

20 Questions for Code: Improving Code Generation with Information-Theoretic Clarification

Stanford CS224N Custom Project

Ria Garg
Dept. of Computer Science
Stanford University
riagarg@stanford.edu

Alexandra Kim
Dept. of Computer Science
Stanford University
alexskim@stanford.edu

Julia Xi
Dept. of Computer Science
Stanford University
juliaxi@stanford.edu

Abstract

Large language models (LLMs) have become astonishingly good at generating working code. Yet even when LLM-generated code appears correct and passes basic sanity checks, it can fail on edge cases when natural language prompts are ambiguous or incomplete, leading to 20 – 40% absolute drops in pass@1 across state-of-the-art code models. Human programmers often resolve this uncertainty by asking follow-up questions before writing code, but LLMs often make assumptions about a user’s intent, producing code misaligned with the task at hand. We investigate whether cheap code models can be improved by asking binary yes/no questions in the form of unit tests before generating code. We build a fully automated pipeline that samples candidate programs, ranks candidate tests by expected information gain (EIG), queries a reference implementation, and stops when another question is no longer worth its cost. On HumanEval and MBPP prompts mutated to introduce uncertainty, EIG-based questioning outperforms one-shot generation and other interactive baselines. For Qwen 2.5 7B, EIG-based questioning improves pass@1 over the one-shot baseline from 0.442 to 0.649 on HumanEval+ and 0.547 to 0.723 on MBPP+; for Llama 3.1 8B, it improves pass@1 from 0.419 to 0.627 on HumanEval+ and 0.501 to 0.698 on MBPP+. Gains are largest on ambiguous and incomplete prompts, while contradictory prompts remain challenging. Comparisons with frontier LLMs show that interactive small models can compete with the performance of much more expensive LLMs: on HumanEval+, Gemini 3 Flash Thinking achieves a pass@1 score of 0.683 using 21k tokens, while Qwen with our EIG strategy reaches a pass@1 score of 0.649 using roughly 4k tokens, resulting in a 5x reduction in tokens and an 80% reduction in per-token inference cost. Our results suggest that a few well-chosen questions can make inexpensive code models behave less like autocomplete and more like experienced software engineers.

1 Key Information to include

- Mentor: Mirac Suzgun
- External Collaborators (if you have any): None
- Sharing project: No
- Late Days (if you are using any):

Team member	Late days remaining
Ria Garg	4
Alexandra Kim	4
Julia Xi	4
Total pooled	12
Team size	3
Late days applied for the report	6

2 Introduction

Anthropic CEO Dario Amodei recently stated that “we might be 6-12 months away from a model that can do everything SWEs do end-to-end.” [1] Large language models have become remarkably strong at code generation from a single prompt, but they remain brittle when task specifications are ambiguous or incomplete. For instance, should a script told to remove duplicate names treat “Bob” and “bob” as the same string? Human programmers resolve these ambiguities by asking clarification questions; LLMs, on the other hand, often make assumptions and produce code that is misaligned.

This matters for both reliability and cost. Prior work shows that code models suffer large accuracy drops when task descriptions are made ambiguous, incomplete, or contradictory [2]. At the same time, recent Bayesian work shows that strategically chosen yes/no questions can sharply reduce uncertainty in collaborative reasoning tasks [3]. We ask whether the same principle can improve code generation: can a model ask a few carefully chosen binary questions to produce more relevant code? We introduce an interactive pipeline in which the model asks executable clarification questions before generating code. These questions take the form of unit tests and are selected using expected information gain (EIG) over candidate program interpretations.

First, we adapt Bayesian question selection to interactive code generation and implement a full ask-or-submit loop. Second, we compare our Bayesian method to baselines, including one-shot code generation, random test selection, and a TiCoder-style heuristic. Third, we analyze not only pass@1 but also the efficiency trade-off: how many questions, tests, and tokens are required for each gain in accuracy. This work demonstrates how small, cheap models can approximate the code generation accuracy of large, expensive LLMs for a fraction of the cost.

