on each benchmark's controlled pairs. Their final column is the Isolation setting; the other columns are Transfer (Section III-F). All cells average over read-out locations, framings, and models; they show Acc_{Val} with best-layer accuracy in parentheses (Section III-A). For the lower blocks, Acc_{Val} selects the layer using a held-out split of the fitting pairs. We separately report the effect of selecting it on benchmark validation data in Section V-6. Table VI checks whether the observed pattern changes with PCA or paired sampling.

TABLE V: RQ2 transfer: mean-diff direction fit on a benchmark's controlled pairs (M-Only/MC), test on BigCodeBench, HumanEval, and held-out split of those pairs. Cells are Acc_{Val} (best layer in parentheses), averaged over read-out locations, framings, and models. All columns report 1-of-4 selection accuracy. The top block is RQ1's mean-diff on the same benchmark test sets; each fit uses its own fitting and validation data. Dashes mark conditions not run; † fits on one benchmark, tests on the other.

	tested on		
fit source	BigCodeBench	HumanEval	held-out
RQ1 sources			
in-distribution	56.7 (61.6)	52.8 (61.7)	—
MBPP+	35.2 (49.2)	44.4 (57.7)	—
synthetic	21.5 (40.5)	42.6 (53.3)	—
fit on BigCodeBench controlled pairs			
M-Only	19.6 (36.8)	29.8 (43.1)†	83.3 (86.4)
MC	19.0 (36.7)	30.0 (43.0)†	81.7 (84.3)
fit on HumanEval controlled pairs			
M-Only	22.2 (38.9)†	47.6 (57.3)	71.6 (78.9)
MC	21.3 (37.1)†	48.5 (57.9)	71.1 (78.5)

TABLE VI: RQ2 robustness (M-Only). Acc_{Val} for a direction fit on a benchmark's controlled pairs, tested in isolation (a held-out split of those pairs) and in transfer (the benchmark), across construction method and sampling. In the reported settings, isolation accuracy is high and transfer accuracy low across the tested variants; paired sampling gives higher isolation accuracy but similar transfer accuracy. MC behaves similarly (Table V).

		BigCodeBench		HumanEval	
method	sampling	isol.	transf.	isol.	transf.
mean-diff	weighted	83.3	19.6	71.6	47.6
mean-diff	paired	85.9	19.1	78.4	48.0
PCA	weighted	81.4	20.0	65.1	46.1
PCA	paired	83.2	19.7	73.0	44.7

1) M-Only pairs are separable in isolation: When fit, validated, and tested on held-out splits of the same controlled-pair data, the M-Only direction reaches 83.3% on BigCodeBench and 71.6% on HumanEval (86.4% and 78.9% at the best layer). On these held-out controlled pairs, the direction distinguishes mutations from their source solutions. This result only shows separation within the constructed data; it does not show that the mutations represent independently generated incorrect programs. The held-out accuracy is lower on HumanEval than on BigCodeBench; the mutant pools also differ in size (4,566 and 27,928, respectively). The following comparisons ask whether this within-source separation transfers to the benchmark candidate sets.

2) Transfer to BigCodeBench: For M-Only, fitting and validation use controlled pairs; the direction scores 19.6% on held-out BigCodeBench candidate sets. RQ1's in-distribution fit scores 56.7% on those same held-out sets,

a difference of 37.1 points. Each fit uses its own fitting and validation data. The M-Only fit is also below the MBPP+ and synthetic RQ1 fits (35.2% and 21.5%), although these comparisons use different fitting data. The contrast with the RQ1 fit shows that separability on controlled pairs does not, in this setting, ensure effective ranking of independently generated BigCodeBench candidates.

3) Transfer to HumanEval: Fit and validated on HumanEval's M-Only pairs and tested on HumanEval, the accuracy reaches 47.6%, compared with 52.8% for RQ1's in-distribution fit on HumanEval, a difference of 5.2 points, smaller than the 37.1-point difference on BigCodeBench. The observed transfer gap is therefore larger on BigCodeBench than on HumanEval in these comparisons. Because both the fitting source and the origin of the incorrect programs change, this contrast does not isolate the effect of fault isolation.

