

Teaching Coding Assistants What to Change: Fine-Tuning from Pairwise Preferences

Hongmo Tao
Graduate School of
Science and Technology,
Meiji University, Japan
ce251024@meiji.ac.jp

Hiroyuki Kamata
School of
Science and Technology,
Meiji University, Japan

Abstract

Real code-editing requests typically mix the target program with other related materials, such as test results or documentation. However, models often make mistakes by incorrectly changing these supporting materials or missing the actual code that needs correction. To investigate this issue, we created multiple task scenarios where the code needing changes and the intended behavior remained constant, while only the surrounding supporting texts varied. This approach allowed us to test if supporting texts that resemble code can confuse the model, making it difficult to identify the correct code to modify. To address this, we developed a simple yet effective training method by showing the model examples of correct and incorrect edits. Specifically, Correct examples only modify the intended code, whereas incorrect examples mistakenly alter or duplicate supporting texts. Our results suggest that reliable code editing depends on the model’s early ability to distinguish between guidance materials and the actual code requiring changes.

1 Introduction

Large language models have become widely adopted as code-editing assistants (Chi et al., 2025), enabling developers to request tasks like fixing bugs, migrating APIs, repairing failing tests, or adapting code snippets using natural-language instructions. However, these requests seldom arrive neatly structured; developers often blend the target code with other contextual material. For instance, a typical editing request might intermingle the problematic source code with error traces, test outputs, or related documentation snippets, creating prompts where multiple types of code-like text coexist.

The mixture of diverse elements within a single editing request poses a fundamental challenge: the model must determine the intended role of

each pasted fragment. Developers frequently include text such as configuration keys or output examples when describing software issues on platforms like Stack Overflow. While these fragments superficially resemble executable code, their purposes often contrast sharply—one might specify the exact code needing modification, while another merely serves as an illustrative example. This subtle yet critical distinction often leads models astray, prompting them to misinterpret the fragment’s role even when correctly identifying its relevance (Zverev et al., 2025). Anticipating and addressing this misinterpretation is the central focus of our study. For instance, Section 3.1 illustrates how models may incorrectly treat illustrative outputs as literal values to embed directly into the code, rather than using them as a guideline for implementing the desired transformation (Yi et al., 2025).

Research on prompt injection provides a useful perspective by highlighting that instruction-following involves not only understanding natural language, but also discerning which textual elements should guide a model’s response (Li et al., 2024). Our setting, however, adds a code-editing-specific binding problem. The pasted span is usually not malicious and not irrelevant: a test output, trace, example, or documentation fragment may be useful evidence that should constrain the patch. The model must therefore do more than separate instruction from data; it must decide which span gives the editing goal, which span supplies evidence, and which span is the object that may actually be modified. In typical debugging requests posted on platforms like Stack Overflow, developers often paste snippets of code or error traces adjacent to the code segment they wish to edit. Although these pasted fragments appear actionable, their actual role is to provide context or evidence supporting the requested edit, rather than serving as the direct target of modification. Our study

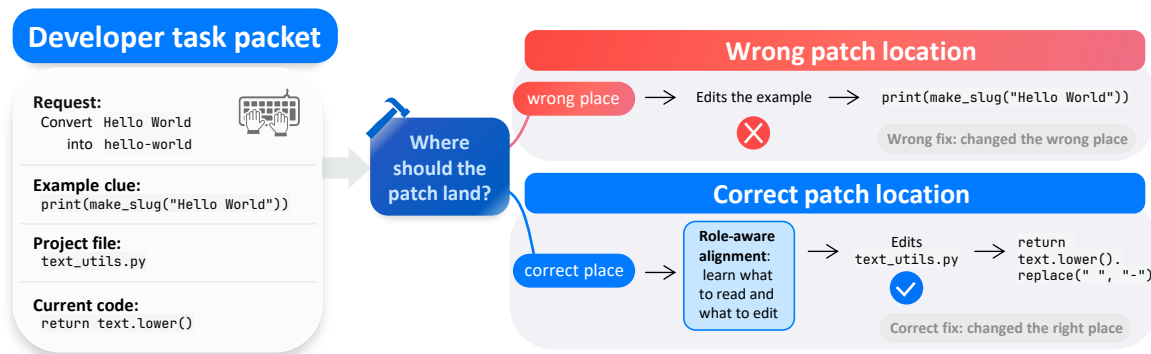


Figure 1: Developer requests can place a behavior hint next to project code. This schematic traces a simple mix-up: the model may copy or rewrite the small snippet that shows what should happen, instead of updating the target code that has to make it happen.

specifically investigates this confusion, defining it as *instruction-code boundary ambiguity*.

When developers ask models to edit code, they often place the instruction alongside various supporting details—like logs or documentation notes—to clarify their request. Yet ironically, this added context might confuse rather than clarify. For instance, language models may incorrectly rewrite logs or insert illustrative outputs directly into the code, instead of correctly updating the actual target code (Liu et al., 2024). Such mistakes happen because the model, while correctly identifying the fragments as relevant, misunderstands their intended role—treating supplementary examples or explanations as code to directly modify, rather than cues guiding the necessary change.

