REAL CODE, SYNTHETIC MISTAKES: TEACHING SMALL CODE MODELS NOT TO HALLUCINATE

Anonymous authors
Paper under double-blind review

ABSTRACT

Code autocomplete runs on small open models: every keystroke needs an answer within milliseconds, and much source code is not allowed to leave the company that owns it. These models often *hallucinate*. They call methods that do not exist, pass arguments an API does not accept, or import packages that were never published. The usual remedies do not fit autocomplete: execution-based filtering needs tests that a half-written file does not have, and preference learning needs labelled pairs that nobody collects for individual cursor positions. We start from a simple observation. Every fill-in-the-middle (FIM) example mined from a real repository already contains a correct answer, the code the developer wrote, so a frontier model only has to write a convincing *mistake*. We turn this into an execution-free pipeline that mines 400K realistic FIM holes across eight languages and produces 2.47M hard negatives in four hallucination categories. With this data we study two questions. First, fine-tuning a 7B model on only 5K curated holes raises exact match on the Delulu hallucination benchmark from 42.9% to 59.7% for Qwen2.5-Coder-7B-Instruct and more than halves the rate of completions that fail Delulu's compiler check. Randomly placed holes and cutting outputs to the right length fall short of this, and the gains transfer to three model families and to languages never seen in training. Second, the synthetic negatives add value when chosen well. Contrastive training on them improves control-flow completion by 1.5–3.5 points over fine-tuning alone, at some cost on long multi-line completions. A cheap, label-free difficulty signal, the fraction of blind LLM judges a negative fools, finds the pairs that help: in a controlled study, ORPO trained on the pairs that fool the judges beats fine-tuning on Delulu by 1.5–1.9 points at the same budget, while rule-based negatives add little. We release the code for every stage of the pipeline after hole mining.

1 INTRODUCTION

Most requests to a code assistant are autocomplete requests. When a developer pauses inside a file, the editor sends the code before and after the cursor to a model and asks for the missing piece, a task known as fill-in-the-middle (FIM) (Bavarian et al., 2022; Fried et al., 2022). A suggestion is only useful if it appears within a few hundred milliseconds (Ziegler et al., 2022; Mozannar et al., 2024), and many companies do not allow their code to leave their own network. Autocomplete is therefore served by small open models in the 3B–7B range (Guo et al., 2024; Hui et al., 2024; Lozhkov et al., 2024), not by frontier models.

These small models often *hallucinate*. They produce completions that fit the surrounding syntax but refer to things that do not exist: a method the library never defined, an argument the API does not accept, a variable that was never declared, or a package that was never published. Figure 1 shows a typical case. The errors are costly precisely because they look right; a developer who accepts one may only notice it when the build breaks. On the Delulu benchmark (Erfanian et al., 2026), which collects such hallucination-prone holes, an untuned Qwen2.5-Coder-7B reproduces the developer's code only 42.9% of the time.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

```
fmt.Printf("Enter new password for %s: ", username)
password, err := terminal.ReadPassword(int(os.Stdin.Fd()))
if err != nil { return "", err }
fmt.Printf("Confirm new password for %s: ", username)
confirmPassword, err := terminal. ⟨cursor⟩
```

Base 7B:
`CapturePassword(. . . )`

After fine-tuning:
`ReadPassword(. . . )`

Figure 1: A FIM hallucination in Go. The correct call appears three lines above the cursor, yet the base model invents `CapturePassword`, which reads as idiomatic Go but does not exist in `golang.org/x/term`. No unit test covers this mid-file position, so execution-based filtering cannot catch it.

Two standard remedies do not transfer well to this setting. Execution-based filtering keeps only generated code that passes unit tests (Chen et al., 2021; Shojaee et al., 2023; Tian et al., 2024) and underlies strong synthetic-data pipelines for code (Wei et al., 2023; Luo et al., 2023). A cursor in the middle of a file, however, rarely has a test that exercises it, the file often does not parse while the developer types, and every language needs its own sandbox. Preference learning (Rafailov et al., 2023; Hong et al., 2024; Lee et al., 2023) avoids execution but needs pairs of good and bad completions, and nobody labels such pairs for individual cursor positions at training scale.

Our starting point is a simple observation: a FIM example mined from a real repository already comes with a correct answer, namely the code the developer actually wrote. We never need to generate or verify a correct completion. What is missing is the other half of a training pair, and frontier models turn out to be very good at writing convincing mistakes. We therefore ask a frontier model to rewrite each developer completion into a plausible but wrong one of a requested kind, borrowing the idea of hard negatives from metric learning and retrieval (Schroff et al., 2015; Robinson et al., 2020; Xiong et al., 2020). This inverts the usual synthetic-data recipe of generating a correct answer and then checking it, and it removes the need for tests, sandboxes, or human labels. It also gives a clean experimental handle: from the same holes we can build one training set with only the developer code and another that adds the synthetic mistake, and compare the two directly. This lets us ask two questions:

Q1. Can fine-tuning on mined FIM holes make a small model hallucinate less?
Q2. Do frontier-written hard negatives add value beyond the developer code, and can we tell in advance which ones will?

