Natural Language Defects in Code Generation Prompts: Empirical Impact and Lightweight Detection

Anonymous ACL submission

Abstract

Large language models are widely used for code generation, yet they rely on an implicit assumption that the task descriptions are sufficiently detailed and well-formed. However, in practice, users may provide defective descriptions, which can have a strong effect on code correctness. To address this issue, we conduct a large-scale empirical analysis evaluating the robustness of 10 code LLMs across 3 benchmarks under three natural language defect types: Lexical Vagueness (LV), Under-Specification (US), and Syntax/Formatting (SF). Our results show that defect impact is not uniform: Under-Specification causes Pass@1 drops of up to 15.3% with no model immune, including state-of-the-art reasoning models; Lexical Vagueness produces moderate, benchmark-dependent degradation; and surface formatting noise has negligible effect. Crucially, robustness is shaped more by the structural properties of the task description than by model size or architecture. Second, to address these defects before they reach the model, we develop *SpecValidator*, a lightweight LoRA-fine-tuned classifier based on a 1.5B code model, achieving F1 = 0.804 and MCC = 0.745, significantly outperforming few-shot GPT-5-mini (F1 = 0.462) and Claude-Sonnet-4 (F1 = 0.557), and demonstrating that strong code generation ability does not transfer to prompt meta-reasoning. Perhaps more importantly, our analysis indicates that SpecValidator can generalize to unseen issues and detect unknown Under-Specification defects in the original (real) descriptions of the benchmarks used.

1 Introduction

Large language models (LLMs) such as Codex (Chen et al., 2021b), AlphaCode (Li et al., 2022), Code Llama (Rozière et al., 2023), and Qwen2.5 Coder (Hui et al., 2024b), along with related LLM agents (OpenAI, 2025; Anthropic, 2024), increasingly support developers across a broad spectrum of programming tasks, such as code generation (Jiang et al., 2026). Previous work demonstrates their practical impact: AI-based coding tools boost developer productivity (Peng et al., 2023). In parallel, evaluation methodologies have evolved substantially. Benchmarks such as HumanEval (Chen et al., 2021b), MBPP (Austin et al., 2021), and LiveCodeBench (Jain et al., 2024) have progressively refined how code correctness is measured, through denser test suites, contamination-free problem sets, and richer formal specifications.

Yet all of the above evaluate the generation of code on well-formed and sufficiently described task descriptions. However, in practice, natural language descriptions are inherently imprecise and may be subject to several description issues, similar to specification defects that appear in software specifications (Meyer, 1985; van Lamsweerde, 2009; Larbi et al., 2025; Vogelsang et al., 2025). Therefore, task descriptions can include vague terminology, omit edge-case constraints, or may contain surface-level noise, such as typographical and formatting errors, all of which can drastically affect the correctness of the generated code (Larbi et al., 2025; Jia et al., 2025; Zhu et al., 2024; Wu and Fard, 2025).

To address this issue, we developed *SpecValidator*, a simple classifier to automatically detect such task description defects prior to code generation. SpecValidator is a lightweight LoRA-fine-tuned predictor trained on a set of artificially injected defects. In particular, we define and apply three defect types — *Lexical Vagueness* (LV), *Under-Specification* (US), and *Syntax and Formatting* (SF) — reflecting the key issues task descriptions may have, based on which we fine-tune our predictor and check its ability to detect them.

We evaluate 10 LLMs spanning from small (6–7B) to large open-source (15–34B) and state-of-the-art reasoning models across 3 benchmarks and

found significant degradation in the models' code correctness. Interestingly, our results show that the structural characteristics of the task descriptions play an important role on the correctness of the generated code. Under-Specification causes the most significant drops, of up to 15.3%, with no model being robust, while benchmarks with richer contextual grounding, such as LiveCodeBench, are substantially more resilient. Lexical Vagueness produces moderate and benchmark-dependent degradation, whereas surface formatting issues have a negligible effect.

SpecValidator shows strong defect detection performance with F1 = 0.804 and MCC = 0.745, outperforming significantly few-shot GPT-5-mini (F1 = 0.462 and MCC = 0.276) and Claude Sonnet 4 (F1 = 0.557 and MCC = 0.412). Perhaps more importantly, by applying SpecValidator on the original set of task descriptions, we reveal that these surprisingly contain Under-Specification defects (confirmed via thorough manual inspection). This demonstrates the ability of SpecValidator to detect real specification defects. This finding could raise a broader concern: if standard benchmarks contain latent specification defects, reported performance differences across models may partly reflect description quality rather than model capability.

The main contributions of this paper are:

- A large-scale analysis of 10 models and 3 benchmarks, showing that defects' impact is shaped both by defect type and task description structure.
- *SpecValidator* A lightweight defect detector, a Qwen2.5-Coder-1.5B LoRA fine-tune model, achieving F1 = 0.804 and MCC = 0.745, outperforming both GPT-5 mini and Claude Sonnet 4. *SpecValidator* is capable of detecting real description defects in the benchmarks we use.
- A publicly available dataset of 6,573 defective task description instances, supporting replication and future research on task description and benchmark design.

2 Related Work

2.1 LLM-based code generation and evaluation

The HumanEval (Chen et al., 2021a), *MBPP* (Austin et al., 2021) and *EvalPlus* (Liu et al., 2023) benchmarks evaluate functional correctness using test pass rates (e.g. *pass@k*). *LiveCodeBench* (Jain et al., 2024) mitigates benchmark contamination by continuously introducing new competitive-programming problems, while *BigCodeBench* (Zhuo et al., 2025), and Siddiq et al. (Siddiq et al., 2024) extend evaluation to more complex, library-intensive tasks that require diverse API usage and more realistic programming workflows. Open-weight models include *Code Llama* (Rozière et al., 2023), *StarCoder2* (Lozhkov et al., 2024), *DeepSeek-Coder* (Guo et al., 2024), and *Qwen2.5-Coder* (Hui et al., 2024b), while proprietary systems such as GPT-5 (OpenAI, 2025) and Claude Code (Anthropic, 2024) represent the state of the art in commercial deployments. *Despite the growing diversity of benchmarks and models, evaluation practices remain largely focused on correctness under well-specified task descriptions, leaving model behavior under realistic description defects unexplored.*

2.2 Prompt engineering for code tasks

Li et al. (Li et al., 2025) proposed Structured Chain-of-Thought (SCoT) prompting, which decomposes generation through program-structure-aware intermediate steps (Wei et al., 2022), yielding consistent improvements over standard prompting on HumanEval and MBPP. Test-driven feedback loops (Fakhoury et al., 2024) ask users to refine the task description using feedback from tests generated from the original description, using a multi-turn LLM dialog. Self-Debugging (Chen et al., 2024) teaches models to self-identify and correct errors in their own outputs via execution-guided feedback. These studies focus on the generation of correct code from well-specified, sufficiently detailed descriptions and excluding potential defects.