The resulting system forms a fully automated pipeline: for each task, the model samples candidate programs representing different plausible interpretations of the prompt. It then proposes executable tests, scores them by their expected reduction in uncertainty among candidate solutions, runs the chosen test against a reference implementation, updates its beliefs, and either asks again or submits the most likely solution. Across HumanEval+ and MBPP+, our EIG-based approach improves pass@1 by 20-30 absolute points over one-shot generation.

If a few structured questions can substantially improve LLM code generation accuracy, we can better understand how to guide LLMs to behave less like autocompleter and more like human engineers.

3 Related Work

Recent work has explored how interactive questioning can improve reasoning in language models. Grand et al. [3] study collaborative reasoning tasks such as Battleship and show that selecting binary questions using expected information gain (EIG) can significantly improve the performance of weak models. Their work demonstrates that carefully chosen yes/no questions can reduce uncertainty over possible solutions. Our work adapts this idea to the domain of code generation, where the hypothesis space consists of candidate programs that represent different interpretations of a prompt.

TiCoder [4] proposes generating discriminative unit tests that distinguish between candidate programs and ranking them using a heuristic that favors tests splitting candidates evenly. While this approach encourages informative questions, the heuristic is not grounded in information theory. ClarifyCoder [5] trains models to generate natural-language clarification questions before producing code. These approaches do not use executable unit tests or explicit information-theoretic scoring.

These lines of work suggest that interactive clarification can improve code generation, but existing methods either rely on heuristic test selection or natural-language questioning. Our work introduces an information-theoretic approach for selecting executable clarification tests for code generation.

4 Approach

4.1 Task setup

Given a natural-language description of a Python function signature and docstring, the model outputs an implementation that passes a hidden test suite. The model is allowed to ask questions before submitting its final program. It may ask up to some K binary yes/no questions, where each question

is an executable unit test proposed by the model and answered by running it against a reference implementation. For example, suppose the prompt is: “Write a function `dedup(names)` that removes duplicate names from a list.” One area of ambiguity is case sensitivity. The model might ask:

```
assert dedup(["Bob", "bob"]) == ["Bob"]
```

If the answer is `True`, the intended behavior is to ignore the capitalization of names; if `False`, the intended behavior is to treat “Bob” and “bob” as distinct. After receiving these K yes/no answers and filtering through candidates, the model outputs its final code, which is evaluated on hidden tests.

This format has two practical advantages over free-form natural-language clarification. First, every question is directly executable against a reference implementation. Second, the effect of an answer is easy to reason about: a test is useful precisely when plausible candidate programs disagree.

Formally, each example provides a natural-language problem statement x describing a Python function. The system may ask up to K binary clarification questions before producing its final program $\hat{y}$. A question is an executable assertion of the form $q(x) \equiv \text{assert } f(\text{input}) == \text{output}$ answered by running the test against a reference implementation. The oracle returns $a \in \{0, 1\}$, corresponding to `False` or `True`. Our objective is to maximize `pass@1` on a hidden evaluation suite while keeping the number of questions and total token cost small.

4.2 Generating candidate solutions

We represent uncertainty over intended behavior with a set of sampled candidate programs $\{c_j\}_{j=1}^N$ and posterior weights $\{w_j\}_{j=1}^N$, initialized uniformly. Intuitively, each candidate corresponds to one plausible interpretation of the prompt. After each oracle answer, we update the posterior.

For a candidate test t , let $f_t(c_j) \in \{0, 1, \perp\}$ denote whether candidate c_j passes, fails, or produces an undefined outcome on that test. The weighted pass probability among defined outcomes is

$$p_t = \frac{\sum_j w_j \mathbf{1}[f_t(c_j) = 1]}{\sum_j w_j \mathbf{1}[f_t(c_j) \neq \perp]}.$$

Under a binary symmetric oracle-noise model with flip probability ϵ , the base information gain is

$$\text{EIG}(t) = H_b(\epsilon + (1 - 2\epsilon)p_t) - H_b(\epsilon),$$

where H_b is binary entropy. This is maximized when posterior mass is split evenly between pass and fail outcomes.