To address this confusion, we treat distinguishing editable code from other instructions as a preference-learning problem. Basic formatting—such as adding headers or clearly marking code boundaries—can initially help the model identify which sections might contain editable content. However, formatting alone isn’t enough. Models might still misinterpret well-marked fragments like example outputs or stack traces, treating them incorrectly as targets for editing. The root of this issue is not visibility, but role misinterpretation. Prompt-injection defense strategies highlight a similar point: training signals can help models prefer intended instructions over distracting or misleading content (Chen et al., 2025a). Following this insight, we introduce lightweight adapters aimed at shaping model preferences. Specifically, these adapters encourage edits that target the intended code sections, while discouraging incorrect modifications to explanatory text or verbatim copying of provided examples.

2 Related Work

2.1 Instruction competition and role assignment

Code-editing requests often place multiple text blocks side-by-side, each potentially influencing how a model understands what needs to be edited. Even though simple visual markers like fences or headers might help distinguish code from contextual material, they are not always sufficient. Prompt-injection research tackles a more aggressive form of this problem, where injected external commands actively compete with legitimate user instructions. Methods such as SecAlign address this by treating it as a preference-optimization issue: models are trained explicitly to favor the intended instruction rather than distracting injected content (Chen et al., 2025b).

The code-editing setting differs in what the model must do with the non-target span. In prompt-injection defense, the unsafe span is typically something to resist or ignore. In code editing, the support span is often benign and relevant: the model should read it, use it to infer the intended transformation, and still avoid modifying or copying it. This creates a target-binding problem rather than a pure instruction-selection problem. A pasted test, log, example, or documentation fragment may influence the model as if it were something to act on, even when it is only part of the problem report. Dimension-reduced reconstruction-map learning highlights a related trade-off between input simplification and information loss (Zhang et al., 2025a); here, we retain the prompt and distinguish the roles of its parts.

2.2 Paired instruction variants and lightweight adaptation

Paired-instruction approaches provide useful insights for addressing this challenge. MUFIN, for example, pairs one input with varied task instructions, teaching models to flexibly interpret the same content differently (Lou et al., 2024). Conversely, code-oriented instruction generation shows that different instructions can highlight distinct roles of the same snippet (Wu et al., 2025). Together, these findings naturally motivate training with paired variations: the edit task remains constant, but its surrounding instructions shift to help the model clarify context boundaries.

We apply this paired-variant approach to code editing tasks. Specifically, we keep the target code and the requested change fixed, then create request variants by adjusting how supporting context (e.g., a test or log) is presented. This setup allows training to directly contrast edits applied correctly to the intended code versus edits mistakenly influenced by supporting evidence.

Fully fine-tuning a pretrained code model could be unnecessarily costly for addressing the boundary distinction problem, and might negatively impact its pretrained coding capabilities. Parameter-efficient methods like LoRA offer a better alternative, as they train lightweight adapters while preserving pretrained knowledge by freezing the original model weights (Hu et al., 2022). We therefore adopt this lightweight adaptation approach, training preference signals that guide the model to correctly focus edits on the intended target code rather than mistakenly altering context or supportive materials. The separation of offline training from reuse on new requests has a statistical analogue in amortized multistate inference (Zhu and Zhang, 2026), which learns reusable parameter posteriors rather than editing policies.

3 Problem Formulation

3.1 Boundary ambiguity in code editing

Consider a Python edit request for a user-record formatter. The request gives the following example output from product documentation:

```
USER-0042: ADA LOVELACE
```

This example is only showing the output shape that the function should produce. It should not be copied as the exact string that the function returns every time. The target code is:

```
def format_user(user):
    return f"{user['id']}: {user['name']}"
```

The practical rule is simple: do not hard-code the sample output; implement the rule behind it. The function should still use the input user’s own fields, pad the numeric ID to four digits, add the USER-prefix, and convert the user’s name to uppercase.

```
def format_user(user):
    return f"USER-{user['id']:04d}: {user['name'].upper()}"
```

This example illustrates the boundary problem. USER-0042: ADA LOVELACE tells the model what the output should look like, but it is not the code to edit and not the exact answer to return every time. The model needs to read 0042 as “take the current user’s ID and pad it to four digits,” and read ADA LOVELACE as “take the current user’s name and uppercase it.” In other words, the function body is the part to change; the documentation example is only a guide for the output style.

A mistaken edit may instead copy the documentation example into the function:

```
def format_user(user):
    return "USER-0042: ADA LOVELACE"
```

This patch can look right at first glance because it matches the example. However, it only works for that one example and fails as soon as the input changes:

```
format_user({"id": 7, "name": "grace hopper"})
# desired: USER-0007: GRACE HOPPER
# pasted-constant output: USER-0042: ADA LOVELACE
```

Real code-editing requests and developer Q&A threads frequently embed the code to edit within other code-like material, such as examples or documentation notes. Humans can usually distinguish which part is the actual target and which merely provides context; however, models may confuse these roles, especially when surrounding text resembles actionable code. We define the task of distinguishing these roles as *instruction-code boundary disambiguation*: identifying which portions of a mixed request are instructions or supporting context, and which portion represents the actual code to modify.

In the formatter example, the mistake is not just an incomplete patch. It is an *example-as-constant boundary confusion*: the model treats a documentation example that should describe a general rule as if it were the final implementation.