2.3 LLM robustness on prompt perturbations

ReCode (Wang et al., 2023) studied over 30 semantic-preserving transformations (e.g., variable renaming, dead-code insertion, and docstring reformatting), revealing substantial but uneven performance degradation. Mastropaolo et al. (Mastropaolo et al., 2023) showed that semantically equivalent paraphrases of Javadoc can alter the GitHub Copilot output in nearly half of the cases, affecting correctness 28% of the cases. *NLPerturbator* (Chen et al., 2026) proposed 18 categories of natural-language variations from real developer

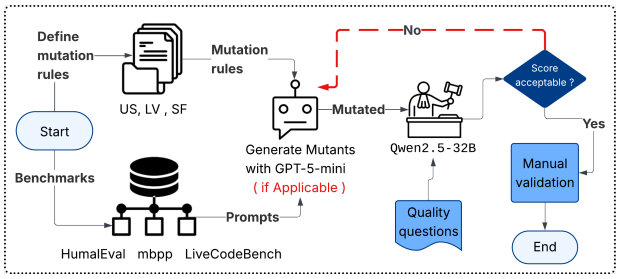

Figure 1: Dataset construction. Defects are generated with GPT-5-mini with defect quality being scored by a Qwen-32B evaluator. Defective descriptions are then manually verified by 3 human reviewers.

data, while Rabbi et al. (Rabbi et al., 2025) extended the analysis to multiple programming languages. Shirafuji et al. (Shirafuji et al., 2023) demonstrated that code LLMs are highly sensitive to superficial modifications of problem descriptions. Sclar et al. (Sclar et al., 2024) highlighted the sensitivity of LLMs to prompt formatting, showing that minor changes such as whitespace or casing impacts code correctness, and PromptRobust(Zhu et al., 2024) showed contemporary LLMs suffer up to 39% average performance drops from word-level perturbations (Zhu et al., 2024; Xia et al., 2024). Larbi et al. (Larbi et al., 2025), found that code correctness is strongly affected by ambiguous, incomplete, and contradictory descriptions.

Our work differs by focusing on semantically meaningful defect categories grounded in the linguistic properties of natural language prompts and by providing a detection tool rather than just an analysis of impact.

3 Dataset Construction

Figure 1 shows the dataset construction process. Starting from three code-generation benchmarks, we apply three complementary defect-injection strategies to produce our benchmark. Data quality is assessed in two levels: automated evaluation by an independent LLM judge across all 6,573 mutated instances, and manual inspection of a stratified random sample for ground-truth calibration.

3.1 Benchmarks

We select three code-generation benchmarks that cover a range of task types, from structured function completion to natural-language descriptions to full programming problems. This variety helps us evaluate model performance across different levels of difficulty, description styles, and reasoning requirements. Table 1 records the key characteristics of the benchmarks we use. HumanEval (Chen et al., 2021a) uses structured docstrings with typed function signatures; MBPP (Austin et al., 2021) provides short, informal descriptions in natural-language with minimal scaffolding; and LiveCodeBench (Jain et al., 2024) provides competitive-programming problem statements with explicit numerical bounds and formal input/output contracts. All benchmarks are for Python.

3.2 Generating defective task descriptions

We construct defective task descriptions by systematically mutating original descriptions. Some examples of our mutations are provided in Appendix Table 7. We generate three defects per task description (one per defect type). In cases where the original descriptions are too short or have no constraint to delete, no defective variant is produced. Mutations are applied only to task descriptions: reference solutions, test cases, and evaluation harnesses are not modified, so any observed change (e.g. in Pass@1 metric) is solely attributable to the altered description.

We generate defective descriptions with `gpt-5-mini` (via the OpenAI Batch). We prompt the model to mutate original descriptions using a predefined set of transformation rules. To ensure that the generated descriptions comply with our requirements, we use an LLM-as-a-judge to assign compliance scores. In case the scores are not above 85% for all defect types and benchmarks, we manually inspect the low-compliance defective descriptions and refine our prompt to gpt-5-mini until we reach at least this 85% compliance threshold. All our generation prompts are available in the replication package.[1].

3.2.1 Defect Types

Previous work has identified token perturbations and lexical ambiguity as a source of performance variance in code generation models (Wang et al., 2023; Chen et al., 2026; Larbi et al., 2025). Missing or under-specified requirements can also cause models to generate code that is syntactically correct but semantically incorrect (Mu et al., 2024; Jia et al., 2025; Fakhoury et al., 2024; Hamidi et al., 2025). Surface-level formatting noise, while seemingly superficial, has been shown to induce performance swings in open-source LLMs (Sclar et al.,

[1] https://anonymous.4open.science/r/prompt_defects-detection

Table 1: Overview of the three benchmarks used in our experiments

Benchmark	Size	Task style	Complexity	Language	Evaluation
HumanEval	164	Function completion (docstring)	Easy–Medium	Python	Unit tests
MBPP	974	NL description of short programs	Easy	Python	Unit tests
LiveCodeBench	1,055	Competitive programming(full problem statement)	Easy–Hard	Python	stdin/stdout

2024; Wang et al., 2023). Together, these three defect classes (lexical vagueness, under-specification, and syntactic/formatting corruption) represent orthogonal failure modes that cover the main ways a natural-language specification defect can appear in the user prompts. Representative examples are provided in Appendix Table 7

3.2.2 Mutation Operators

Lexical Vagueness (LV) LV mutations replace precise, task-relevant vocabulary with semantically broader or less informative alternatives, reducing the specificity of the task without removing any explicit constraint. This mirrors the paraphrasing and synonym-substitution perturbations studied by Wang et al. (Wang et al., 2023) and Chen et al. (Chen et al., 2026). Concretely, an LV mutation consists of the following transformations:

- Replace domain-specific terms with broader synonyms (e.g., *sort* → *arrange*).
- Weaken action verbs to less precise ones (e.g., *insert* → *put*).
- Rename function and parameter identifiers with less informative synonyms (e.g., `delimiter` → `filler`).
- Generalize or remove type annotations.
- Make multiple lexical changes, compounding the ambiguity.