To answer them, we mine 399,556 realistic holes at third-party API calls in eight languages and generate 2.47M hard negatives in four categories (Section 3). We then fine-tune Qwen2.5-Coder models under matched budgets, using supervised fine-tuning (SFT) on the developer code and two contrastive objectives, ORPO (Hong et al., 2024) and anchored DPO (Rafailov et al., 2023), on the pairs (Section 4).

Q1: yes, and with little data. Five thousand holes lift Delulu exact match of Qwen2.5-Coder-7B-Instruct from 42.9% to 59.7%, and more than halve the share of completions that fail Delulu's compiler check. This is not a side effect of learning to stop: giving the base model the reference length reaches only 48.8%. Nor is it generic fine-tuning: random holes reach only 54.4%, while holes placed where a call is being typed reach 59.2%, and 59.7% when they target third-party APIs. The gains carry over to StarCoder2, CodeLlama, and the 3B model, and to C++ and TypeScript, which never appear in training.

Q2: yes, when the pairs are chosen well. Training on the pairs improves control-flow completion by up to 3.5 points over SFT on the same holes. How convincing the mistake is matters: rule-based corruptions help least, while the most convincing model-written negatives help most. To find the useful pairs we introduce the *fool rate*, the fraction of a panel of small LLM judges that accept a negative as correct without being told it may be wrong. ORPO trained on the pairs that fool most judges beats SFT by 1.9 points at a fixed budget, while easy pairs give nothing. Selecting the hardest pair within each category brings this gain to our main setting. Developers agree with the judges: they accept a negative that fooled every judge 4.6 times as often as one that fooled none.

In summary, we contribute:

- an execution-free pipeline that turns open-source repositories into paired FIM data: realistic holes with developer-written answers and category-specific hard negatives (2.47M pairs, eight languages, 99.5% well-formed, 92.3% of the requested category);
- a controlled study showing that fine-tuning on a few thousand well-chosen holes substantially reduces FIM hallucinations in small models and transfers across benchmarks, languages, and model families;
- evidence that synthetic hard negatives improve contrastive training, especially on control flow, and that a cheap, label-free difficulty signal identifies the pairs worth training on.

2 Related Work

Code models and fill-in-the-middle. Code models such as Codex (Chen et al., 2021), StarCoder2 (Lozhkov et al., 2024), DeepSeek-Coder (Guo et al., 2024), and Qwen2.5-Coder (Hui et al., 2024) learn infilling by reordering training documents into prefix, suffix, and middle (Bavarian et al., 2022). Infilling quality is usually measured on HumanEval-Infilling (Fried et al., 2022), SAFIM (Gong et al., 2024), and, more recently, real repository edits in Real-FIM-Eval (Gong et al., 2025). We use the same format and benchmarks, and target a specific failure mode within them.

Code hallucination. CodeHalu (Tian et al., 2024) detects hallucinations in generated programs by executing them. Liu et al. (2026) annotate hallucinated spans in model-written functions under an eleven-category taxonomy. Delulu (Erfanian et al., 2026) provides a multilingual FIM benchmark of hallucination-prone holes, each verified in a container, covering the four categories we train on. These works measure hallucination; we study how to reduce it with training data that needs no execution. We use Liu et al. (2026) as an out-of-taxonomy test.

Synthetic data for code. Self-Instruct (Wang et al., 2023), Magicoder (Wei et al., 2023), and WizardCoder (Luo et al., 2023) build instruction data from model outputs, and execution feedback is the standard way to keep only correct samples (Zhang et al., 2023; Shojaee et al., 2023). We generate the *incorrect* side instead, which lets the developer's own code serve as the verified answer.

Hard negatives and preference learning. Contrastive learning works best with negatives that are close to the positive but wrong (Schroff et al., 2015; Robinson et al., 2020; Xiong et al., 2020). DPO (Rafailov et al., 2023) and ORPO (Hong et al., 2024) learn from chosen and rejected responses, and RLAIF (Lee et al., 2023) obtains the preference labels from a model. Our pairs are built rather than ranked: the chosen side is real code and the rejected side is a controlled, category-specific edit of it.

LLM judges. LLMs are widely used as evaluators (Zheng et al., 2023; Gu et al., 2024), and panels of diverse judges reduce single-model bias (Verga et al., 2024). We use a panel only to estimate how convincing a known mistake is, and use that estimate to select training pairs.