4) Crossing benchmarks: Fitting on one benchmark's controlled pairs and testing on the other (the daggered cells) changes both the fitting source and test benchmark, so these comparisons do not isolate the effect of fault construction. In the reported settings, fitting on BigCodeBench pairs and testing on HumanEval gives 29.8%, while the reverse gives 22.2%. This asymmetry appears in these comparisons, but we do not establish its cause.

5) Direction quality: On BigCodeBench, the M-Only fit's validation-selected and test-oracle accuracies are 19.6% and 36.8%, a 17.2-point difference. For the in-distribution fit, they are 56.7% and 61.6%, a 4.9-point difference. Selecting the best layer on the test set gives 36.8%. Even at this oracle layer, the M-Only result is 24.8 points below the in-distribution result. This comparison changes the fitting source, so it describes performance under the two sources rather than attributing the remaining difference to a single cause. On HumanEval, the validation-selected and test-oracle M-Only results are 47.6% and 57.3%; the latter is 4.4 points below the in-distribution oracle result of 61.7%.

6) Selecting the layer on the benchmark: To assess layer selection separately, we select the layer using a held-out benchmark validation split rather than the controlled-pair validation split (Transfer* in Figure 1). On BigCodeBench, this raises the M-Only result from 19.6% to 32.9%, accounting for 13.3 points of the 17.2-point difference to the test-oracle result (36.8%). The test oracle is not a deployable selection procedure. On HumanEval, benchmark validation changes the M-Only result from 47.6% to 48.6%. These comparisons show that the measured effect of layer selection differs between the two benchmarks; they do not establish why.

7) Adding controlled confounds: The MC variant applies behaviour-preserving rewrites, e.g. variable renaming, so that the two implementations differ in surface form while the fault stays fixed (Section III-F). In the reported settings, MC tracks M-Only. It is comparably separable in isolation (81.7% on BigCodeBench and 71.1% on HumanEval, against 83.3% and 71.6%), its transfer accuracy is just as low (19.0% on BigCodeBench and 48.5% on HumanEval, against 19.6% and 47.6%), and it changes similarly when the layer is selected on benchmark validation (19.0% to 32.7% on BigCodeBench). The held-out isolation results use different test pairs for M-Only and MC, so they are not a same-test-set comparison. These results show that this set of controlled rewrites does not close the observed BigCodeBench transfer gap; they do not rule out effects from other surface features.

8) Robustness to construction and sampling: The low transfer accuracy persists across the construction and sampling variants tested here. Table VI repeats each same-benchmark fit with PCA in place of mean-diff and with paired in place of weighted sampling, reporting both isolation and transfer. In transfer the accuracy stays within a narrow band on each benchmark — roughly 19–20% on BigCodeBench and 45–49% on HumanEval across all four method–sampling combinations (M-Only shown; MC is comparable, Table V). In isolation, each reported combination stays highly separable, and paired sampling, which fits on every available pair rather than one per task, separates the held-out pairs better than weighted sampling (M-Only 85.9% against 83.3% on BigCodeBench, 78.4% against 71.6% on HumanEval). The higher isolation accuracy with paired sampling does not correspond to higher transfer accuracy in these comparisons.

RQ2: Key takeaway

Directions fit on controlled pairs separate held-out mutations from their source solutions (83.3% on BigCodeBench and 71.6% on HumanEval), but this does not carry over to ranking independently generated benchmark candidates: M-Only reaches 19.6% on BigCodeBench and 47.6% on HumanEval, against 56.7% and 52.8% for the in-distribution fit on the same candidates. MC and the tested method/sampling variants show a similar pattern. This gap does not mean that correctness signals are absent; rather, the direction is best fit on data resembling the candidates it will rank.