Under-Specification (US) US mutations remove one explicit constraint from the task description, producing a description that remains grammatically well-formed but is semantically under-specified. This defect type is motivated by empirical findings that incomplete requirements cause most of the intent mismatch in LLM-based code generation (Larbi et al., 2025). To ensure under-specified defects, we restrict deletion to the following elements of the description:

- Numeric bounds or threshold values.
- Ordering or sorting rules.
- Input preconditions or domain restrictions.
- Output edge-case behavior.
- Formula or behavioral pinning constraints.
- Error or exception handling requirements.
- Output type or format constraints.

Syntax and Formatting (SF) SF mutations introduce surface-level noise into the task description without altering its semantic content, targeting the susceptibility of LLMs to low-level textual corruption (Sclar et al., 2024). Sclar et al. (Sclar et al., 2024) demonstrated that format variations alone can have a significant impact, motivating this defect type. Each SF mutation applies one of the following corruption types:

- Token-level corruption (character transposition, typos).
- Delimiter, bracket, or colon corruption.
- Indentation or whitespace layout corruption.
- Example-block formatting corruption.

3.3 Defect validation

We assess defect quality at two levels: automated scoring by an independent LLM judge and manual inspection of a stratified sample.

Automated assessment. We use Qwen2.5-Coder-32B-Instruct (Hui et al., 2024a) as judge, a strong open-source code model distinct from the generation model (`gpt 5-mini`), deployed locally for reproducibility. Each mutant is scored for compliance (mutation correctly applied) and naturalness (reads like a real user prompt). The LV and SF mutations achieve near-perfect compliance (0.98–1.00) and naturalness (0.96–1.00). US shows lower compliance (0.85–0.89), mainly due to structural constraints in shorter MBPP prompts and stricter LiveCodeBench specifications.

Manual validation. Three researchers independently annotated a stratified random sample of 100 mutants (∼11 per benchmark–defect pair), using the same compliance and naturalness criteria, with disagreements resolved by majority vote. Human annotators rated 97% of instances as compliant and 86% as natural, closely matching the automated judge.

4 Study Design

4.1 Research questions

Developers typically write imperfect descriptions: they may use imprecise vocabulary, omit edge cases, or write noisy statements. The robustness of code LLMs to such issues remains underexplored and there is also no systematic infrastructure to detect or mitigate these defects. Our work explores the following questions:

RQ1 (Impact of defects): How lexical vagueness, under-specification, and syntactic noise affect generated code correctness across benchmarks of varying complexity? We evaluate 10 LLMs (3 small, 5 large, 2 reasoning) on HumanEval, MBPP, and LiveCodeBench. For each benchmark, we introduce 3 types of defect (US, LV, SF). We measure the performance of each model in generating correct code using Pass@1 metric.

RQ2 (Defect detection): Can a small fine-tuned model detect description defects as well as large model baselines? We fine-tune Qwen2.5-Coder-1.5B under three parameter budgets (linear probe, LoRA, full fine-tuning) and compare against zero-shot and few-shot GPT-5-mini and Claude Sonnet 4.

RQ3 (Defect detection on original prompts): Does SpecValidator detect (unseen) real-world defects? We apply it to the original benchmark descriptions and verify whether flagged instances reflect real specification gaps.

4.2 Defect detection and SpecValidator

We classify task descriptions in one of four categories: CLEAN, LV, US, or SF. We evaluate two families of approaches.

Proprietary LLM systems. We query GPT-5-mini and Claude Sonnet 4 with a structured prompt that presents the code generation task description and instructs the model to identify the defect type, if any. In addition to the zero-shot setting, we also evaluate a few-shot variant (Brown et al., 2020) where the model is provided with four labeled examples prior to classification. These serve as strong baselines that do not require training data and leverage the broad knowledge of frontier models.

SpecValidator (fine-tuned classifier) We fine-tune Qwen2.5-Coder-1.5B (Hui et al., 2024b) as a sequence classifier on a labeled dataset constructed from our mutation approach. Each example consists of a task description (original or mutated) paired with its defect label. We train in three parameter budgets to assess the cost–accuracy trade-off: (i) *linear probing*, where only the classification head is trained and the backbone is frozen; (ii) *LoRA adaptation*, where low-rank updates are applied to attention projection layers as described in the following implementation section; and (iii) *full fine-tuning*, where all parameters are updated.

The training set is composed of equal numbers of original and mutated description (8,766 samples in total). The dataset is split 80/10/10 into train, validation, and test sets, stratified by label. The finetuning is performed twice with two different seeds. [2]

5 Experimental Results

5.1 Impact of defects on code generation

Table 2 summarizes the evaluation of ten code LLMs on three benchmarks (HumanEval, MBPP, and LiveCodeBench) under three types of mutation: Under-Specification (US), Lexical Vagueness (LV), and Syntactic/Formatting errors (SF). The results show that these mutation types induce substantially different levels of performance degradation in models and benchmarks, suggesting that description defects vary in their impact and should not be treated uniformly.

Under-specification is consistently the most damaging mutation. As shown in Table 2, US mutations cause Pass@1 drops of 7.9–15.3% on HumanEval across all model scales. Even state-of-the-art reasoning models are substantially affected: GPT-5-mini and Claude Sonnet 4, despite baselines of 96.3% and 95.7% respectively, each decline by 9.7%. A consistent pattern holds on MBPP, where US induces drops of 5.1–10.8% across all tiers.

This pattern holds uniformly across all model families, with no model showing immunity at any scale or architecture. The uniformity of this degradation across architectures and scales indicates that the cause is intrinsic to the mutation itself: removing a constraint leaves the specification compatible with multiple plausible implementations, most of which are incorrect.

Lexical vagueness produces benchmark-dependent, moderate drops. On HumanEval, LV mutations produce drops ranging from 1.3% (Qwen2.5-Coder-32B) to 12.2% (DeepSeek-Coder-6.7B), with CodeLlama-7B and StarCoder2-15B

[2] Replication package: https://anonymous.4open.science/r/prompt_defects-detection

Table 2: Pass@1 results on all benchmarks and mutation types. For each mutation (US, LV, SF), Pass@1 deltas wrt to the original are shown in red (↓ drop) and green (↑ gain). **US** = Under-Specification; **LV** = Lexical Vagueness; **SF** = Syntax and Formatting.