3 Method

Our pipeline turns open-source repositories into training pairs without running any code (Figure 2). It has three stages: find holes that look like real autocomplete requests, ask a frontier model to write a convincing mistake for each hole, and package the results so that different training objectives see exactly the same examples. An optional fourth stage scores how convincing each mistake is.

3.1 Mining realistic FIM holes

A useful training hole should look like a moment in real editing and should keep the developer's answer untouched. We mine permissively licensed repositories in eight languages and focus on calls to popular third-party libraries, because that is where a model has to remember an API it cannot see in the file, and therefore where it tends to invent one. Tree-sitter lists every function and method call in a file, and a small model (GPT-4.1-mini) marks which calls belong to a list of target libraries, or abstains. The cursor is then placed by fixed syntax rules, right after the member-access token (`.`, `->`, `::`, `?->`) or after `new`, which is where an editor would fire a request while the developer types the call. The text before and after the cursor becomes the prefix p and suffix s, and the rest of the call is the answer g. We keep a hole only if p, g, and s concatenate back to the original file byte for byte. The

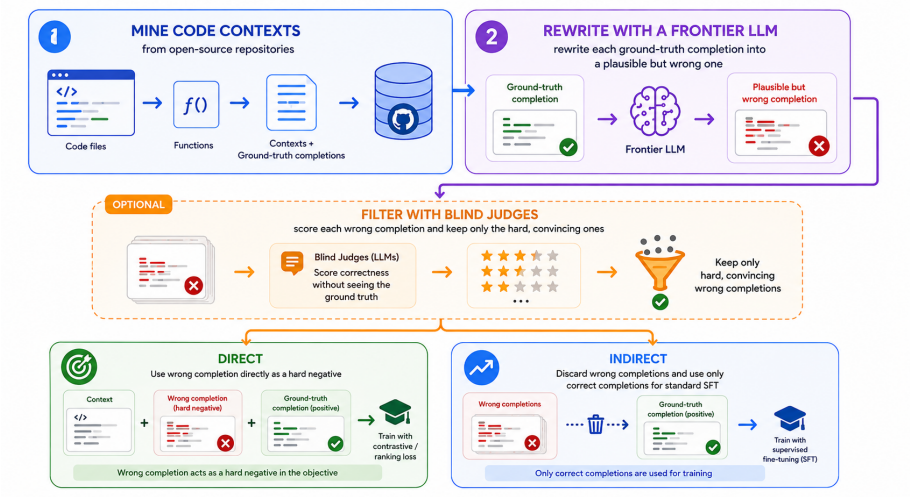


Figure 2: Pipeline overview. (1) We mine FIM holes whose answer is the developer's own code. (2) A frontier model rewrites that answer into a plausible but wrong completion. (3) Optionally, blind LLM judges estimate how convincing each mistake is. The same holes then feed supervised fine-tuning on the correct code alone (right) or contrastive training on correct–wrong pairs (left).

Table 1: The four hallucination categories used to write hard negatives, with a real example of each from our data.

Category	**Edit to the developer code**	**Example** ($g \rightarrow h$)
METHOD	invent a plausible method	`createTextNode` → `createPlainTextNode`
PARAMETER	add an argument the API lacks	`delete()` → `delete(force: true)`
UNDEF. VARIABLE	use a name not in scope	`attach(csv)` → `attach(uploaded_csv)`
IMPORT	import a non-existent package	`org.junit.jupiter.api` → `org.junit5.api`

result is 399,556 holes with developer-written answers, disjoint from every evaluation benchmark (Appendix A).

3.2 Writing plausible mistakes

For each hole, a frontier model (GPT-5.2-Codex, GPT-5.4, or GPT-5.5) receives p, g, and s and one of four hallucination categories taken from Delulu (Table 1). It returns a completion h that differs from g by exactly that kind of error and still fits the context. The prompts forbid obvious tells such as names containing "fake" or "invalid", since a model can learn to reject those without learning anything about APIs (Appendix C).

Because we ask for mistakes on purpose, we check that they are the mistakes we asked for. A structural check discards outputs that break the expected format or, for imports, fail to rebuild the file; it keeps 2,473,312 of 2,484,731 candidates (99.5%). A separate audit of 36K balanced samples confirms that 92.3% realise the requested category, and manual inspection shows that most flagged cases are detector misses rather than wrong edits (Appendix B).

3.3 Aligned training sets and objectives

We balance the pool across languages and categories, cap the rows taken from any one repository, and freeze an ordered list of hole IDs for each budget. From that list we write two files: one with only (p, s, g), and one that also includes h. Every objective therefore sees the same prompts and the same developer answers in the same order; the only difference is whether it also sees the mistake.