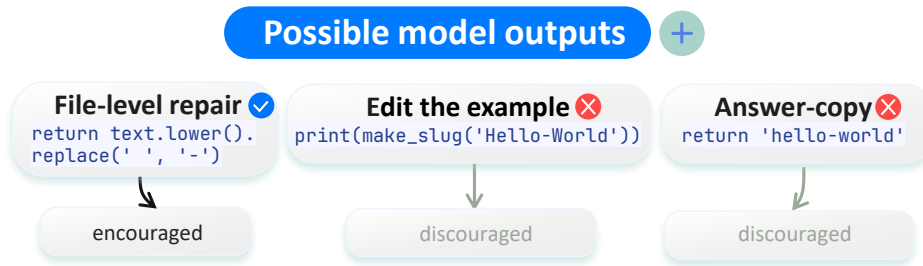


Figure 2: During alignment training, each code-editing request is associated with several candidate outputs generated by the model. These candidates fall into two distinct groups: one includes outputs that correctly modify the intended target code, which the model is trained to favor; the other includes outputs that mistakenly edit supporting material—such as examples or explanatory context—or directly paste example outputs as fixed return strings, which the model is explicitly taught to avoid.

3.2 Research questions

We focus on two questions.

RQ1. When the required edit to the target code is already known and fixed, does placing a code-like example inside the request make the model more likely to copy the example or edit the wrong place?

RQ2. Can training the model to distinguish between correctly editing the intended target code and mistakenly copying an example or editing an irrelevant section reduce such errors, without hurting the model’s general code-generation ability?

Plain-language rewrites test whether the issue is merely instruction wording, while visible separators test whether simple visual separation is sufficient. The main comparison asks whether code-like support material is assigned the wrong role.

3.3 Operationalization

For each benchmark task, which already pairs a piece of code with a natural-language edit request (Cassano et al., 2024), we keep the target code and the correct patch unchanged and create several matched versions of the request. One version explains the change in plain language. Another version includes a concrete output example, like the formatter example above. A third version rewrites the same idea in ordinary language without relying on the concrete example. A fourth version presents the example as documentation context rather than as the main instruction.

The key comparison is the model’s behavior across these versions. If the model succeeds when the rule is written in plain words but fails when the same rule is shown as an output example, then the problem is probably not missing Python knowledge. It is misunderstanding what the example is for. If training with correct edits and these tempt-

ing wrong edits reduces copying and wrong-place changes, then the model has learned a clearer distinction between the code to edit and the text that only explains the edit.

4 Data and Prompt Variants

4.1 Benchmark-derived editing problems

To clearly observe how code-like context can lead to editing mistakes, we first set up controlled experiments using existing code-editing benchmark tasks (Cassano et al., 2024). Specifically, we keep the desired code edits unchanged while varying how the instructions are presented. Our dataset is derived from established benchmarks such as CodeEditorBench (Guo et al., 2024b). Matching variants within each edit group limits between-task heterogeneity, analogous to within-cohort treatment comparisons used to control background variability in experimental studies (Shannon et al., 2025).

4.2 Instruction variants

Using the formatter example from Section 3.1, the four instruction forms look as follows.

These four instruction variants require the exact same final edit. We generate the variants with the OpenAI API model gpt-4o-2024-08-06 using a constrained JSON schema. For each edit group, the API receives the original instruction, the target-code span, the gold edited code, and the requested variant type, and it may rewrite only the surrounding instruction and supporting context. It is not used to generate gold patches or edit-success labels. We render four requests per edit group, producing 12,960 variant prompts for 3,240 edit groups. Automatic validation rejects a variant if it changes identifiers in the target code, alters the gold patch, drops the required support span, or introduces a

Version	Instruction
Clean	Format each user as a USER- record with a four-digit zero-padded ID and an uppercase name.
Code-like	Use the example shape USER-0042: ADA LOVELACE as the formatting pattern.
Natural-language rewrite	Prefix the record with USER-, pad the user’s numeric ID to four digits, and uppercase the user’s name.
Role-shifted	A documentation note shows the expected shape as USER-0042: ADA LOVELACE; update only the Python function so it produces that shape for each input user.

Table 1: Matched instruction variants for the same edit.

different edit requirement. In the final generation pass, 12,064 variants pass validation on the first attempt, 823 after one regeneration, and 73 after a final regeneration, leaving all accepted variants with the same target code and gold repair.

5 Role-Aware Boundary Alignment and Results

5.1 Method

Prompt-injection defenses motivate our training design by framing instruction following as a problem of resolving competition among prompt spans: a model must learn which text should control its response and which text should be treated as context (Chen et al., 2025b). We use this as a training analogy rather than treating code editing as ordinary prompt injection. In our setting, the competing span is often useful evidence, so the desired behavior is not to ignore it, but to bind it to the correct operation: read the evidence, infer the transformation, and apply the patch only to the target program. Role-aware boundary alignment therefore aims to induce a target-aware editing preference: the model should condition patch generation on pasted evidence without allowing that evidence itself to become the edit target. In the formatter example from Section 3.1, USER-0042: ADA LOVELACE should constrain the transformation inside `format_user`; a boundary-confused edit may instead incorporate that example into the implementation. A full code-model fine-tune would be an unnecessarily heavy intervention and may also risk degrading the strong code-generation and instruction-following abilities already present in code LMs (Aggarwal et al., 2025). We therefore freeze the base model and train lightweight adapters to favor role-consistent patches.