	HumanEval				MBPP				LiveCodeBench			
Model	**Orig**	**US**	**LV**	**SF**	**Orig**	**US**	**LV**	**SF**	**Orig**	**US**	**LV**	**SF**
Small models (≤7B)												
CodeLlama-7B	37.2	29.3 ↓7.9	29.3 ↓7.9	37.2 =	33.3	26.8 ↓6.5	31.2 ↓2.1	33.7 ↑0.4	8.1	8.4 ↑0.3	8.2 ↑0.1	7.8 ↓0.3
DeepSeek-Coder-6.7B	72.6	57.3 ↓15.3	60.4 ↓12.2	68.9 ↓3.7	44.1	36.2 ↓7.9	41.6 ↓2.5	43.5 ↓0.6	16.3	16.0 ↓0.3	14.8 ↓1.5	15.2 ↓1.1
Qwen2.5-Coder-7B	82.3	67.7 ↓14.6	75.6 ↓6.7	81.7 ↓0.6	50.1	41.2 ↓8.9	48.6 ↓1.5	49.0 ↓1.1	20.5	20.3 ↓0.2	22.2 ↑1.7	20.5 =
Large models (15B–34B)												
StarCoder2-15B	65.9	51.2 ↓14.7	54.3 ↓11.6	62.8 ↓3.1	42.1	35.3 ↓6.8	39.7 ↓2.4	42.1 =	6.8	8.0 ↑1.2	7.3 ↑0.5	7.6 ↑0.8
Codestral-22B	72.6	58.5 ↓14.1	65.2 ↓7.4	70.7 ↓1.9	47.6	38.9 ↓8.7	45.0 ↓2.6	47.7 ↑0.1	23.0	22.5 ↓0.5	22.7 ↓0.3	22.6 ↓0.4
CodeLlama-34B	51.2	42.1 ↓9.1	45.1 ↓6.1	50.0 ↓1.2	37.9	29.7 ↓8.2	34.7 ↓3.2	36.3 ↓1.6	13.6	11.6 ↓2.0	13.1 ↓0.5	12.4 ↓1.2
DeepSeek-Coder-33B	72.0	61.6 ↓10.4	68.3 ↓3.7	73.8 ↑1.8	47.0	39.8 ↓7.2	45.4 ↓1.6	46.4 ↓0.6	20.3	19.4 ↓0.7	19.9 ↓0.4	18.8 ↓1.5
Qwen2.5-Coder-32B	85.4	73.2 ↓12.2	84.1 ↓1.3	86.6 ↑1.2	51.6	40.8 ↓10.8	48.6 ↓3.0	51.4 ↓0.2	32.0	30.6 ↓1.4	31.7 ↓0.3	31.2 ↓0.8
Reasoning models (API)												
GPT-5-mini	96.3	86.6 ↓9.7	89.0 ↓7.3	93.3 ↓3.0	49.7	44.6 ↓5.1	46.9 ↓2.8	49.5 ↓0.2	52.5	48.5 ↓4.0	52.5 =	53.2 ↑0.7
Claude Sonnet 4	95.7	86.0 ↓9.7	89.0 ↓6.7	95.7 =	53.0	44.1 ↓8.9	50.3 ↓2.7	53.5 ↑0.5	51.1	49.6 ↓1.5	49.9 ↓1.2	50.7 ↓0.4

losing 7.9% and 11.6%, respectively (Table 2). The nearly negligible drop for Qwen2.5-Coder-32B suggests that larger Qwen-family models are better calibrated to lexical synonymy. On MBPP, however, LV-induced drops shrink markedly across all tiers, ranging from only 1.5% to 3.2%, indicating that the shorter and more informal task description style of MBPP limits the impact of vocabulary-level ambiguity.

Surface-level formatting noise has minimal impact. SF mutations produce drops below 3.7% on HumanEval and below 1.6% on MBPP for virtually all models (Table 2), with several models showing marginal gains (e.g., DeepSeek-Coder-33B: +1.8% on HumanEval; Qwen2.5-Coder-32B: +1.2%). Across all benchmarks and models, typographical noise is the weakest of the three mutation types. This suggests that code-specialized LLMs are robust to surface-level corruption, extending the observations of Sclar et al. (Sclar et al., 2024), from general-purpose LLMs to code generation ones.

LiveCodeBench is nearly immune to all mutation types. In all models, Pass@1 deltas on LCB remain within ±2.0% for all defect types (Table 2), with many models recording marginal gains under US (e.g., StarCoder2-15B: +1.2%; CodeLlama-7B: +0.3%). This near-zero sensitivity stands in sharp contrast to HumanEval and MBPP, where drops are consistent and large. This is explained by the benchmark design: LCB problems include explicit sample input/output examples, so models can reconstruct the correct solution from the examples alone. US mutations remove one textual constraint while the sample I/O, which often encodes the same constraint implicitly, remains intact. Consequently, we find that descriptions with richer context leads to more robustness in code generation, highlighting an important boundary condition for the generality of our findings.

Model size does not relate to robustness; sensitivity is description-dependent. Comparison of small and large models shows that model size does not relate to model robustness. GPT-5-mini and Claude Sonnet 4 lose nearly 10% on code correctness due to US defects. Sensitivity appears to be description-dependent rather than capacity-dependent: Table 2 shows different decreases in Pass@1 across all pairs of model-defect types, suggesting that model size alone does not determine a clear trend.

> **RQ1 Takeaway.** Not all defect types harm code generation equally: Under-Specification degrades performance by up to 15% on HumanEval and MBPP, LV causes moderate losses, and SF is negligible. Benchmarks with richer, more realistic descriptions, such as LiveCodeBench, are inherently more resilient to description defects. Model size and architecture alone do not determine robustness.

5.2 Defect detection

The RQ1 results suggest that the robustness to defects is primarily determined by the characteristics of the input description, rather than the underlying models. Consequently, improving the correctness of code generation requires identifying and addressing defects before code generation.

Table 3 reports the classification performance of all five approaches (embedding projections in

Appendix Figure 4)

Zero-shot frontier models perform poorly in defect classification. GPT-5-mini achieves an F1 of 0.469 and an MCC of 0.281, barely above the chance for a four-class problem. Claude Sonnet 4 does slightly better (F1=0.518, MCC=0.359) but still falls well short of practical utility. This result is notable because both models achieve the highest Pass@1 scores in RQ1; strong code generation does not transfer to meta-reasoning about description quality in a zero-shot regime.