Let $x = \mathrm{FIM}(p, s)$ be the model-specific FIM prompt, $y^+ = g$, and $y^- = h$. Supervised fine-tuning (SFT) uses only the developer code, $\mathcal{L}_{\mathrm{SFT}} = -\log \pi_\theta(y^+ \mid x)$. The contrastive objectives also push

probability away from y^-. ORPO (Hong et al., 2024) adds an odds-ratio term to the SFT loss,

$$\mathcal{L}_{\text{ORPO}} = \mathcal{L}_{\text{SFT}} - \beta \log \sigma\left(\log \frac{\text{odds}_\theta(y^+ \mid x)}{\text{odds}_\theta(y^- \mid x)}\right), \qquad \text{odds}_\theta(y \mid x) = \frac{\pi_\theta(y \mid x)}{1 - \pi_\theta(y \mid x)}. \tag{1}$$

DPO (Rafailov et al., 2023) compares both completions to a frozen reference model π_{ref}. On FIM data, plain DPO can widen the gap by lowering the probability of both completions, which makes the model ramble past the hole, so we add the SFT loss as an anchor:

$$\mathcal{L}_{\text{DPO+NLL}} = -\log \sigma\left(\beta \log \frac{\pi_\theta(y^+ \mid x)}{\pi_{\text{ref}}(y^+ \mid x)} - \beta \log \frac{\pi_\theta(y^- \mid x)}{\pi_{\text{ref}}(y^- \mid x)}\right) + \lambda\, \mathcal{L}_{\text{SFT}}. \tag{2}$$

All losses are computed on the middle tokens only; prompt tokens are masked. Following the reference implementations, ORPO and the NLL anchor use the mean token log-probability of a completion, and the DPO margin uses the sum.

3.4 Fool rate: how convincing is a mistake?

Not every synthetic mistake teaches something. A negative that any reader rejects at a glance adds little contrast. We estimate how convincing a negative is with a panel $\mathcal{J}$ of three small LLM judges (GPT-4o-mini, GPT-4.1-mini, GPT-5.4-mini). Each judge sees only p, h, and s; it is not told that h may be wrong, and it never sees g. The fool rate is the fraction of judges that accept the negative,

$$\tau(h) = \frac{1}{|\mathcal{J}|} \sum_{j \in \mathcal{J}} \mathbb{1}[j(p, h, s) = \text{correct}]\,. \tag{3}$$

The fool rate is computed once, offline, and is used only to choose training pairs; it never weights the loss. It needs only small models and no human labels.

4 Experiments

4.1 Setup

Training. Our main student is Qwen2.5-Coder-7B-Instruct, a popular open model for local autocomplete. We also train the 3B sibling, StarCoder2-7B, and CodeLlama-7B. All runs are full fine-tuning with the same learning rate (5×10^{-6}, cosine), global batch (256), and context length (8K tokens). We compare SFT, ORPO, and anchored DPO on nested 5K and 20K subsets of the curated set. Small budgets need care: one epoch over 5K holes is only about 20 optimizer updates, and such undertrained models ramble far past the hole. We therefore train for 7 epochs at 5K and 2 at 20K, so that every run receives 137–156 updates. Unless stated otherwise, the SFT and contrastive runs in a comparison share data, order, and updates (Appendix D). We repeat the 5K runs with three training seeds; the seed standard deviation is at most 0.2 points on Delulu and 1.1 on SAFIM control flow (Appendix F).

Benchmarks and metrics. Delulu (Erfanian et al., 2026) is our main target: 1,950 hallucination-prone holes in seven languages, two of which (C++ and TypeScript) never appear in our training data. To check that general infilling does not suffer, we use the three HumanEval-Infilling splits (Fried et al., 2022), the four SAFIM subsets (Gong et al., 2024), and Real-FIM-Eval (Gong et al., 2025). We report exact match (EM), and in the appendix also edit similarity and CodeBLEU (Ren et al., 2020). Exact match can penalise a valid alternative, so we also measure hallucination directly. Every Delulu item ships with a container that compiles or type-checks a completion in its pinned environment; the *verified hallucination rate* is the share of completions this verifier rejects (Appendix D.1). All decoding is greedy.

4.2 Q1: Does fine-tuning reduce hallucinations?

A few thousand holes are enough. Table 2 shows that every objective lifts Delulu from 42.9% to 58–60% with only 5K training holes, and more data adds little: 20K and 35K reach about 60%, and 100K reaches 61.7% (Appendix E). The gains are not limited to hallucination-prone holes. Part of the improvement on general benchmarks is learning where to stop: the base model often keeps writing after the hole is filled, and truncating its output to the reference length raises its EM by up to 24.2

Table 2: Exact match (%) of Qwen2.5-Coder-7B before and after training on the same aligned holes. SFT uses only the developer code; ORPO and DPO+NLL also use the synthetic negative. Bold marks the best result within each budget.