Because the boundary error involves a preference among plausible edits, we apply Direct Preference Optimization after a supervised warm-up. DPO directly optimizes a language model from preference comparisons without fitting a sep-

arate reward model (Rafailov et al., 2023), which matches our setting: the model must prefer the patch that updates the target program over patches that are pulled toward nearby pasted text. In training, each example is built around this same contrast. The request contains an instruction, supporting evidence, and target code. A correct answer uses the evidence to update the target code. An incorrect answer instead copies evidence into the target program, edits supporting text that should remain unchanged, or inserts a nearby literal, example, or trace into the program. The supervised warm-up contains four rendered requests for each of the 2,040 training groups, yielding 8,160 supervised instances. For preference training, each rendered request is paired with two rejected candidates, giving 16,320 chosen–rejected pairs. The rejected candidates are built from two sources: rule-based transformations that hard-code or edit the support span, and base-model outputs that match the same failure patterns under the failure protocol in Appendix B. During preference training, the correct patch is compared against these tempting wrong patches. This turns the paper’s main intuition into a training signal: pasted material may explain the edit, but the edit itself should land on the target program.

To make the comparison fair, the ordinary code-edit SFT baseline uses the same adapter architecture and the same number of supervised training instances. The difference is what the examples ask the model to practice. In the baseline, each example is a standard code-edit request: it gives the code to be changed, states the requested change, and trains the model to return the same single edited-code-block format used at evaluation time. It does not include contrastive cases in which the model must choose between revising the target code and copying pasted evidence or returning a rewritten support span. This comparison tests whether our method improves robustness by training the model to keep its attention on the intended code region.

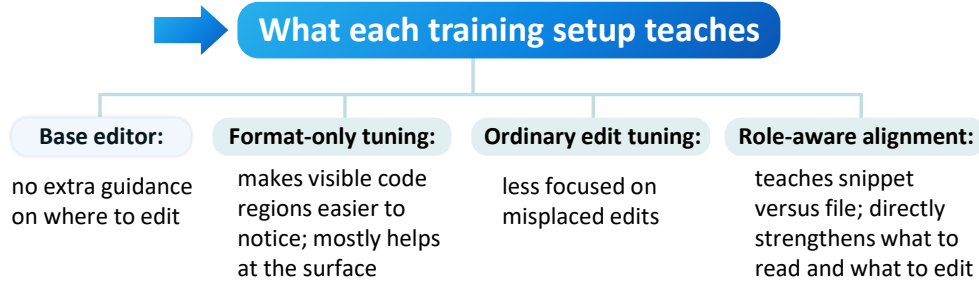


Figure 3: We compare several methods for training models to handle code-editing requests containing multiple code-like sections. First, we evaluate the unadapted instruction-tuned checkpoint, which already has general code-generation and instruction-following ability but has not been trained with our boundary-disambiguation objective. Then, we add basic formatting cues—such as fences, headers, or separators—to visually separate the target code from other text. Next, we apply ordinary edit-tuning, providing additional training examples that demonstrate the edited-code output format. Finally, we introduce role-aware alignment, which explicitly trains the model to recognize the intended editing target and avoid copying evidence or returning a rewritten support span as the edit.

5.2 Evaluation setup

We evaluate five open-weight instruction-tuned code checkpoints and report macro-averages unless otherwise noted. We refer to the checkpoint before any training on our data as the *unadapted checkpoint*, rather than as a pretrained base model. The compared systems are the unadapted checkpoint, a format-only baseline, ordinary code-edit SFT, and role-aware boundary alignment. The format-only baseline uses one shared no-training template selected by mean development-set Edit Success across the five checkpoints: section headers plus a fenced target-code block.

The main test slice contains 600 stratified editing problems. For each problem, we evaluate matched clean, code-like, and natural-language rewrite variants under a fixed prompt template. Edit Success and General Pass@1 are defined in Appendix A.4.

5.3 Main results

Table 2 reports edit success macro-averaged across the five models under matched instruction variants.

Table 2 reports the full condition-wise results, while Figure 4 highlights the trade-off on the code-like variant. The key comparison is between clean and code-like variants under the fixed prompt template. The base model drops from 58.7 on clean instructions to 49.5 on code-like instructions, a 9.2-point loss. Because the required target-code edit is held fixed across variants, this drop points to a role-disambiguation burden introduced by the code-like support span. Natural-language rewrites recover part of the loss, raising the base model from 49.5 to 55.0. In practical terms, a trace, output sketch,