Few-shot prompting provides only limited gains for frontier models. Supplying four in-context examples yields marginal improvements for Claude Sonnet 4 (F1=0.557, MCC=0.412), while GPT-5-mini shows no consistent benefit and slightly degrades overall performance (F1 = 0.462, MCC = 0.276). This suggests that, unlike generation tasks, defect classification is not easily improved through in-context learning alone.

Even a frozen linear probe substantially outperforms few-shot frontier models. Training only a linear classification head on frozen Qwen2.5-Coder-1.5B embeddings yields F1=0.748 and MCC=0.665, a jump of 0.23% F1 over Claude Sonnet 4 while updating just 0.0004% of parameters.

SpecValidator with LoRA fine-tuning achieves the best classification performance, outperforming full fine-tuning. With only 0.14% of the parameters updated, LoRA achieves F1=0.804 and MCC=0.745, the best results across all classifiers. In contrast, complete fine-tuning (100% parameters) performs slightly worse (F1=0.788, MCC=0.722). The cause is likely overfitting: the dataset, while containing 8766 instances, is still constrained in size, and full fine-tuning may degrade the backbone's generalization across the input distribution. This result advocates for LoRA as the practical choice: superior accuracy, dramatically lower computations, and local deployability.

The ambiguity between clean and under-specified descriptions is the dominant error mode. Unlike LV and SF mutations, which introduce detectable surface signals, US mutations remove a constraint without leaving any positive textual trace: the resulting defective description is grammatically well-formed and stylistically indistinguishable from a clean one. This makes the clean/US boundary structurally harder to learn, as the classifier must detect the *absence* of information rather than the presence of an anomaly. Under all three fine-tuning regimes, SF and LV form well-separated clusters, while US and CLEAN exhibit the greatest residual overlap, a pattern visible in the embedding projections reported in Appendix Figure 4. The LoRA projection shows much better separation compared to the linear probe, yet the US/clean boundary remains challenging.

> **RQ2 Takeaway.** SpecValidator with LoRA fine-tuning of a 1.5B code model achieves F1=0.804 and MCC=0.745, outperforming both zero-shot and few-shot GPT-5-mini and Claude Sonnet 4 by a wide margin, and also outperforming full fine-tuning. The dominant confusion is between the clean and US classes.

5.3 Generalization to real (unseen) task descriptions

The previous RQ results evaluate the classification on test samples that include the mutated data. A natural concern is whether the classifier generalizes beyond this setting to detect real specification issues in the original descriptions. To investigate this, we analyze running the classification on 456 clean descriptions from the three benchmarks, we use the test data of RQ2 (40 HumanEval, 197 MBPP, 219 LiveCodeBench). For each description, the classifier predicts either clean or one of the three defect classes (LV, US, SF); we treat any non-clean prediction as a flagged defect. We then assess whether flagged defects are truly problematic by measuring Pass@1 on the original descriptions across all ten models and by performing a manual inspection.

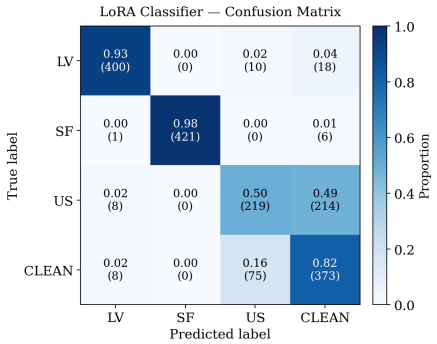

Figure 2: Confusion matrix of the LoRA classifier.

As shown in Figure2, 83 of the clean descriptions are classified as US (75) and LV(8). Table 4 reports the classification of clean descriptions from the test set. Of the 456 descriptions, SpecValidator flags 83 (18.2%) as defective, all of which

Table 3: Classification results for LV, SF, US, clean; macro-averaged metrics, reported as the mean over two seeds. GPT-5-mini and Claude Sonnet 4 are evaluated zero-shot and few-shot (4 examples) as baselines.

Classifier	Trainable %	Accuracy	Precision	Recall	F1	MCC
GPT-5-mini (zero-shot)	0%	0.46	0.490	0.460	0.469	0.281
Claude Sonnet 4 (zero-shot)	0%	0.51	0.550	0.520	0.518	0.359
GPT-5-mini (few-shot)	0%	0.45	0.520	0.450	0.462	0.276
Claude Sonnet 4 (few-shot)	0%	0.55	0.620	0.560	0.557	0.412
Linear Probe	0.0004%	0.75	0.750	0.745	0.748	0.665
LoRA Fine-tune (r=16)	0.14%	**0.81**	**0.810**	**0.805**	**0.804**	**0.745**
Full Fine-tune	100%	0.79	0.800	0.785	0.788	0.722

Table 4: SpecValidator on the 456 real unseen description. TN = correctly predicted CLEAN; FP = incorrectly flagged as defective (all as US). Specificity = TN / Total; FP rate = FP / Total.

Benchmark	Total	TN	FP	Specificity	FP Rate
HumanEval	40	40	0	100.0%	0.0%
MBPP	197	164	33	83.2%	16.8%
LiveCodeBench	219	169	50	77.2%	22.8%
Total	456	373	83	81.8%	18.2%

are Under-Specified: 0 from HumanEval, 33 from MBPP, and 50 from LiveCodeBench. The absence of false positives in HumanEval is consistent with its structured docstring format, which tends to be more explicit than the short MBPP descriptions or the challenging LCB problem statements.

The defects flagged by the classifier yield near-zero Pass@1 across all ten models, suggesting that they are originally defective rather than arbitrarily misclassified. Pass@1 on the flagged descriptions confirms the same pattern (see Appendix Table 6). For MBPP, 90.9% of flagged defects fail across all models (failing on an average of 6.7 out of 10); for LiveCodeBench, 98.0% fail (failing on 7.7 out of 10). These descriptions consistently prevent correct code generation despite being labeled clean in the ground truth benchmark.

72.7% of flagged MBPP descriptions are confirmed as under-specified upon manual inspection. Three researchers independently reviewed the 33 flagged MBPP descriptions, which are short enough for human assessment, and 24 of 33 (72.7%) cases are confirmed as under-specified. Representative examples are given in Appendix Table 5. The 50 LiveCodeBench cases could not be inspected due to the length and complexity of competitive programming statements, but their near-zero Pass@1 is consistent with the same pattern. Taken together, these results indicate that the classifier generalizes beyond the injected defects and indicates latent quality issues in the benchmark ground truth.