		5K holes			20K holes		
Benchmark	**Base**	**SFT**	**ORPO**	**DPO+NLL**	**SFT**	**ORPO**	**DPO+NLL**
Delulu (hallucination)	42.9	**59.7**	59.6	58.0	60.0	**60.1**	59.2
HumanEval single-line	40.2	**56.7**	54.3	54.3	**57.3**	54.9	**57.3**
HumanEval multi-line	16.1	**30.5**	24.6	28.0	**28.8**	23.7	26.3
HumanEval random-span	34.6	**47.2**	43.3	45.7	**48.8**	45.7	48.0
SAFIM API	53.9	**70.7**	68.7	68.4	71.0	**71.3**	69.0
SAFIM block	12.4	**27.9**	27.1	27.4	**27.9**	26.1	26.9
SAFIM block-v2	13.4	**34.6**	33.7	34.1	**35.2**	32.8	34.1
SAFIM control flow	4.8	28.8	31.6	**32.3**	31.4	32.9	**33.2**

Table 3: What makes the data work (Qwen2.5-Coder-7B, 5K holes, same files and updates). Random holes place the cursor at uniform positions; all-call-site holes include in-file and standard-library calls. Hall. = verified hallucination rate on Delulu (lower is better); HE = mean of the three HumanEval-Infilling splits. Full results in Appendix F.

Training data	**Delulu↑**	**Hall.↓**	**SAFIM API↑**	**SAFIM ctrl↑**	**HE↑**
None (base model)	42.9	29.3	53.9	4.8	30.3
+ cut to reference length	48.8	24.6	58.7	7.7	34.3
SFT, random holes	54.4	19.2	64.5	**33.8**	41.4
SFT, all call sites	59.2	13.5	69.5	30.4	44.0
SFT, our API-call holes	**59.7**	**12.7**	**70.7**	28.8	**44.8**
ORPO, our holes + negatives	59.6	12.9	68.7	31.6	40.7

points, while the same truncation changes trained models by at most 1.6. Even against this truncated base, SFT improves five of the seven HumanEval and SAFIM splits, by up to $+21.1$ points on SAFIM control flow; on the two SAFIM block splits the gain is entirely due to stopping (Appendix D.1).

The gain is not just learning to stop. Could the trained model simply end its completions in the right place? Cutting the base model's output to the length of the reference lifts Delulu only to 48.8%, 10.9 points below SFT, and lowers its hallucination rate only to 24.6% (Table 3). SFT, in contrast, cuts the verified hallucination rate from 29.3% to 12.7%, so the model writes different code, not just shorter code. SFT helps every category and imports most, and most remaining failures pass the wrong type or number of arguments to a real API rather than invent a name (Appendix F).

Where the holes come from matters. Fine-tuning on arbitrary FIM data is not enough. Random holes from the same files help general infilling, and most of all control flow, whose targets they resemble, but reach only 54.4% on Delulu, 5.3 points below our holes. Holes at every call site, including in-file and standard-library calls, close most of the gap (59.2%): placing the cursor where a call is being typed is what matters most, and focusing on third-party APIs adds a further half point. The verified hallucination rate follows the same order, and ORPO keeps it at the same level (12.9%). The model is not just memorising the APIs it saw: 29% of Delulu holes call a method whose name also appears as a training answer, and the gain is $+21.6$ points on those and still $+14.8$ on the rest (Appendix A).

The effect transfers. The same recipe works across model families and languages (Figure 3). On the 100K set, Delulu improves by 18.8 points for Qwen2.5-Coder, 15.4 for StarCoder2, and 35.1 for CodeLlama; 22 of the 24 model–benchmark cells improve, and StarCoder2, already a strong Python infiller, loses points on two HumanEval splits. Qwen2.5-Coder-3B trained on the same 100K holes reaches 58.7% on Delulu, above both 7B starting points. C++ and TypeScript have no training rows, yet C++ rises from 27.2% to 67.2% (125 holes, paired 95% CI of the gain $[31.2, 48.8]$) and TypeScript from 32.1% to 43.3% (420 holes, $[8.1, 14.5]$).