4.3 EIG-based test scoring

Our final implementation uses a richer score than pure EIG. In addition to the base information gain, we multiply by three quality factors:

$$\text{score}(t) = \text{EIG}(t) \cdot w_{\text{def}}(t) \cdot ((1 - \lambda) + \lambda 4p_t(1 - p_t)) \cdot (1 - \rho u_t),$$

where $w_{\text{def}}(t)$ is the posterior mass on candidates with defined outcomes, u_t is the undefined-outcome mass, λ controls how strongly we reward balanced splits, and ρ penalizes tests that tend to trigger runtime errors. This adjustment de-prioritizes tests that are syntactically plausible but uninformative.

4.4 Ask-or-Submit Decision

After identifying the top-scoring candidate test, the model decides whether it should ask the question or submit immediately. Let p_{now} be the current maximum posterior mass and let $p_{\text{next}}(t)$ be the expected maximum posterior mass after asking test t . We ask if

$$\gamma \cdot p_{\text{next}}(t) > p_{\text{now}},$$

where $\gamma \in [0, 1]$ is an ask-discount. Higher γ makes the system ask more often. If the oracle returns answer $\tilde{a} \in \{0, 1\}$, we update the posterior with a soft Bayesian rule:

$$w'_j \propto w_j \cdot \begin{cases} 1 - \epsilon & \text{if } f_t(c_j) = \tilde{a} \\ \epsilon & \text{if } f_t(c_j) \in \{0, 1\} \text{ and } f_t(c_j) \neq \tilde{a} \\ \eta & \text{if } f_t(c_j) = \perp, \end{cases}$$

where η is a fixed undefined-outcome likelihood.

4.5 Baselines

We compare against four baselines, shown in Table 1.

Baseline	Method Description	Purpose
One-shot	Generate a single program directly from the prompt.	Standard code generation baseline.
Random-tests	Use the same interactive pipeline but sample clarification tests uniformly rather than scoring them.	Tests whether interaction alone improves performance.
Self-consistency	Sample candidate programs and select the one that passes the most generated tests w/o consulting oracle.	Tests whether additional sampling alone explains improvements.
Repair	Generate code and iteratively rewrite the solution using failing assertions from tests as feedback.	Tests pre-generation clarification versus post-hoc correction.

Table 1: Baselines used to evaluate the EIG-based clarification pipeline.

5 Experiments

5.1 Data

We use standard code-generation benchmarks with concrete tasks, reference solutions, and unit tests.

HumanEval: HumanEval consists of 164 hand-written Python programming problems, each with a function signature, natural-language docstring, a canonical solution, and unit tests [6, 7]. We evaluate on EvalPlus HumanEval+, which extends the original HumanEval test suites by 80 times [8, 9].

MBPP: Mostly Basic Programming Problems contains 974 Python tasks solvable by entry-level programmers [10]. We evaluate on MBPP+ from EvalPlus, which extends MBPP by 35 times [8, 9].

We prompt the low-cost **Qwen 2.5 7B Instruct** and **Llama 3.1 8B Instruct** on (1) the original HumanEval and MBPP tasks and (2) HumanEval and MBPP tasks that have been obfuscated by GPT-4 to be incomplete, ambiguous, or contradictory. The mutated prompts are sourced from Larbi et. al [2], defined in Figure 1. Ambiguous prompts have multiple plausible interpretations, incomplete prompts omit crucial details, and contradictory prompts contain conflicting instructions.

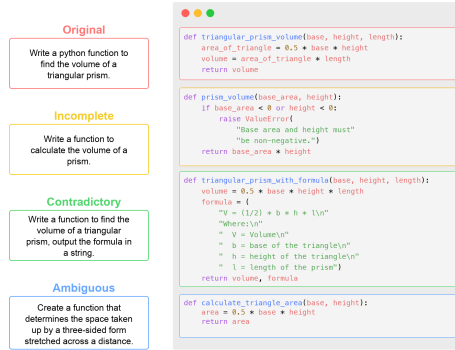


Figure 1: Examples of mutated task descriptions from [2]

5.2 Experimental details

We ask the model to generate 16 candidate programs and up to 16 candidate tests per round, and to ask at most 4 clarification questions. We set $\epsilon = 0.02$ and $\gamma = 0.85$, require minimum candidate coverage for generated tests, and filter malformed, non-discriminative, and incompatible assertions before asking the oracle. We fine-tuned these parameters and conditions by testing small subsets of tasks and choosing the runs with maximum pass@1 within a 5 minute task average time constraint.