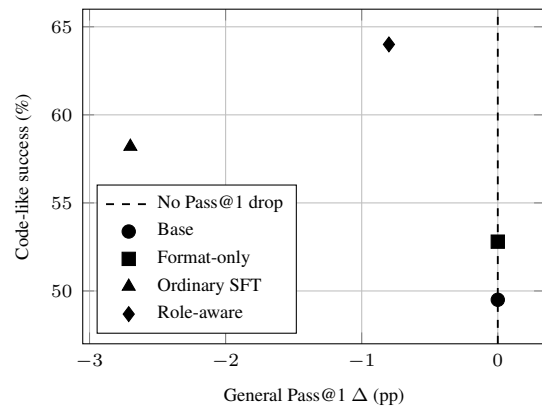


Figure 4: Trade-off on the code-like variant. The x-axis reports the change in overall coding skill. The y-axis shows how often the model correctly edits the intended target code, instead of mistakenly editing nearby examples or explanations. Dots near the upper-right corner represent models that are better at changing the right code without getting confused by nearby code-looking examples or explanations.

regex, or short literal near the requested change can become a competing action target: the model may copy it, rewrite it, or follow it too literally instead of updating the target program. This pattern is closer to a mild instruction-steering problem than ordinary patch-writing difficulty: pasted evidence gains too much influence over what the model edits. Formatting also helps, but only modestly: the fixed format-only template raises code-like success to 52.8, leaving a large gap to clean instructions.

Role-aware alignment gives the strongest targeted gain. It raises code-like success from 49.5 to 64.0, a 14.5-point gain over the base model. Relative to ordinary code-edit SFT, it adds 5.8 points, or

System	Clean	Code-like	NL rewrite	Pass@1 Δ
Base model	58.7	49.5	55.0	0.0
Format-only	60.0	52.8	56.4	0.0
Ordinary code-edit SFT	64.1	58.2	61.3	-2.7
Role-aware alignment	65.3	64.0	64.8	-0.8

Table 2: Edit success on the primary controlled test slice under matched instruction variants. Scores are percentages macro-averaged across the five checkpoints; Pass@1 Δ is in percentage points.

about a 10% relative improvement. The observed General Pass@1 drop is 0.8 points rather than 2.7. The clean-to-code-like gap shrinks from 9.2 points in the base model to 1.3 points after alignment. This matches the method’s intended effect: the model still has to find and apply the instruction, but it becomes less likely to let a nearby example, trace, or literal take control of the edit.

5.4 Diagnostic analysis

For the code-like variant, the failure protocol in Appendix B gives a diagnostic view of where the answer first goes off course. One path is target first, patch wrong: the model reaches the intended code region but writes an incomplete or incorrect repair. The other path is evidence first, edit wrong: the model acts on a pasted log, example, documentation note, or other span that should have guided the edit rather than received the edit. Table 3 reports the distribution of primary failure labels among failed outputs, using the failure categories defined in Appendix B.

Failures where the answer follows the wrong pasted span fall from 41.7% in the base model to 18.3% after role-aware alignment. In the formatter example from Section 3.1, this is the difference between using USER-0042: ADA LOVELACE to infer the padding and uppercasing rule, and putting that exact string into the return statement; it is also the difference between updating format_user and rewriting the documentation example while the function remains wrong. In real developer requests, the same pattern can come from traces, tests, examples, or documentation notes. The larger share of “right region, wrong patch” among the remaining failures is a useful shift in the error pool: after alignment, the model more often reaches the target code first, so the remaining failures more often come from ordinary patch quality.

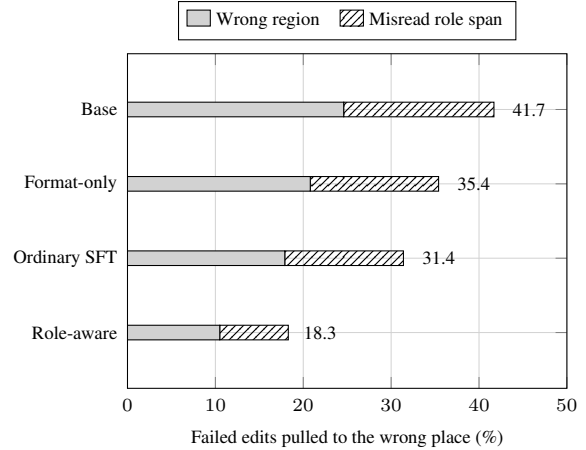


Figure 5: Among failed edits, this figure counts cases where the output follows the wrong pasted span before the patch is written: editing context that should only be read, or treating an example-looking span as the program to modify. Each bar adds these two mistakes together. A larger bar-end number means that more failed edits come from changing the wrong thing rather than writing a bad patch in the right place.

6 Discussion and Conclusion

6.1 Discussion

Return to the example from Section 3.1, but only as a small illustration of a broader pattern. The point of the example is not that this one user-record task is hard. The model may already know how to make the small code change. The paper’s central answer is no: code-editing failures do not always come from weak patch writing. They can also come from an answer-looking span that captures the model’s attention and pulls the edit toward the wrong place. In the example, the sample output should guide the target program, not become the program’s fixed answer. In real developer requests, the same pull can come from any pasted support material that looks close to code. The material is relevant, but relevance is not the same as permission to copy or edit it. This is the key code-editing distinction from ordinary instruction-data separation: the support span should influence the edit without becoming

System	Wrong region	Misread role span	Right region, wrong patch	Other
Base model	24.6	17.1	36.5	21.8
Format-only	20.8	14.6	39.1	25.5
Ordinary code-edit SFT	17.9	13.5	43.8	24.8
Role-aware alignment	10.5	7.8	52.0	29.7

Table 3: Failure types in the code-like setting. Percentages are computed over failed outputs.

the edited object.