VI. Discussion

A. The nature of the captured direction

Our findings suggest that a code correctness direction is not a single property of a model: its accuracy, and which configuration recovers it best, depend on how the direction is constructed and what it is contrasted against. RQ1 varies the construction: mean-diff has higher average accuracy than PCA across the two benchmarks in the reported sweeps, though not in every configuration; the preferred read-out location, framing, and model vary (Section IV). RQ2 varies the fitting data. A direction fit on controlled pairs separates held-out mutations from their source solutions (83.3% on BigCodeBench and 71.6% on HumanEval), but its accuracy on independently generated

BigCodeBench candidates is 19.6%, compared with 56.7% for the in-distribution fit. The difference is smaller on HumanEval (47.6% compared with 52.8%). Paired sampling raises accuracy on held-out mutations but does not improve transfer. These results show that performance on one contrast set does not establish performance on another. They qualify broad readings of internal "truth" or "correctness" directions [12], [13], [15]: the measured accuracy depends on the configuration and the data used, not on the model alone. This echoes the truthfulness literature, where some report a near-universal hyperplane [37] and others uneven generalization across logical transformations [38] or a graded property rather than a single boundary [39].

B. Controlling the fitting data

We expected that removing incidental differences between a correct program and a failing attempt would improve the direction's transfer. The reported results do not show that improvement. On held-out BigCodeBench candidate sets, the M-Only fit scores 19.6%, while the in-distribution fit scores 56.7% on those same sets. On held-out HumanEval candidate sets, the corresponding scores are 47.6% and 52.8%. Because the data also differs in composition and size, these accuracy differences do not show that fault isolation caused the results.

Selecting the layer with benchmark validation data raises the BigCodeBench M-Only result from 19.6% to 32.9%; selecting the best layer on the test set gives 36.8%. The latter remains 24.8 points below the 61.6% in-distribution oracle result. This gap compares directions fit on different data and is not an estimate of a causal loss in direction quality. On HumanEval, the same benchmark-validation selection changes the M-Only result from 47.6% to 48.6%.

The MC variant adds behaviour-preserving rewrites to the mutations, but its transfer results are similar to M-Only (19.0% on BigCodeBench and 48.5% on HumanEval). This shows that the tested rewrites do not close the observed transfer gap. The high accuracy on held-out mutations shows separation within that constructed data; it does not establish that the mutations are realistic or that correctness information is absent when ranking independent candidates.

C. Practical use of the technique

The direction could help a tool rank generated candidates so developers can decide which to inspect or test first; it is not a replacement for tests or review. A tool builder could fit a direction on labelled correct and incorrect examples, choose the construction method, framing, read-out location, and layer using representative validation tasks, then assess the selected setup on held-out tasks. Our experiments report the held-out accuracy of every method, framing, and read-out combination, selecting the layer on validation data for each. A deployed tool would choose one of these combinations using validation data and keep it fixed for testing and subsequent use. At run time, the selected direction scores new candidates without retraining the LLM; its accuracy is most likely to hold when those candidates resemble the workload represented by the fitting and validation data.

Across our reported sweeps, mean-diff has higher average accuracy than PCA on both benchmarks, though not in every configuration, making it a reasonable starting point rather than a universal choice (Section IV). The read-out location and framing still depend on the benchmark. Best-layer accuracy uses test labels to choose the layer, so it is a diagnostic upper bound, not a deployable result. Instead, validation-selected performance should be reported and the selected setup should be evaluated on held-out tasks.

VII. Threats to Validity

Construct Validity. Using unit tests as a proxy for correctness may be incomplete and does not capture non-functional aspects, *e.g.*, performance. Moreover, high probing accuracy alone is weak evidence that the captured direction encodes correctness and not a correlated feature [28]: an expressive probe may fit labels by memorising patterns in the data rather than detecting the intended property [29], score highly on a property the model does not use [30], and prove unreliable for concept removal and detection [31], [32]. We use single linear directions, limiting probe complexity, and do not rely on a single configuration: we vary the construction method, framing, and read-out location, select the layer on a validation split, and test on two benchmarks with cross-validation. RQ2 further controls the fitting data with mutations that isolate the fault and behaviour-preserving rewrites that vary only surface form, to test whether the observed transfer changes with the fitting data.