5.3 Evaluation method

Our primary metric is pass@1 on the hidden HumanEval+ and MBPP+ tests. To assess if clarification questions help cheap LLMs, we also plot performance as a function of maximum question budget.

We report average number of clarification questions, average total tokens, bootstrap 95% confidence intervals, pass@1 per 1k tokens, and gain in pass@1 per additional 1k tokens versus one-shot.

5.4 Results

Table 2 and Table 3 show that across both small models and both benchmarks, EIG-based test selection is the strongest overall strategy by pass@1. On HumanEval+, Qwen improves from 0.442 with one-shot generation to 0.649 with EIG-based test selection, while Llama improves from 0.419 to 0.627. On MBPP+, Qwen improves from 0.547 to 0.723 and Llama improves from 0.501 to 0.698.

Comparing against random-tests, we observe that EIG-based test selection also beats random questioning on accuracy and token usage. These gains cannot be explained by interaction alone; they depend on deliberately choosing informative tests. On Qwen HumanEval+, EIG-based test selection improves pass@1 by 10.3% over random questioning while using about 2.1k fewer tokens on average. The same dominance appears on Qwen MBPP+ (+5.8 points, 1.8k fewer tokens), Llama HumanEval+ (+10.5 points, 2.4k fewer tokens), and Llama MBPP+ (+6.6 points, 3.1k fewer tokens).

Model	Strategy	pass@1 (95% CI)	Avg. questions	Avg. tokens	pass / 1k tokens
Qwen	EIG-based test selection	0.649 [0.613, 0.686]	0.839	4183.1	0.155
Qwen	one-shot	0.442 [0.404, 0.480]	0.000	319.1	1.385
Qwen	random-tests	0.546 [0.508, 0.584]	0.942	6325.9	0.086
Qwen	repair	0.438 [0.399, 0.476]	0.000	1642.8	0.266
Qwen	self-consistency	0.498 [0.460, 0.537]	0.000	3694.0	0.135
Llama	EIG-based test selection	0.627 [0.588, 0.663]	0.839	4383.5	0.143
Llama	one-shot	0.419 [0.381, 0.457]	0.000	334.1	1.255
Llama	random-tests	0.521 [0.482, 0.559]	0.942	6819.2	0.076
Llama	repair	0.427 [0.389, 0.465]	0.000	1812.3	0.236
Llama	self-consistency	0.494 [0.456, 0.532]	0.000	3743.4	0.132

Table 2: HumanEval+ results.

Model	Strategy	pass@1 (95% CI)	Avg. questions	Avg. tokens	pass / 1k tokens
Qwen	EIG-based test selection	0.723 [0.709, 0.737]	0.967	4361.0	0.166
Qwen	one-shot	0.547 [0.531, 0.563]	0.000	209.8	2.607
Qwen	random-tests	0.665 [0.650, 0.680]	1.133	6201.0	0.107
Qwen	repair	0.563 [0.547, 0.578]	0.000	1040.4	0.541
Qwen	self-consistency	0.585 [0.570, 0.600]	0.000	7117.1	0.082
Llama	EIG-based test selection	0.698 [0.684, 0.712]	0.967	5213.0	0.134
Llama	one-shot	0.501 [0.485, 0.517]	0.000	301.0	1.665
Llama	random-tests	0.632 [0.617, 0.647]	1.133	8271.0	0.076
Llama	repair	0.529 [0.514, 0.545]	0.000	1040.4	0.508
Llama	self-consistency	0.566 [0.551, 0.581]	0.000	7101.1	0.080

Table 3: MBPP+ results.

5.4.1 Types of prompt mutations

The efficacy of different strategies varies depending on how the prompt was mutated. Asking questions helps most on ambiguous and incomplete prompts, exactly the intended setting for interactive clarification. On HumanEval+, EIG-based test selection raises Qwen from 0.556 to 0.875 on ambiguous prompts and from 0.555 to 0.696 on incomplete prompts. On MBPP+, the same pattern holds, likely because the task definition is underspecified but still recoverable. Contradictory prompts remain difficult to nail. As shown in Table 4, on Qwen HumanEval+, EIG-based test selection reaches only 0.136 on contradictory prompts, compared with 0.073 for one-shot and 0.119 for random-tests.