RQ3 Takeaway. SpecValidator flags 83/456 clean descriptions as under-specified, with near-zero Pass@1 across all models (90.9% failure on MBPP, 98.0% on LCB) and 72.7% manually confirmed as defective, demonstrating generalization beyond synthetic training data.

6 Conclusion

This study investigates the impact of task description defects on the LLM-based code generation and ways to automatically detect such defects. We find that defect impact depends jointly on defect type and benchmark specification structure: Under-Specification is the most damaging class (up to 15.3% Pass@1 drop, no model immune), Lexical Vagueness causes moderate and structure-dependent losses, and surface formatting noise is negligible for code-specialized LLMs. Description structure, not model capacity, is the primary predictor of robustness.

On the detection side, SpecValidator, a LoRA-fine-tuned 1.5B classifier, achieves F1 = 0.804 and MCC = 0.745, substantially outperforming frontier models prompted zero-shot or few-shot, confirming that code generation ability does not transfer to prompt meta-reasoning. Crucially, SpecValidator generalizes beyond its training distribution, surfacing real under-specification defects in three established benchmarks. We release our dataset of 6,573 defective task descriptions to support future work on prompt quality and benchmark design.

Limitations

Our study has several validity boundaries worth noting. Regarding internal validity, mutations are validated via a two-tier pipeline (automated LLM judge + manual sampling), with human annotators rating 97% of instances compliant; nonetheless, the subjectivity of "removing one constraint" in US mutations means some MBPP prompts may be borderline, and greedy decoding (temperature = 0), while ensuring reproducibility, may underestimate sampling variance. Regarding external validity, our findings are based on three Python benchmarks; results may not transfer directly to library-intensive tasks (BigCodeBench) or multi-language settings, and the near-immunity of LiveCodeBench to defects may not generalize to all structured specification formats. All open-source models use instruction-tuned variants; base models may exhibit higher sensitivity. Model-family specific effects (e.g., low LV sensitivity of Qwen2.5-Coder-32B) should be interpreted with caution. Known contamination in HumanEval and MBPP may artificially compress sensitivity; we include LiveCodeBench as a contamination-free control. Regarding construct validity, Pass@1 treats all failures equally and cannot distinguish near-misses from total failures. User prompts may also contain compound or mixed defects that do not fall cleanly into our three categories.

References

Anthropic. 2024. Claude 4 model card. Anthropic Documentation.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Tom B. Brown, Benjamin Mann, Nick Ryder, and 1 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 1877–1901.

Junkai Chen, Zhenhao Li, Xing Hu, and Xin Xia. 2026. Nlperturbator: Studying the robustness of code llms to natural language variations. *ACM Trans. Softw. Eng. Methodol.*, 35(4).

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021a. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Mark Chen, Jerry Tworek, Heewoo Jun, and 1 others. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient finetuning of quantized LLMs. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36.

Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering*, 50(9):2254–2268.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wenjie Zhang, Wenhu Chen, Kexin Bi, and 1 others. 2024. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Asma Hamidi, Ahmed Khanfir, and Mike Papadakis. 2025. Intent-based mutation testing: From naturally written programming intents to mutants. In *IEEE International Conference on Software Testing, Verification and Validation, ICST 2025 - Workshops, Naples, Italy, March 31 - April 4, 2025*, pages 347–357. IEEE.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 2790–2799.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, and 1 others. 2022. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024a. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Binyuan Hui, Jian Yang, Zeyu Cui, and 1 others. 2024b. Qwen2.5-coder technical report. *arXiv preprint arXiv*.

Naman Jain, Jiayi Han, Alex Gu, William Yang, Yiming Li, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination-free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Haoxiang Jia, Robbie Morris, He Ye, Federica Sarro, and Sergey Mechtaev. 2025. Automated repair of ambiguous problem descriptions for llm-based code generation. *arXiv preprint arXiv:2505.07270*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72.

Maya Larbi, Amal Akli, Mike Papadakis, Rihab Bouyousfi, Maxime Cordy, Federica Sarro, and Yves Le Traon. 2025. When prompts go wrong: Evaluating code model robustness to ambiguous, contradictory, and incomplete task descriptions. *arXiv preprint arXiv:2507.20439*.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–23.

Yujia Li and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*. ArXiv:2305.01210.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*.

Anton Lozhkov, Raymond Li, Vaibhav Chaudhary, and 1 others. 2024. Starcoder2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the robustness of code generation techniques: An empirical study on GitHub Copilot. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 2149–2160. IEEE.

Brian W. Matthews. 1975. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta*, 405(2):442–451.

Bertrand Meyer. 1985. On formalism in specifications. *IEEE Softw.*, 2(1):6–26.

Mistral AI. 2024. Codestral: Hello, world! Technical report.

Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering*, 1(FSE):2332–2354.

OpenAI. 2025. Gpt-5 technical report. https://openai.com. Accessed: 2026.

Shunyu Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*.

Fazle Rabbi, Zishuo Ding, and Jinqiu Yang. 2025. A multi-language perspective on the robustness of LLM code generation. *arXiv preprint arXiv:2504.19108*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Adi, Jingyu Liu, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. 2024. Quantifying language models' sensitivity to spurious features in prompt design. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Atsushi Shirafuji, Yutaka Watanobe, Takumi Ito, Makoto Morishita, Yuki Nakamura, Yusuke Oda, and Jun Suzuki. 2023. Exploring the robustness of large language models for solving programming problems. *arXiv preprint arXiv:2306.14583*.

Mohammed Latif Siddiq, Simantika Dristi, Joanna C.S. Saha, and Joanna Santos. 2024. The fault in our stars: Quality assessment of code generation benchmarks. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 201–212. IEEE.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605.

Axel van Lamsweerde. 2009. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley.

Andreas Vogelsang, Alexander Korn, Giovanna Broccia, Alessio Ferrari, Jannik Fischbach, and Chetan Arora. 2025. On the impact of requirements smells in prompts: The case of automated traceability. In *Proceedings of the IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE/ACM.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, and 1 others. 2023. ReCode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 13234–13274.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35.

Jie J.W. Wu and Fatemeh H. Fard. 2025. HumanEval-Comm: Benchmarking the communication competence of code generation for LLMs and LLM agents. *ACM Transactions on Software Engineering and Methodology*, 34(7).

Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. 2024. EvoEval: Evolving coding benchmarks via LLM. In *Proceedings of the Conference on Language Modeling (COLM)*.