Internal Validity. RQ2 compares directions fit on mutations of canonical solutions with directions fit on the pre-generated failing programs used in RQ1. The mutation pipeline checks types but not whether each edit is semantically plausible, so some mutants may not resemble realistic faults. High accuracy on held-out mutants shows separation within this constructed data; it does not establish their realism or explain the transfer difference. Prior work likewise reports that injected faults and detectors trained on them may transfer poorly to realistic faults [33]–[35], and that models complete code containing injected faults less reliably [36]. These findings reinforce the need to treat mutations as a controlled probe rather than a direct substitute for realistic failing attempts. The fitting sources also differ in content and size, so the comparison does not isolate the effect of fault isolation. The model used to generate the rewrites, `Qwen2.5-Coder-32B-Instruct`, belongs to the same family as one probed model, so similarities in style could bias the results. The aggregate MC and M-Only results are similar (Section V-7), but this comparison does not rule out a model-specific effect.

External Validity. We study four 7–8B-parameter LLMs on two Python benchmarks, covering in-distribution and out-of-distribution stimuli. RQ1's in-distribution fit/validation/test splits contain 45/45/367 BigCodeBench tasks and 15/15/121 HumanEval tasks. Each OOD fit/validation split contains 24 MBPP+ tasks or five synthetic tasks (96 and 20 unique tasks, respectively, across four folds). The RQ2 mutant pools also differ in size: 27,928 for BigCodeBench and 4,566 for HumanEval. These differences in sample size and task content may affect direction fitting, validation-based layer selection, and accuracy estimates. Cross-benchmark comparisons therefore reflect both benchmark content and sample size, while within-benchmark comparisons use the same test tasks. The observed effects may not hold for larger or smaller models, other architectures, or other programming languages, and our single-function tasks do not cover repository-level settings such as SWE-bench [3]. We include both general and code models, benchmarks of differing complexity, and out-of-distribution data, but these may not establish broader generality. For RQ1, the reported η^2 values describe variance shares, not causal effects of source or sample size. The fitting-data group accounts for 75% of the variance on BigCodeBench and 18% on HumanEval; these are benchmark-specific decompositions, not a pooled result.

Conclusion Validity. The tables report means, fold-level standard deviations, and η^2 shares. RQ1 asks which factors drive accuracy across the configuration space, so we use ANOVA to attribute the variance of fold-averaged $\mathrm{Acc}_{\mathrm{Val}}$ values to factors, not to test individual differences; nor do we test RQ2 accuracy differences for significance, such as those between controlled-pair and in-distribution fitting on the same benchmark candidates. Thus, our results report effect sizes and fold-level variability, not statistical significance.

VIII. Related Work

Prior work introduced the setting we build on: reading a model's hidden states to judge the correctness of generated code without executing it [11]. Bui et al. [40] report that internal representations correlate strongly with whether a program passes its tests. Vu et al. [41] raise transferability by choosing the read-out layer and token per model rather than fixing them, while others read the same signal before generation to predict failure [42], as a risk score over internal states [43], or as span-localized calibrated uncertainty [44]. Mechanistic accounts ask where and how information about code correctness is represented inside the model, tracing it through sparse autoencoders [45] and attribution graphs [46], extending evidence that code models encode program semantics internally [47], while a parallel security line reads or steers a vulnerability direction in the same space [48]–[51]. Each of these fixes one construction of the direction and inherits whatever correct and incorrect programs its benchmark supplies; we instead measure how much the read-out location, construction method, and prompt framing change the signal, and find that the best location reverses between HumanEval and BigCodeBench. The closest contrast, Vu et al.'s per-model selection [41], sidesteps this choice rather than quantifying its cost.