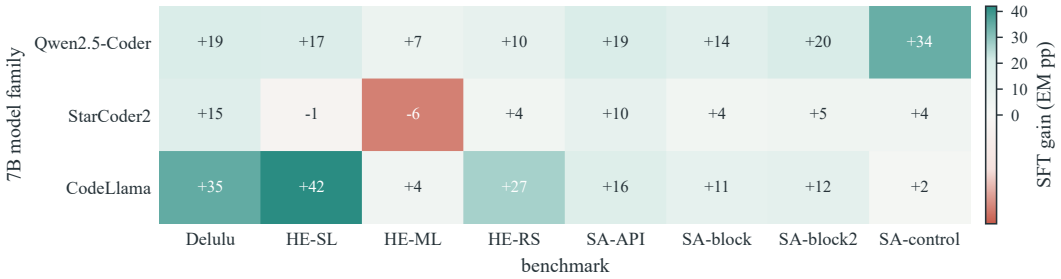


Figure 3: EM gain from SFT on the same 100K holes for three 7B model families (raw numbers in Appendix G). Delulu improves by 15 to 35 points and 22 of 24 cells improve; StarCoder2, already strong on short Python infilling, loses a few points on two HumanEval splits.

Table 4: Who writes the negative (ORPO, same 5K holes and updates). Rule-based negatives rename the callee, add a random argument, or swap in an out-of-scope name or a fake module; model samples are wrong completions sampled from the base or the 5K SFT model, preferring ones that use a name that does not exist (Appendix F). $\bar{\tau}$ is the mean fool rate of the negatives each writer produced. The last row continues the 5K SFT model with ORPO for the same number of updates.

Negatives	$\bar{\tau}$	**Delulu↑**	**Hall.↓**	**SAFIM ctrl↑**
None (SFT)	–	59.7	**12.7**	28.8
Rule-based edits	0.09	59.6	13.4	30.6
Base-model samples	0.63	**59.9**	13.0	**32.0**
SFT-model samples	0.43	59.6	13.0	31.8
Frontier model (ours)	0.22	59.6	12.9	31.6
Frontier, SFT→ORPO	0.22	59.5	14.6	31.5

4.3 Q2: Do synthetic hard negatives add value?

Contrast helps where the boundary is tight. The two objectives that use the negatives match SFT on Delulu: ORPO is within 0.15 points at both budgets (95% paired intervals include zero), since Delulu rewards reproducing the developer code, which SFT already learns. The picture changes on control flow, where a completion must close exactly the right condition or loop. On SAFIM control flow, ORPO and DPO+NLL beat SFT by 2.8 and 3.5 points at 5K and by 1.5 and 1.8 at 20K, with matching gains in edit similarity and CodeBLEU (Table 2). Every one of these gains is positive, and the 5K ORPO gain holds across seeds ($+2.5 \pm 0.4$). Why would API negatives help control flow? Most negatives differ from the developer code in one name or argument near the end of the span, so the contrastive term focuses on the tokens where a completion must end. Consistent with this, in 79% of the control-flow items that ORPO fixes over SFT, SFT had the right start but ended in the wrong place, almost always by running past the hole (75%).

On long multi-line targets SFT remains stronger, so the two are complementary: SFT on developer code is a strong default, and the negatives add a sharper sense of what must *not* be written. The result does not depend on a lucky setting: ORPO varies by at most 0.3 points across $\beta \in [0.05, 1]$, and DPO+NLL improves steadily as the anchor weight λ grows (Appendix E). Without the anchor, DPO collapses to 29.4% on Delulu at 100K, which is why we anchor it.

Convincing negatives help most. Simple rule-based corruptions cannot replace model-written mistakes. Table 4 trains ORPO on four kinds of negatives for the same holes. Rule-based edits are easy to spot (mean fool rate 0.09; 82% fool no judge) and give the smallest gain: none on Delulu and 1.8 points on control flow. Model-written negatives are more convincing, and their control-flow gain grows with how often they fool the judges: frontier negatives add 2.8 points over SFT (paired 95% CI $[2.3, 3.3]$) while keeping SFT's verified hallucination rate, and mistakes sampled from the base model, which fool the judges most often (mean fool rate 0.63), add 3.2. On Delulu every kind of negative stays within half a point of SFT. Continuing the SFT model with ORPO keeps its Delulu score and reaches 31.5 on control flow; SFT continued for the same number of updates reaches a similar level (Appendix F), so ORPO is most useful when it replaces, rather than follows, SFT on the

Table 5: Selecting pairs by fool rate (C#, 5K pairs per stratum, Delulu EM %). Each stratum has the same size and category mix, and every run receives the same number of updates. SFT never sees the negatives, so it acts as a control for the quality of the holes. Paired 95% intervals are given in the text.

Objective	Easy ($\tau = 0$)	Medium ($\tau = 1/3$)	Hard ($\tau \geq 2/3$)
SFT	54.6	54.7	54.5
ORPO (β=1)	54.6	56.3	**56.4**
ORPO − SFT	+0.0	+1.5	**+1.9**

Table 6: Hard pairs (Qwen2.5-Coder-7B-Instruct, 5K holes, matched updates, EM % except Hall.). "Hard pairs" keeps, for each hole, the most convincing of its candidate negatives.