Benchmark	Condition	EIG-based test selection	one-shot	random-tests
HumanEval+	original	0.889	0.836	0.850
HumanEval+	ambiguous	0.875	0.556	0.612
HumanEval+	incomplete	0.696	0.555	0.605
HumanEval+	contradictory	0.136	0.073	0.119

Table 4: Qwen performance on HumanEval+ broken down by prompt type.

5.4.2 Cost comparisons

Figure 2 compares cost and accuracy. One-shot uses only 209 – 334 tokens, EIG uses roughly 4.2k–5.2k, and random-tests uses 6.2k–8.3k. In terms of pass@1, that extra spend is often worthwhile, although for per-token efficiency, one-shot is the best strategy on pass per 1k tokens.

Among interactive methods, however, EIG-based test selection is consistently the best use of additional compute. Relative to one-shot, it yields +0.176 to +0.208 pass@1 for roughly 3.9k–4.9k extra tokens, or +0.040 to +0.054 pass@1 per additional 1k tokens. Random-tests achieves only +0.016 to +0.020 per extra 1k tokens.

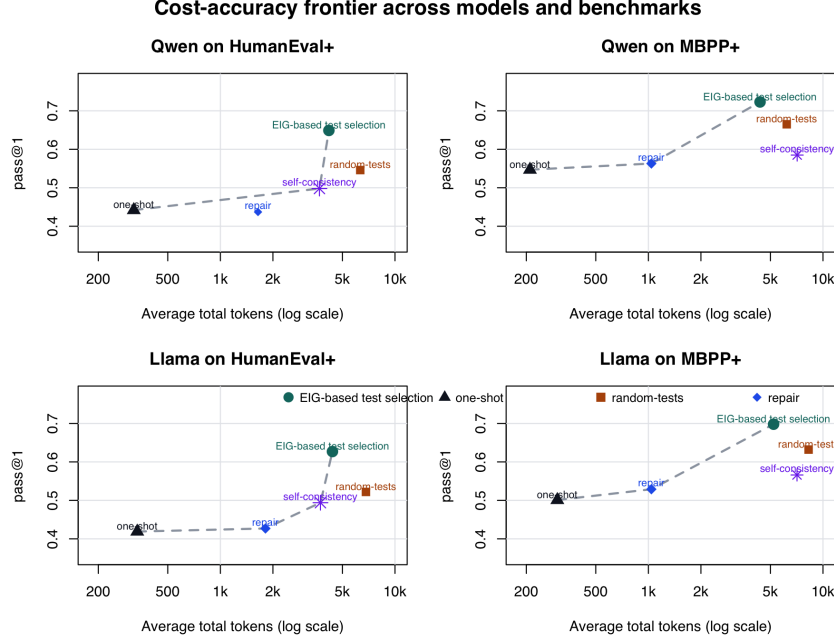


Figure 2: Cost-accuracy comparison across Qwen and Llama on HumanEval+ and MBPP+.

5.4.3 Comparisons to frontier LLMs

We also ran one-shot HumanEval+ comparisons with Gemini 2.5 and Gemini 3 Flash to discern how our EIG strategy on inexpensive models stacked up with large LLMs. Results are shown in Table 5.

Model / strategy	pass@1 (95% CI)	Avg. tokens	pass / 1k tokens
Gemini 2.5 Flash one-shot	0.449 [0.372, 0.530]	628	0.719
Gemini 3 Flash (Thinking) one-shot	0.683 [0.610, 0.756]	21089	0.032
Qwen with EIG-based test selection	0.649 [0.613, 0.686]	4183	0.155

Table 5: HumanEval+ comparison with Gemini one-shot runs.

Surprisingly, a tiny 7B model with interactive EIG generation can approximate the performance of a much larger and more expensive LLM. On the HumanEval+ comparison, the drop from 21089 (Gemini 3 Flash) to 4183 (Qwen) average tokens implies $\sim 80\%$ lower spending on token billing.