The probing recipe itself originates outside code, where hidden states linearly separate true from false statements [12], [13] and representation engineering generalises it to reading and steering many concepts [15]. The same construction has since been applied to a model's own outputs: probing reasoning traces for self-verification [52], predicting answer accuracy before any answer is produced [53], and reading correctness from class centroids in the spirit of mean-diff [54]. We bring this line to the correctness of generated code.

Our second question needs correct and incorrect programs that differ only in the fault-causing change, which mutation testing supplies by seeding single localised faults into working code [26]. Behaviour-preserving rewrites such as renaming and reformatting are the standard control for surface form [55]–[57]. We combine both: mutations isolate the fault, behaviour-preserving rewrites vary surface form. In our previous work [58], we prompted an LLM to inject one semantic error from a fixed taxonomy into HumanEval and BigCodeBench solutions, discarded the variants that still pass the tests, and released 1,217 labelled pairs. In this paper, we chose to construct our own pairs for two reasons: **(1)** LLM injection leaves residual label noise, whereas our deterministic operators act on a typed AST, so each mutation is localised and type-compatible by construction; **(2)** the MC variant needs a behaviour-preserving rewrite *of each mutant* (Section III-F), and no released dataset we know of pairs its faults with such rewrites.

IX. Conclusion

We studied how construction method, prompt framing, and hidden-state location affect code-correctness directions (RQ1), and whether directions fit on controlled mutations transfer to independently generated candidates (RQ2). Across the reported sweeps, no configuration is best in every setting; mean-diff has higher average accuracy than PCA on both benchmarks, but not in every configuration. Directions fit on controlled mutations separate held-out mutations, yet do not improve candidate selection in the tested settings. Since the fitting sources differ in content and size, we cannot attribute this result to fault isolation alone. The results support treating the signal as a potential workload-specific aid for prioritizing candidates, not as a general correctness guarantee. Future work can test whether more realistic fault data improve transfer, identify which internal features support ranking, and examine properties such as efficiency and security.

References

[1] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: code generation using transformer," in *Pro-*

ceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2020, pp. 1433–1443.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[3] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can language models resolve real-world GitHub issues?" in *The Twelfth International Conference on Learning Representations*, 2024.

[4] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, 2023.

[5] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, "Security weaknesses of copilot-generated code in GitHub projects: An empirical study," *ACM Transactions on Software Engineering and Methodology*, 2025.

[6] Z. Wang, Z. Zhou, D. Song, Y. Huang, S. Chen, L. Ma, and T. Zhang, "Towards understanding the characteristics of code generation errors made by large language models," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 2587–2599.

[7] A. Gu, W.-D. Li, N. Jain, T. X. Olausson, C. Lee, K. Sen, and A. Solar-Lezama, "The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations?" in *Findings of the Association for Computational Linguistics: ACL 2024*, 2024, pp. 74–117.

[8] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[9] K. Tian, E. Mitchell, A. Zhou, A. Sharma, R. Rafailov, H. Yao, C. Finn, and C. D. Manning, "Just ask for calibration: Strategies for eliciting calibrated confidence scores from language models fine-tuned with human feedback," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023, pp. 5433–5442.

[10] C. Spiess, D. Gros, K. S. Pai, M. Pradel, M. R. I. Rabin, A. Alipour, S. Jha, P. Devanbu, and T. Ahmed, "Calibration and correctness of language models for code," in *International Conference on Software Engineering (ICSE)*, 2025.

[11] F. Ribeiro, C. Spiess, P. Devanbu, and S. Nadi, "On LLMs' internal representation of code correctness," in *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, 2026.

[12] A. Azaria and T. Mitchell, "The internal state of an LLM knows when it's lying," in *Findings of the Association for Computational Linguistics: EMNLP*, 2023.