Kaijie Zhu, Jindong Wang, Jiaheng Zhou, Zichen Wang, Hao Chen, Yidong Wang, Linyi Yang, Wei Ye, Neil Zhenqiang Gong, Yue Zhang, and Xing Xie. 2024. Promptrobust: Towards evaluating the robustness of large language models on adversarial prompts. In *Proceedings of the 1st ACM Workshop on Large AI Systems and Models with Privacy and Safety Analysis (LAMPS)*, pages 57–68. ACM.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 14 others. 2025. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *Proceedings of the International Conference on Learning Representations (ICLR)*. ArXiv:2406.15877.

A Implementation details

A.1 Selected LLMs

To study the impact of defects, we choose a diverse set of code generation models to ensure that our findings are not artifacts of a single architecture or training regime. Models are grouped into the following three categories/families:

- **Small open-source models** ($\sim$7B parameters): Qwen2.5-Coder-7B (Hui et al., 2024b), DeepSeek-Coder-6.7B (Guo et al., 2024), and CodeLlama-7B (Rozière et al., 2023). These models are representative of deployable, resource-constrained settings and are widely used as baselines by the literature.

- **Large open-source models** (15B–34B parameters): Qwen2.5-Coder-32B (Hui et al., 2024b), DeepSeek-Coder-33B (Guo et al., 2024), CodeLlama-34B (Rozière et al., 2023), Codestral-22B (Mistral AI, 2024), and StarCoder2-15B (Lozhkov et al., 2024). This tier allows us to examine whether larger capacity yields greater robustness to prompt mutations, independent of proprietary training data or alignment procedures.

- **Closed-source LLM-based systems**: GPT-5 mini (OpenAI, 2025), and Claude Sonnet 4 (Anthropic, 2024). These represent the current state-of-the-art in code generation and serve as upper-bound references. Their inclusion enables a direct comparison between frontier models and open-source alternatives under identical evaluation conditions.

Collectively, this selection spans five model families, two provider categories (open-source and proprietary), and two scale regimes, enabling controlled cross-family and cross-scale comparisons.

A.2 Metrics

Pass@1 (Chen et al., 2021b) is used as the primary correctness metric, as it reflects the real-world scenario where a developer accepts the model's first suggestion. This metric is the de facto standard in code generation evaluation (Chen et al., 2021b; Austin et al., 2021) and enables a direct comparison with previous work.

For the four-class mutation classification task (LV, SF, US, CLEAN), we report the Matthews Correlation Coefficient (MCC) (Matthews, 1975), macro-F1, precision, recall, and accuracy. MCC is preferred over accuracy when there is class imbalance as it accounts for all entries in the confusion matrix.

A.3 Technical Details

Inference. All models are run with greedy decoding and temperature $= 0$), ensuring that the outputs are theoretically deterministic given a fixed prompt. Under this setting, each model produces exactly one outcome per problem, and re-running the same experiment yields identical results.
The reasoning models (GPT and Claude) are queried via the OpenAI and Anthropic batch APIs, while open-source models are served locally on NVIDIA A100 GPUs in float16 precision. Generated solutions are extracted via markdown fenced code block parsing, with fallback to model-specific delimiters (e.g., CodeLlama's `[PYTHON]` tags). Each solution is executed in an isolated subprocess with a 20-second timeout. The code and data are available. [3]

LoRA fine-tuning. We apply LoRA (Hu et al., 2022) to the attention projection layers with rank r=16 and α=32, which prior work has identified

[3] Replication package: `https://anonymous.4open.science/r/prompt_defects-detection`

as an effective configuration for code model adaptation (Hu et al., 2022; Dettmers et al., 2023; Houlsby et al., 2019). Training uses AdamW (Loshchilov and Hutter, 2019) with learning rate 2×10^{-4}, weight decay 10^{-4}, cosine decay scheduling, and 100 warmup steps. An effective batch size of 16 is achieved via gradient accumulation. Early stopping (patience = 4) is applied based on the validation performance to prevent overfitting. All finetuning experiments use both seeds 42 and 123456; the results are averaged.

Prompt used for zero-shot classification. *(4 labeled examples are prepended in the few-shot setting.)*

```
You are a code-benchmark quality auditor.
Classify the following coding prompt into
exactly one of:

  LV   - The prompt uses vague or imprecise
wording.
  SF   - The prompt contains syntax or
formatting errors.
  US   - The prompt is missing a constraint or
condition.
  CLEAN - The prompt is complete and
well-formed.
```

B Additional tables and figures

Table 5: Example UNDER-SPECIFIED (real) defects detected by SpecValidator.

ID	Prompt	Ambiguity
MBPP 277	Filter a dictionary based on values.	By what condition? A threshold?
MBPP 466	Find the peak element in the given array.	Local peak or global max? Index or value?
MBPP 382	Find the number of rotations in a circularly sorted array.	Left or right rotation? Ascending or descending?

Table 6: Pass@1 results on CLEAN test samples predicted as non-CLEAN by the classifier. **F** = pass@1 False (failing); **P** = pass@1 True (passing). The *Misclassified* row aggregates across all samples; the parenthetical reports the average number of models on which a misclassified sample fails.

Model	N	F	P	%F
MBPP				
CodeLlama-34B	33	25	8	75.8
CodeLlama-7B	33	24	9	72.7
StarCoder2-15B	33	23	10	69.7
DeepSeek-Coder-33B	33	21	12	63.6
Qwen2.5-Coder-32B	33	20	13	60.6
DeepSeek-Coder-6.7B	33	20	13	60.6
Codestral-22B	33	20	13	60.6
GPT-5-mini	33	19	14	57.6
Qwen2.5-Coder-7B	33	15	18	45.5
Claude Sonnet	33	15	18	45.5
Misclassified (Fail on avg 6.7/10 models)	33	30	3	90.9
LiveCodeBench				
StarCoder2-15B	50	49	1	98.0
CodeLlama-7B	50	45	5	90.0
CodeLlama-34B	50	43	7	86.0
DeepSeek-Coder-33B	50	43	7	86.0
DeepSeek-Coder-6.7B	50	41	9	82.0
Codestral-22B	50	41	9	82.0
Qwen2.5-Coder-7B	50	39	11	78.0
Qwen2.5-Coder-32B	50	36	14	72.0
GPT-5-mini	50	27	23	54.0
Claude Sonnet	50	15	35	30.0
Misclassified (Fail on avg 7.7/10 models)	50	49	1	98.0