5.5 Ablation Studies

The three ablation studies are depicted in Table 6 and the full results are in Tables 7, 8, and 9.

Test selection: EIG scores a question by the expected information gain and TiCoder scores based on how well it can split the set. We compare these two technique on the basis of performance gains.

Belief update: We can either use the soft update to update beliefs or use hard pruning to remove incorrect candidates. We check whether keeping uncertainty or complete elimination helps more.

Stopping: Asking a clarification question can incur higher latency and token cost while potentially improving code accuracy, whereas outputting code early can save cost but risk accuracy. We compare stopping early (enable VOI-stop) and continuing to ask questions (disable VOI-stop).

Ablation Study	Metric	Pass@1	Avg. #Q	Avg. tokens
Test selection	EIG	0.757	2.157	10274.7
	TiCoder	0.667	2.167	9510.7
Belief update	Soft update	0.800	2.000	10254.1
	Hard prune	0.776	1.672	10228.0
Stopping rule	VOI-stop on	0.700	1.167	8704.5
	VOI-stop off	0.767	1.833	9965.3

Table 6: Summary of ablation results. Full tables are included in the appendix.

6 Analysis

An EIG loop can only choose among the hypotheses and tests it is given, so the limiting reagents are test quality and candidate solution quality. When the candidate pool contains several plausible interpretations and the test pool contains executable unit tests, EIG works phenomenally.

6.1 Representative Success and Failure Cases

Success (HumanEval Task 14): The ambiguous prompt was to write code for “*some calculations related to a three-dimensional geometric shape with triangular surfaces.*”

One-shot solution (incorrect tetrahedron-like formula) <pre>def find_Volume(int_arg, int_arg2, int_arg3): return (int_arg * int_arg2 * int_arg3) / 6</pre>	Solution after EIG clarification <pre>def find_Volume(int_arg, int_arg2, int_arg3): base_area = (int_arg * int_arg2) / 2 height = int_arg3 volume = base_area * height return volume</pre>
--	--

EIG-based test selection asks:

```
assert find_Volume(1, 2, 3) == 0      observed=False
assert find_Volume(10, 8, 6) == 240  observed=True
```

The second test anchors the triangular prism formula, while the first rules out trivial cases. Unlike random tests, which asked low-quality questions, the EIG method identified the correct interpretation.

Failure (HumanEval Task 134): The incomplete prompt asked whether *the last character of a string is a letter and not part of a word*. EIG-based test selection asked:

```
assert check_if_last_char_is_a_letter('12345 a') == False
assert check_if_last_char_is_a_letter('!a') == True
```

and produced the following code:

```
def check_if_last_char_is_a_letter(txt):
    if len(txt) == 0: return False
    last_char = txt[-1]
    second_last_char = txt[-2] if len(txt) > 1 else None
    return last_char.isalpha() and (second_last_char == ' ' or second_last_char
    == '\t' or second_last_char == '\n' or second_last_char is None)
```

This code fails due to the edge case where a trailing space is not considered a character. The model’s questions weren’t diverse enough to expose this nuance; perhaps more questions could help.

6.2 Good and Bad Tests

Test quality has a clear bottleneck. A good EIG-selected pair from HumanEval Task 140 is:

```
assert fix_spaces('abc  ') == 'abc-'    observed=True
assert fix_spaces('abc ') == 'abc-'    observed=False
```

This pair isolates the exact threshold between two spaces and three spaces, separating candidate solution families. By contrast, malformed or non-discriminative tests waste tokens:

```
assert fix_spaces('___End') == '-End
assert fix_spaces('Example') == 'Example'
```

The first is syntactically malformed; the second is valid Python but non-discriminative for the candidate set that produced it.

6.3 Ablation Studies

As seen in Tables 6 and 7, EIG outperforms TiCoder overall (0.757 vs. 0.667 pass@1), with the largest gains on ambiguous prompts (0.833 vs. 0.600 pass@1), which supports our main claim that information-theoretic test selection is especially useful when prompts are underspecified. TiCoder is slightly better on original prompts (0.900 vs. 0.864) and uses fewer tokens, suggesting that simpler heuristics can be useful when the task is already well specified.