This explains why a fixed format-only template and ordinary SFT help but do not solve the problem. Visual separation can give the model landmarks in a messy request, and ordinary SFT gives the model more practice writing patches. Neither one directly teaches the central judgment: read the support material, but update the target program. Role-aware alignment targets that judgment more directly. Its rejected outputs are not random bad patches; they are the tempting mistakes that mixed prompts naturally invite, such as copying evidence, editing the support block, or turning a sample answer into the model’s answer.

The results answer the two questions that motivate the paper. First, code-like support material adds a specific source of error beyond the ordinary difficulty of finding an instruction in a long or messy prompt: the model can be drawn toward the wrong span even when that span is only evidence. Second, preference training reduces that mistake without forcing the model to relearn programming from scratch. Role-aware alignment raises code-like success by 14.5 points over the base model and reduces the clean-to-code-like gap from 9.2 to 1.3 points. Relative to ordinary SFT, it gives about a 10% relative improvement on the code-like variant with a smaller drop in general code-generation ability. The practical lesson is therefore not just that this benchmark gets better. It is that code-editing systems should treat “what text should I act on?” as a first step, before asking “what patch should I write?”

6.2 Conclusion

This paper identifies a practical boundary failure in code editing: models can use relevant pasted material but apply it to the wrong text. By holding the requested edit fixed while varying the surrounding code-like material, we separate this failure from ordinary patch-writing errors. The central issue is not whether the model understands the desired behavior, but whether it can distinguish evidence that should guide the edit from the program that

should be changed.

The main finding is that this boundary decision can be trained with lightweight preference signals. Instead of asking the model to relearn programming, the training contrasts a target-code repair with plausible mistakes that copy or modify surrounding material. This pressure makes the model more stable under mixed requests while preserving most of its general code-generation ability.

These results suggest that boundary awareness should be treated as part of code-editing robustness, not merely as a formatting detail. Future work should examine whether this capability remains stable in multi-turn, tool-mediated, and multi-file repair settings, where new evidence accumulates and the intended edit target can become harder to localize.

Limitations

Our failure analysis is behavioral rather than explanatory of model internals. The failure types in Table 3, defined in Appendix B, describe observable output errors, such as editing supporting context or reaching the target region but producing an incorrect patch. The reduction in *Wrong region* and *Misread role span* failures indicates that role-aware alignment reduces boundary-related mistakes, but this analysis does not explain how the model internally distinguishes editable code from supporting context. Our scores characterize the evaluated editing setup, not model capability in isolation (Tan et al., 2026).

The study focuses on single-turn edits with an identifiable target-code region. Multi-file build failures, generated files, or tool-mediated repair loops introduce additional boundaries beyond instruction versus object code. Finally, boundary alignment is designed for role confusion. It does not supply missing library knowledge, redesign algorithms, or guarantee the best patch once the correct location is found.

Uncertainty over alternative patches remains unmeasured; likelihood-free posterior learning targets parameter distributions (Zhang et al., 2025b), so

applying it to structured code outputs requires a separate probabilistic formulation.

References

- Tushar Aggarwal, Swayam Singh, Abhijeet Awasthi, Aditya Kanade, and Nagarajan Natarajan. 2025. [NextCoder: Robust adaptation of code LMs to diverse code edits](#). In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 616–638. PMLR.
- Miltiadis Allamanis. 2019. [The adverse effects of code duplication in machine learning models of code](#). In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019*, pages 143–153, New York, NY, USA. Association for Computing Machinery.
- 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](#). *Preprint*, arXiv:2108.07732.
- Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Jacob Ginesin, Edward Berman, George Chakhnashvili, Anton Lozhkov, Carolyn Jane Anderson, and Arjun Guha. 2024. [Can it edit? evaluating the ability of large language models to follow code editing instructions](#). In *Conference on Language Modeling*.
- 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, and 39 others. 2021. [Evaluating large language models trained on code](#). *Preprint*, arXiv:2107.03374.
- Sizhe Chen, Julien Piet, Chawin Sitawarin, and David Wagner. 2025a. [StruQ: Defending against prompt injection with structured queries](#). In *34th USENIX Security Symposium (USENIX Security 25)*, pages 2383–2400, Seattle, WA. USENIX Association.
- Sizhe Chen, Arman Zharmagambetov, Saeed Mahlouljifar, Kamalika Chaudhuri, David A. Wagner, and Chuan Guo. 2025b. [SecAlign: Defending against prompt injection with preference optimization](#). In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS ’25*, pages 2833–2847, New York, NY, USA. Association for Computing Machinery.
- Wayne Chi, Valerie Chen, Ryan Shar, Aditya Mittal, Jenny Liang, Wei-Lin Chiang, Anastasios Nikolas Angelopoulos, Ion Stoica, Graham Neubig, Ameet Talwalkar, and Chris Donahue. 2025. [EDIT-Bench: Evaluating LLM abilities to perform real-world instructed code edits](#). *Preprint*, arXiv:2511.04486.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024a. [DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence](#). *Preprint*, arXiv:2401.14196.
- Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi Li, Ruibo Liu, Yue Wang, Shuyue Guo, Xingwei Qu, Xiang Yue, Ge Zhang, Wenhui Chen, and Jie Fu. 2024b. [CodeEditorBench: Evaluating code editing capability of large language models](#). *Preprint*, arXiv:2404.03543.
- Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. [The curious case of neural text de-generation](#). In *International Conference on Learning Representations*.
- Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. [LoRA: Low-rank adaptation of large language models](#). In *International Conference on Learning Representations*.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi 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, Shangkuan Qian, and 5 others. 2024. [Qwen2.5-Coder technical report](#). *Preprint*, arXiv:2409.12186.
- Zekun Li, Baolin Peng, Pengcheng He, and Xifeng Yan. 2024. [Evaluating the instruction-following robustness of large language models to prompt injection](#). In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 557–568, Miami, Florida, USA. Association for Computational Linguistics.
- 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*, volume 36.
- Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. [Lost in the middle: How language models use long contexts](#). *Transactions of the Association for Computational Linguistics*, 12:157–173.
- Renze Lou, Kai Zhang, Jian Xie, Yuxuan Sun, Janice Ahn, Hanzi Xu, Yu Su, and Wenpeng Yin. 2024. [MUFFIN: Curating multi-faceted instructions for improving instruction following](#). In *The Twelfth International Conference on Learning Representations*.
- Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 2 others.