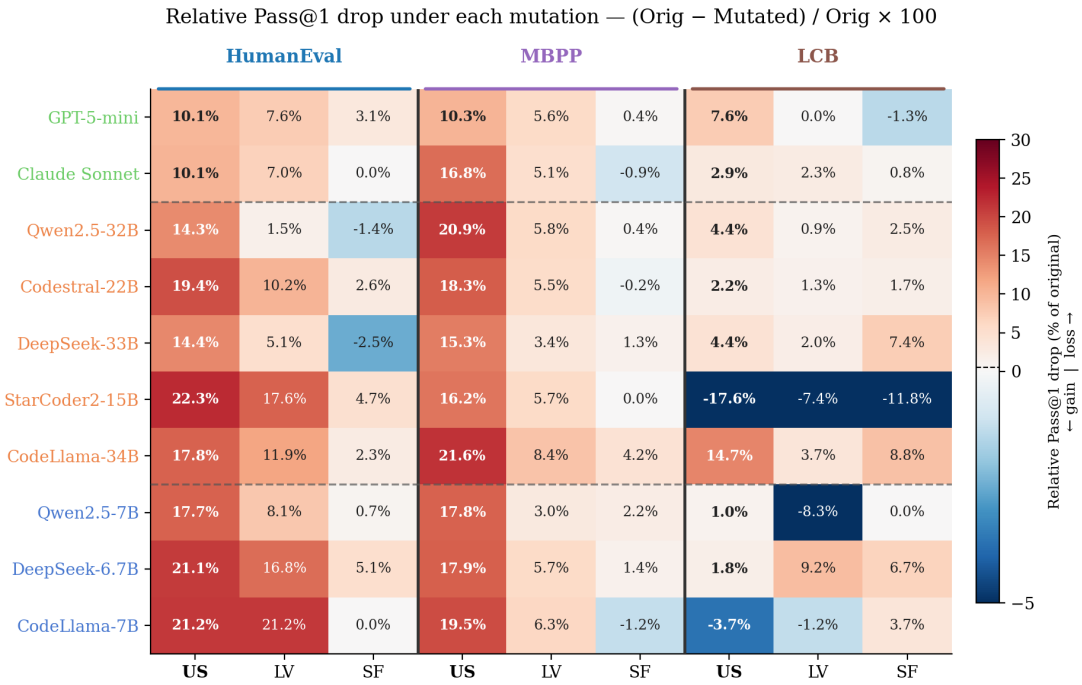

Figure 3: Pass@1 drop (%) with all defect types in all models and benchmarks, computed as (Orig−Mutated)/Orig× 100. Red cells indicate performance degradation; blue cells indicate a slight gain. Column labels: **US** = Under-Specification; **LV** = Lexical Vagueness; **SF** = Syntax and Formatting.

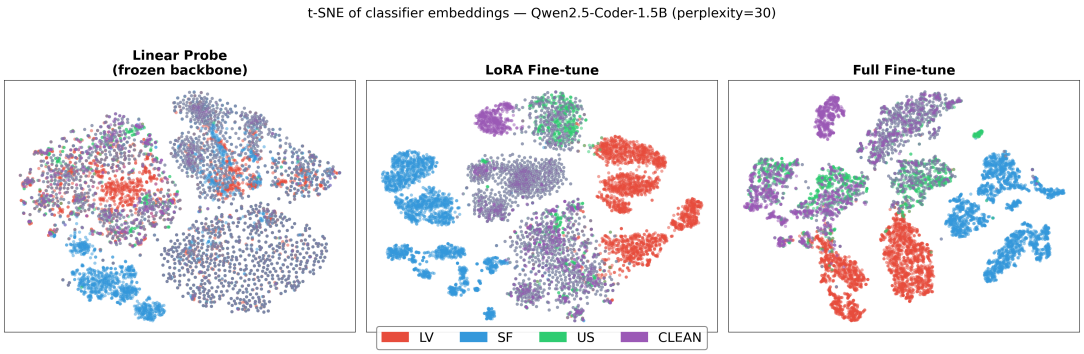

Figure 4: t-SNE projections (van der Maaten and Hinton, 2008) of the embedding space of the classifiers (Qwen2.5-Coder-1.5B backbone). From left to right: Linear Probe (frozen backbone), LoRA fine-tune, and Full fine-tune.

Table 7: Representative examples of the three mutation strategies. **US**: removes a constraint; **LV**: paraphrases vocabulary and renames identifiers; **SF**: introduces typographical and formatting noise.

Mutation	**Prompt**
HumanEval / 5	
Original	`def intersperse(numbers: List[int], delimeter: int) -> List[int]:` *"Insert a number 'delimeter' between every two consecutive elements of 'numbers'."* `»> intersperse([1,2,3], 4)` ⇒ `[1,4,2,4,3]`
US	`def intersperse(numbers: List[int], delimeter: int) -> List[int]:` *"Insert a number 'delimeter' into the input list 'numbers'."* `»> intersperse([1,2,3], 4)` ⇒ `[1,4,2,4,3]`
LV	`def place_between(items, filler):` *"Put a value 'filler' between neighboring entries of the sequence."* `»> place_between([1,2,3], 4)` ⇒ `[1,4,2,4,3]`
SF	`def intersperse(numbers: List[int), delimter: int) -> List[int) "' Insert a number 'delimter' between every two consecutive elements . . .`
MBPP / 12	
Original	*"Write a function to sort a given matrix in ascending order according to the sum of its rows."*
US	*"Write a function to sort a given matrix according to the sum of its rows."*
LV	*"Write a function to arrange a given matrix in ascending order according to the total of its rows."*
SF	*"Write a function to sort a given matrix in ascdcending order according to the sum of its rows."*
LiveCodeBench / 2811	
Original	*"An array of distinct positive integers is called a k-avoiding array if no pair sums to k. Return the minimum possible sum of length n."* `n=5,k=4⇒18 n=2,k=6⇒3`
US	*"An array of integers is called a k-avoiding array if no pair sums to k. Return the minimum possible sum of length n."* `n=5,k=4⇒18 n=2,k=6⇒3`
LV	*"A list of different positive numbers where no two add up to k. Return the smallest total of length n."* `n=5,k=4⇒18 n=2,k=6⇒3`
SF	`INPUT;{You are given two integres` n `and` k `. . . **INVALID****OCUMENT**}**`