[13] S. Marks and M. Tegmark, "The geometry of truth: Emergent linear structure in large language model representations of true/false datasets," *arXiv preprint arXiv:2310.06824*, 2023.

[14] C. Burns, H. Ye, D. Klein, and J. Steinhardt, "Discovering latent knowledge in language models without supervision," in *International Conference on Learning Representations (ICLR)*, 2023.

[15] A. Zou, L. Phan, S. Chen, J. Campbell, P. Guo, R. Ren, A. Pan, X. Yin, M. Mazeika, A.-K. Dombrowski *et al.*, "Representation engineering: A top-down approach to AI transparency," *arXiv preprint arXiv:2310.01405*, 2023.

[16] S. Im and S. Li, "A unified understanding and evaluation of steering methods," 2026. [Online]. Available: https://arxiv.org/abs/2502.02716

[17] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul *et al.*, "BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions," *arXiv preprint arXiv:2406.15877*, 2024.

[18] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[19] 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 *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[21] K. Li, O. Patel, F. Viégas, H. Pfister, and M. Wattenberg, "Inference-time intervention: Eliciting truthful answers from a language model," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[22] H. Abdi and L. J. Williams, "Principal component analysis," *WIREs Computational Statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[23] S. Kadavath, T. Conerly, A. Askell, T. Henighan, D. Drain, E. Perez, N. Schiefer, Z. Hatfield-Dodds, N. DasSarma, E. Tran-Johnson *et al.*, "Language models (mostly) know what they know," *arXiv preprint arXiv:2207.05221*, 2022.

[24] Z. Sun, L. Li, Y. Liu, X. Du, B. Wang, J. Zhu, C. Wang, and Y. Liu, "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)*, 2024, pp. 229–241.

[25] Logilab and PyCQA, "astroid: A common base representation of python source code," https://github.com/pylint-dev/astroid, 2024.

[26] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[27] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2.5-Coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[28] J. Liu, S. Chen, Y. Cheng, and J. He, "On the universal truthfulness hyperplane inside LLMs," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024.

[29] Y. Bao, X. Zhang, T. Du, X. Zhao, Z. Feng, H. Peng, and J. Yin, "Probing the geometry of truth: Consistency and generalization of truth directions in LLMs across logical transformations and question answering tasks," in *Findings of the Association for Computational Linguistics: ACL*, 2025.

[30] Z. Ying, S. Ravfogel, N. Kriegeskorte, and P. Hase, "The truthfulness spectrum hypothesis," *arXiv preprint arXiv:2602.20273*, 2026.

[31] Y. Belinkov, "Probing classifiers: Promises, shortcomings, and advances," *Computational Linguistics*, vol. 48, no. 1, pp. 207–219, 2022.

[32] J. Hewitt and P. Liang, "Designing and interpreting probes with control tasks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019.

[33] A. Ravichander, Y. Belinkov, and E. Hovy, "Probing the probing paradigm: Does probing accuracy entail task relevance?" in *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, 2021.

[34] Y. Elazar, S. Ravfogel, A. Jacovi, and Y. Goldberg, "Amnesic probing: Behavioral explanation with amnesic counterfactuals," *Transactions of the Association for Computational Linguistics*, vol. 9, pp. 160–175, 2021.

[35] A. Kumar, C. Tan, and A. Sharma, "Probing classifiers are unreliable for concept removal and detection," *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[36] J. Patra and M. Pradel, "Semantic bug seeding: A learning-based approach for creating realistic bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[37] C. Richter and H. Wehrheim, "Learning realistic mutations: Bug creation for neural bug detectors," in *2022, IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022.

[38] J. He, L. Beurer-Kellner, and M. Vechev, "On distribution shift in learning-based bug detectors," in *International Conference on Machine Learning (ICML)*, 2022.