Model	**Delulu↑**	**Hall.↓**	**HE multi↑**	**SAFIM block↑**	**SAFIM ctrl↑**
Base	42.9	29.3	16.1	12.4	4.8
SFT	59.7	12.7	**30.5**	**27.9**	28.8
ORPO, all pairs	59.6	12.9	24.6	27.1	31.6
ORPO, hard pairs	**60.4**	**12.5**	25.3	27.6	**32.4**
SFT→ORPO, hard pairs	59.9	14.0	25.1	27.4	31.9

same budget. The ordering follows the fool rate, which suggests that how convincing a negative is, rather than how it was produced, is what matters. We test this directly next.

Fool rate finds the useful pairs. If negatives differ in how much they teach, the convincing ones should help most. We split pairs into easy ($\tau = 0$, no judge fooled), medium ($\tau = 1/3$), and hard ($\tau \geq 2/3$, most judges fooled) strata of equal size and composition (Table 5). SFT is flat across strata, which shows that the strata do not differ in the quality of their holes. ORPO, by contrast, gains nothing from easy negatives and beats SFT by 1.5 points on medium ones (paired 95% CI $[0.6, 2.5]$) and 1.9 points on hard ones ($[0.9, 3.0]$) in the C# study, and the hard-stratum gain holds across training seeds ($+2.4 \pm 0.7$). The strata do not merely reflect the taste of GPT-family judges: Claude Haiku 4.5 and a code-specialised model from a third provider, judging the same pairs, rank them consistently with the GPT panel (Spearman $\rho = 0.50$), and they are fooled seven times as often by hard pairs as by easy ones (Appendix H). After matching the strata on edit distance and completion length, ORPO still beats SFT by 1.4 points on hard pairs, but easy pairs now gain nearly as much ($+1.3$), so part of the stratum effect reflects how large the edit is. The hard-pair gain also carries beyond Delulu: on SAFIM control flow, hard-pair ORPO beats SFT by 2.2 points and easy-pair ORPO by 0.4.

Hard pairs pay off in the main setting too. Why does the mixed 5K set show no ORPO gain on Delulu when hard strata do? Most of its negatives are easy: 60% fool no judge and only 19% fool two or more (Appendix H), so the mixed set behaves much like the easy stratum, where ORPO gains nothing. Choosing, for each hole, the most convincing of its candidate negatives raises the mean fool rate from 0.22 to 0.29 without changing the holes (Table 6): hard-pair ORPO beats SFT by 0.7 points on Delulu and 3.6 on control flow, and lowers verified hallucinations to 12.5%. Continuing the SFT model with ORPO instead keeps most of the control-flow gain. The fool rate is therefore a cheap, label-free way to decide which negatives are worth training on.

Fool rate tracks human judgment. Three professional developers reviewed 400 negatives, balanced across the four fool-rate levels, and marked each completion as correct or not (Appendix H). They accepted 41% of the negatives that fooled all three judges but only 9% of those that fooled none, and developer acceptance correlates with fool rate (Spearman $\rho = 0.38$, $p < 10^{-6}$). A high fool rate does not mean the negative is secretly correct: for imports, where a registry lookup can check it, only 4.6% of the fabricated packages in hard pairs exist at all, fewer than in easy pairs (9.5%; Appendix B). The judges thus approximate what a busy developer would let slip.

Among frontier teachers, the choice matters little. To check that the negatives do not just teach the quirks of GPT-style mistakes, we regenerate the negatives for 4,861 shared holes with Claude Sonnet 5 and Gemini 2.5 Pro and retrain ORPO and DPO+NLL. On the four benchmarks with more than a

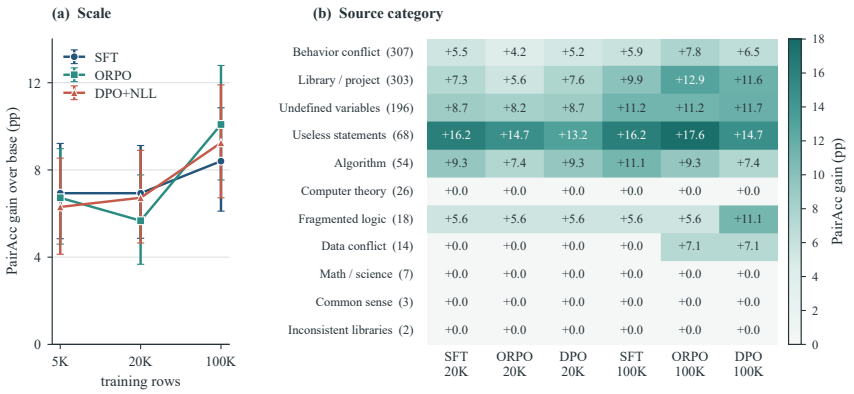


Figure 4: Transfer to human-annotated CoderEval hallucinations outside our four categories. (a) Gain in pairwise accuracy over the base model with 95% intervals clustered by task. (b) Gain by annotated category; numbers in parentheses are pair counts. Details in Appendix J.

thousand examples, the spread between teachers is at most 0.6 points, and every teacher yields the same Delulu and control-flow gains (Appendix I).