Soft posterior updates, reported in Tables 6 and 8, perform slightly better overall than hard pruning (0.800 vs. 0.776), although hard pruning is better on incomplete prompts (0.737 vs. 0.600). This suggests that preserving uncertainty is usually beneficial, but aggressive elimination can help when the hypothesis space contains many weak candidates.

Finally, Tables 6 and 9 show that disabling the VOI stopping rule improves overall accuracy (0.767 vs. 0.700) but increases both the number of questions and total token cost. Thus, the stopping rule introduces a clear cost-accuracy trade-off. Overall, the ablations suggest that the strongest configuration depends on whether the priority is robustness under ambiguity or lower-cost inference.

7 Conclusion

Self-executable clarification provides a powerful mechanism for improving code generation when a task is underspecified. In such settings, a few well-chosen binary tests can isolate plausible interpretations and make the difference between correct and incorrect code. Across two inexpensive LLMs, Qwen 2.5 7B and Llama 3.1 8B, EIG-based test selection consistently outperforms one-shot generation, repair, self-consistency, and random questioning on HumanEval+ and MBPP+. Our results highlight that (1) interaction itself improves code generation, since random questioning already outperforms one-shot generation, and (2) selecting tests with an EIG-based criterion yields additional gains, showing the importance of asking high-quality questions.

The goal is not necessarily for a cheap, interactive 7B model to beat every frontier model on raw pass@1; it is to close the accuracy gap sufficiently so that the cost difference becomes favorable. We show that a 7B model like Qwen with EIG-based clarification comes close to a much larger model (Gemini 3 Flash Thinking) while using roughly $\frac{1}{5}$ as many tokens. For real deployment, inference cost often matters as much as accuracy.

One promising future direction is to investigate how models, when given underspecified requirements, move beyond clarifying edge cases to reason at a high level about software design. To handle contradictory prompts, for example, the model might be allowed to ask higher-level natural-language questions about which instruction should take priority. More broadly, interactive clarification represents a general strategy for improving LLM reliability. By asking the right questions, small code models can learn to behave less like autocomplete and more like pair programmers.

Team contributions

R.G., A.K., and J.X. jointly designed the project methodology, drafted/revised the report, and discussed results. R.G. implemented the EIG pipeline and ablation studies. A.K. implemented the baseline training loop and ran the LLM evaluations. J.X. implemented the Bayesian framework.

References

- [1] DRM News. Full discussion: Google’s demis hassabis, anthropic’s dario amodei debate the world after agi | ai1g, 2026. YouTube video.
- [2] Maya Larbi, Amal Akli, Mike Papadakis, Rihab Bouyousfi, Maxime Cordy, Federica Sarro, and Yves Le Traon. When prompts go wrong: Evaluating code model robustness to ambiguous, contradictory, and incomplete task descriptions, 2025.
- [3] Gabriel Grand, Valerio Pepe, Jacob Andreas, and Joshua B. Tenenbaum. Shoot first, ask questions later? building rational agents that explore and act like people, 2025.
- [4] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering*, 50(9):2254–2268, September 2024.
- [5] Jie JW Wu, Manav Chaudhary, Davit Abrahamyan, Arhaan Khaku, Anjiang Wei, and Fatemeh H. Fard. Can code language models learn clarification-seeking behaviors?, 2025.
- [6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.
- [7] Hugging Face Datasets. openai/openai_humaneval dataset card. https://huggingface.co/datasets/openai/openai_humaneval, 2021.
- [8] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, 2023.
- [9] EvalPlus Team. evalplus/evalplus: Rigorous evaluation framework for llm4code (humaneval+ and mbpp+). <https://github.com/evalplus/evalplus>, 2023.
- [10] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

A Appendix

A.1 Sample of a real task and model output

Write a function to get items from a dataset.