2024. [StarCoder 2 and The Stack v2: The next generation](#). *Preprint*, arXiv:2402.19173.
- Juri Opitz. 2024. [A closer look at classification evaluation metrics and a critical reflection of common evaluation practice](#). *Transactions of the Association for Computational Linguistics*, 12:820–836.
- Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. 2023. [Direct preference optimization: Your language model is secretly a reward model](#). *Advances in Neural Information Processing Systems*, 36:53728–53741.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2023. [Code Llama: Open foundation models for code](#). *Preprint*, arXiv:2308.12950.
- Brandon Shannon, Rui Zhang, Lucas Marsh, and Reed M. Johnson. 2025. [Adjuvants to improve efficacy of miticides in managed honey bee \(*Apis mellifera*\) colonies to control *Varroa destructor*](#). *PLOS ONE*, 20(6):e0320037.
- Yuanyuan Tan, Jinming He, Youwen Zhu, and Shijin Wang. 2026. Agent leaderboard scores are not model measurements: Identifiability and a reference-anchor design. *Preprint*.
- Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro von Werra, Arjun Guha, and Lingming Zhang. 2024. [SelfCodeAlign: Self-alignment for code generation](#). In *Advances in Neural Information Processing Systems*, volume 37.
- Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Yewen Pu, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao Yuan, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, Dawei Yin, Xing Hu, and Yunji Chen. 2025. [InverseCoder: Self-improving instruction-tuned code LLMs with inverse-instruct](#). In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25525–25533.
- Jingwei Yi, Yueqi Xie, Bin Zhu, Emre Kiciman, Guangzhong Sun, Xing Xie, and Fangzhao Wu. 2025. [Benchmarking and defending against indirect prompt injection attacks on large language models](#). *Preprint*, arXiv:2312.14197. Accepted by KDD 2025.
- Rui Zhang, Oksana Chkrebtii, and Dongbin Xiu. 2025a. [Dimension-reduced reconstruction map learning for parameter estimation in likelihood-free inference problems](#). *Journal of Machine Learning for Modeling and Computing*, 6(4):29–50.
- Rui Zhang, Oksana A. Chkrebtii, and Dongbin Xiu. 2025b. [Likelihood-free posterior density learning for uncertainty quantification in inference problems](#). *Preprint*, arXiv:2508.00167.
- Yuxi Zhu and Rui Zhang. 2026. [AIM: Amortized inference for multistate transition models](#). *Preprint*, arXiv:2607.23294.
- Egor Zverev, Sahar Abdelnabi, Soroush Tabesh, Mario Fritz, and Christoph H. Lampert. 2025. [Can LLMs separate instructions from data? and what do we even mean by that?](#) In *International Conference on Learning Representations*.

A Implementation and Evaluation Details

A.1 Model set

All experiments use single-turn code-editing prompts rendered through each checkpoint’s recommended chat template. The evaluated checkpoints come from the Code Llama, DeepSeek-Coder, StarCoder2-Instruct, and Qwen2.5-Coder model families (Rozière et al., 2023; Guo et al., 2024a; Lozhkov et al., 2024; Wei et al., 2024; Hui et al., 2024). The model is asked to return a single edited code block for the span it believes should be changed, enclosed in code fences when possible. For scoring, we strip explanatory text, extract the largest code block or raw code response, and align it against the annotated target and support spans in the request. This alignment converts the response into a normalized candidate diff. Thus, references below to a generated or normalized diff refer to this post-processed candidate diff, not necessarily to a raw unified diff emitted by the model. Responses that are closest to a support span or unrelated span are counted as wrong-region edits. No tool calls, JSON mode, multi-turn interaction, or external execution interface is used.

A.2 Data split and training budget

Before splitting, we remove 184 code-polishing groups with non-binary rubric scores, leaving 3,240 binary-scored benchmark-derived edit groups. The remaining groups are split by edit group into 2,040 training groups, 300 development groups, and 900 held-out evaluation groups. All four instruction variants of the same edit group remain in the same split to avoid leakage through near-duplicate code-edit instances, a risk known to inflate evaluation results in machine-learning-on-code settings (Alamanis, 2019). The 600-problem primary slice is drawn from the held-out pool.