[39] T. Dinh, J. Zhao, S. Tan, R. Negrinho, L. Lausen, S. Zha, and G. Karypis, "Large language models of code fail at completing code with potential bugs," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[40] T.-D. Bui, T. T. Vu, T.-T. Nguyen, S. Nguyen, and H. Vo, "Correctness assessment of code generated by large language models using internal representations," *Journal of Systems and Software*, vol. 230, p. 112570, 2025.

[41] T. T. Vu, T.-D. Bui, T.-T. Nguyen, S. Nguyen, and H. Vo, "Model-agnostic quality assessment for LLM-generated code via dynamic internal representation selection," *Journal of Systems and Software*, vol. 237, p. 112852, 2026.

[42] W. Lugoloobi, T. Foster, W. Bankes, and C. Russell, "LLMs encode their failures: Predicting success from pre-generation activations," *arXiv preprint arXiv:2602.09924*, 2026.

[43] Y. Huang, L. Ma, K. Nishikino, and T. Akazaki, "Risk assessment framework for code LLMs via leveraging internal states," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE)*, 2025.

[44] D. Gros and P. Devanbu, "Localized calibrated uncertainty in code language models," *arXiv preprint arXiv:2512.24560*, 2025.

[45] K. Tahimic and C. Cheng, "Mechanistic interpretability of code correctness in llms via sparse autoencoders," 2025. [Online]. Available: https://arxiv.org/abs/2510.02917

[46] Y. He, Z. Zhao, K. Zhou, B. Dai, J. Fu, and Y. Yang, "CodeCircuit: Toward inferring LLM-generated code correctness via attribution graphs," *arXiv preprint arXiv:2602.07080*, 2026.

[47] C. Jin and M. Rinard, "Emergent representations of program semantics in language models trained on programs," ser. ICML'24. JMLR.org, 2024.

[48] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[49] W. Yu, R. Mangal, T. Zhuo, M. Fredrikson, and C. S. Pasareanu, "A mixture of linear corrections generates secure code," 2025. [Online]. Available: https://arxiv.org/abs/2507.09508

[50] J. Li, L. Cui, J. Zhang, H. Fei, Y. Chen, and H. Zhu, "Steering large language models for vulnerability detection," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2025.

[51] R. Melo, C. Mamede, A. Catarino, R. Abreu, and H. L. Cardoso, "Are sparse autoencoders useful for java function bug detection?" 2026. [Online]. Available: https://arxiv.org/abs/2505.10375

[52] A. Zhang, Y. Chen, J. Pan, C. Zhao, A. Panda, J. Li, and H. He, "Reasoning models know when they're right: Probing hidden states for self-verification," *arXiv preprint arXiv:2504.05419*, 2025.

[53] I. V. M. Cencerrado, A. P. Masdemont, A. G. Hawthorne, D. D. Africa, and L. Pacchiardi, "No answer needed: Predicting llm answer accuracy from question-only linear probes," 2026. [Online]. Available: https://arxiv.org/abs/2509.10625

[54] S. Cho, Z. Wu, K. D. Costa, and A. Koshiyama, "The confidence manifold: Geometric structure of correctness representations in language models," 2026. [Online]. Available: https://arxiv.org/abs/2602.08159

[55] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia, R. Nallapati, M. K. Ramanathan, D. Roth, and B. Xiang, "ReCode: Robustness evaluation of code generation models," in *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023.

[56] J. Henkel, G. Ramakrishnan, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 526–537.

[57] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. Gonzalez, and I. Stoica, "Contrastive code representation learning," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, Eds. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 5954–5971. [Online]. Available: https://aclanthology.org/2021.emnlp-main.482/

[58] M. Kassem, F. Ribeiro, and S. Nadi, "An automated methodology for generating labeled datasets of semantic errors in code," in *Proceedings of the 3rd International Workshop on Large Language Models for Code (LLM4Code)*, ser. LLM4Code '26. New York, NY, USA: Association for Computing Machinery, 2026, pp. 14–20.