One-shot solution which failed the tests:

```
def small_nnum(list_arg, int_arg):  
    return [x for x in list_arg  
            if x < int_arg]
```

Working solution after asking high EIG questions:

```
def small_nnum(list_arg, int_arg):  
    return sorted(list_arg)[:int_arg]
```

Below are the high-EIG clarifying questions the model asked:

```
assert small_nnum([10,20,30,40,50],10)==[] -> False  
assert small_nnum([1,2,3,4,5],2)==[1,2] -> True  
assert small_nnum([10,...,100],5)==[10,20,30,40,50] -> True
```

A.2 Ablation Studies Tables

Condition	Scorer	Pass@1	Average #Q	Average tokens
original	EIG	0.864	1.636	11036.8
original	TiCoder	0.900	1.700	10303.1
incomplete	EIG	0.583	2.083	9639.1
incomplete	TiCoder	0.500	2.000	8913.5
ambiguous	EIG	0.833	2.708	10211.6
ambiguous	TiCoder	0.600	2.800	9315.4
All	EIG	0.757	2.157	10274.7
All	TiCoder	0.667	2.167	9510.7

Table 7: Ablation on the test-selection rule.

Condition	Update	Pass@1	Average #Q	Average tokens
original	Soft update	0.900	1.600	11001.7
original	Hard prune	0.842	1.526	11524.5
incomplete	Soft update	0.600	2.000	9493.8
incomplete	Hard prune	0.737	1.474	9278.7
ambiguous	Soft update	0.900	2.400	10266.7
ambiguous	Hard prune	0.750	2.000	9898.2
All	Soft update	0.800	2.000	10254.1
All	Hard prune	0.776	1.672	10228.0

Table 8: Ablation on the belief update rule.

Condition	VOI stop	Pass@1	Average #Q	Average tokens
original	On	0.700	1.300	10163.1
original	Off	0.800	1.500	10694.3
incomplete	On	0.600	0.700	8099.8
incomplete	Off	0.700	1.800	9832.2
ambiguous	On	0.800	1.500	7850.7
ambiguous	Off	0.800	2.200	9369.3
All	On	0.700	1.167	8704.5
All	Off	0.767	1.833	9965.3

Table 9: Ablation on the stopping policy.

A.3 Another Example of Success

Success (HumanEval Task 138): The ambiguous prompt was ‘write a function that given a number, judge whether it could be assembled by summing exactly four even parts.’ One-shot failed with:

```
def is_equal_to_sum_even(int_arg):
    if int_arg < 8: return False
    while int_arg % 2 == 0:
        int_arg -= 2
    return int_arg == 0
```

This loops forever on even inputs such as powers of two. EIG-based test selection asked three questions:

```
assert is_equal_to_sum_even(0) == False    observed=True
assert is_equal_to_sum_even(6) == True     observed=False
assert is_equal_to_sum_even(14) == False   observed=False
```

These tests rule out several distinct families of bad solutions; for instance, the first test rules out candidates that only check parity. The revised solution passed all tests:

```

def is_equal_to_sum_even(int_arg):
    for i in range(1, int_arg):
        for j in range(i, int_arg):
            for k in range(j, int_arg):
                for l in range(k, int_arg):
                    if i + j + k + l == int_arg
                       and i % 2 == 0 and j % 2 == 0
                       and k % 2 == 0 and l % 2 == 0:
                        return True
    return False

```

A.4 Another Example of Failure

Since contradictory prompts were the category which the models consistently struggled with, we analyze why every strategy failed on HumanEval Task 126. The contradictory prompt says *duplicates are not allowed, duplicates should be ignored, the return type must be an integer, and the time complexity must be $O(1)$* . EIG-based test selection asked:

```

assert is_sorted([1, 1, 2, 3]) == False
assert is_sorted([1, 2, 3, 4, 5, 6, 6]) == True

```

and returned:

```

def is_sorted(lst):
    s = ''.join(str(x) for x in sorted(lst))
    t = ''.join(str(x) for x in lst)
    return int(t == s and len(set(lst)) >= len(lst) / 2)

```

This code uses `len(set(lst)) >= len(lst)/2` as a proxy for the duplicate constraint, so it fails edge cases such as `[1,2,2,2,3,4]`. Here the issue is not only weak tests. The prompt is internally inconsistent, so many candidates converge on the same wrong abstraction. When the candidate pool is not diverse enough, posterior updates cannot recover the missing semantics.