We train rank-8 LoRA adapters with alpha 16 and dropout 0.05 on attention projection modules, keeping dense model weights frozen in line with LoRA’s parameter-efficient design of freezing pre-trained weights while adding trainable low-rank

Model	Checkpoint	Size
CodeLlama-Instruct	codellama/CodeLlama-13b-Instruct-hf	13B
DeepSeek-Coder-Instruct	deepseek-ai/deepseek-coder-6.7b-instruct	6.7B
StarCoder2-Instruct	bigcode/starcoder2-15b-instruct-v0.1	15B
Qwen2.5-Coder-Instruct	Qwen/Qwen2.5-Coder-7B-Instruct	7B
Qwen2.5-Coder-Instruct	Qwen/Qwen2.5-Coder-14B-Instruct	14B

Table 4: Open-weight code models used in the evaluation.

Split / slice	Groups	Executable	Patch-check
Training	2,040	1,346	694
Development	300	198	102
Held-out pool	900	594	306
Primary slice	600	396	204

Table 5: Composition of binary-scored edit groups. Executable cases are scored by running benchmark tests; patch-check cases are scored with frozen edit constraints defined before decoding.

update matrices (Hu et al., 2022). For the five-model macro results, we train a separate supervised adapter and a separate role-aware DPO adapter for each evaluated checkpoint; adapters are never shared across model families, architectures, or parameter sizes. Training prompts are capped at 8,192 tokens. The supervised warm-up and ordinary SFT baseline use AdamW with learning rate 2×10^{-4} , effective batch size 64, one epoch, cosine decay, and a 3% warm-up ratio. Role-aware DPO starts from the corresponding supervised adapter and uses $\beta = 0.1$, learning rate 5×10^{-5} , effective batch size 64, one epoch, and the same prompt and response length caps.

A.3 Decoding and patch checks

All reported decoding uses greedy generation with `max_new_tokens = 768`, giving a deterministic maximization-based decoding setting; decoding strategy can materially affect generation behavior, so we keep it fixed across systems (Holtzman et al., 2020).

For each benchmark task, we define the patch-checking rules before generating any model outputs and keep these rules fixed throughout evaluation. Depending on the task, these rules check whether the post-processed candidate diff is localized to the intended target span, inserts tokens that must appear in the repair, removes tokens that should no longer appear, uses the expected API-call form, satisfies specified regex or literal constraints, and matches the predefined edit constraints after superficial formatting differences are normalized. We

Metric	Scoring rule
Edit Success	Binary. Executable benchmark cases must pass tests after the extracted edit is applied. Non-executable cases must satisfy frozen patch checks defined before decoding. Code-polishing cases with non-binary scores are excluded from Table 2.
General Pass@1	HumanEval+ and MBPP+ under greedy decoding. General Pass@1 Δ is the unweighted average of the two Pass@1 deltas relative to the corresponding base checkpoint.

Table 6: Evaluation metrics and scoring rules.

do not revise these checkers after inspecting model outputs. In the primary slice, 396 problems are executable and 204 are scored with frozen patch checks.

A.4 Metrics

Unless otherwise noted, we report model-level macro-averages: for each metric, we first compute the score individually for each model, and then take an unweighted average across all evaluated models. This approach ensures that each model contributes equally to the final reported result, rather than pooling all outputs into a single combined score (Opitz, 2024). General code-generation ability is measured using HumanEval+ and MBPP+, which are extended benchmarks from EvalPlus based on HumanEval and MBPP (Chen et al., 2021; Austin et al., 2021; Liu et al., 2023).

B Failure Classification Protocol

Each failed output is assigned one primary failure label. The labels describe the visible failure in the generated output, not the model’s internal mechanism. To keep the categories mutually exclusive, we apply the labels in the order below. If an output fits more than one category, we use the first applicable label.

We apply the ordered definitions with the OpenAI API model `gpt-4o-2024-08-06` using a fixed

prompt. The classifier receives the original request, the target-code span, support spans, gold patch, generated output, and the ordered label definitions below. It must return exactly one label and a short evidence span. Invalid JSON outputs or labels outside the schema are retried once and otherwise assigned to OTHER.

Wrong region. This label is used when the extracted response is localized to text that should have been left unchanged instead of to the target code. Examples include returning or rewriting an example output, a log message, a test case, a documentation snippet, or an unrelated helper function as if it were the edit target.

Misread role span. This label is used when the model edits the target code, but uses nearby supporting text in the wrong way. For example, the model may hard-code an example output, paste a nearby literal or trace into the program, or keep an old name or value that the instruction says should be replaced or removed.

Right region, wrong patch. This label is used when the model changes the intended code region, but the change is incomplete, incorrect, or does not satisfy the requested edit.

Other. This label is used for failed outputs that cannot be assigned cleanly to the three categories above. This includes malformed or truncated outputs, outputs that omit the edited code, responses that are mostly explanation rather than code, edits that are too broad to localize to one primary error, and cases where multiple unrelated mistakes make the first failure point unclear.