Beyond the four categories. Finally, we test hallucinations our training never targeted, using the human-annotated CoderEval errors of Liu et al. (2026) (952 pairs, 11 categories). We measure whether the model assigns higher likelihood to the correct function than to the annotated hallucinated one. Fine-tuning raises this pairwise accuracy by up to 6.9 points at 20K and 10.1 points at 100K over the base model, with the largest gains on library, undefined-variable, and wrong-behaviour errors, which make up most of the pairs (Figure 4).

5 Discussion and Conclusion

Small code models hallucinate APIs because they must recall, from a short window, names they cannot see. We showed that the data needed to correct this is already sitting in open-source repositories. Holes placed at real third-party API calls come with a verified answer, the developer's own code, and a frontier model can supply the matching mistake. Fine-tuning a 7B model on only 5K such holes lifts Delulu exact match by 17 points, cuts compiler-verified hallucinations by more than half, teaches the model where a completion should stop, and transfers to other benchmarks, languages, and model families. The synthetic mistakes add value on top of this: contrastive training on them improves control-flow completion, and the fool rate, a label-free difficulty signal from small blind judges, identifies the pairs that make contrastive training beat fine-tuning alone.

Practical guidance. For a team training an autocomplete model, our results suggest a simple recipe: mine API-call holes and fine-tune on the developer code. Where control flow matters, train ORPO instead, on the same budget and with the highest-fool-rate pair per category: it improves control flow and keeps the hallucination rate, at some cost in long multi-line completions. Five thousand holes and a few hundred optimizer updates already capture most of the gain.

Limitations. Our negatives cover four identifier-level categories. Pairwise tests on human-annotated CoderEval errors show transfer to other categories, but logic errors such as off-by-one mistakes need their own prompts. On arbitrary real edits (Real-FIM-Eval), exact match stays low for every model, so we see our method as a targeted fix for API hallucinations rather than a general improvement in code synthesis. Our main experiments use 7B models and budgets up to 100K holes; how the benefit of hard negatives scales with larger students is an open question (Appendix K).

AI USE STATEMENT

We used generative AI models as part of the research method: frontier models write the synthetic hard negatives, a small model classifies third-party API calls during hole mining, and small models act as blind judges for the fool rate (Section 3). These uses are described in full, with prompts, in the appendix. We also used AI assistants to draft and revise parts of the text of this paper, to run mock reviews of drafts, and to write parts of the data-processing and plotting code. All AI-assisted text was rewritten or checked by the authors, all code was reviewed and tested by the authors, and every number in the paper was produced by our own training and evaluation runs. We take full responsibility for the content of this work.

ETHICS STATEMENT

All training code comes from public, permissively licensed repositories, and we remove every file and repository that appears in an evaluation benchmark. The negatives we generate are deliberately wrong code; they are used only as the rejected side of training pairs and are never presented as correct. The human study involved professional developers who reviewed code completions as part of their work, received no personal questions, and whose responses were stored without identifiers. We are not aware of new dual-use risks beyond those of code models in general; reducing hallucinated API calls should make accepted suggestions safer.

REPRODUCIBILITY STATEMENT

Section 3 and Appendices A–C describe hole mining, negative generation, validation, and the fool-rate judges, including every prompt. Appendix D lists all training and evaluation settings, including the matched update counts and the preference-objective sweeps. We release the code for every pipeline stage after hole extraction (negative generation, validation, fool-rate judging, curation, FIM formatting, and training recipes); it can be applied to any set of FIM holes. The curated training data is not released. All benchmarks we evaluate on are public, and the evaluation scripts, including the Delulu verification harness, are part of the release.

REFERENCES

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. 2022. doi: 10.48550/arXiv.2207.14255.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. 2021.

Mahdi Erfanian, Nelson Troncoso, Aashna Garg, Amabel Gale, Xiaoyu Liu, Pareesa Ameneh Golnari, and Shengyu Fu. Delulu: A verified multi-lingual benchmark for code hallucination detection in fill-in-the-middle tasks. 2026. doi: 10.48550/arXiv.2605.07024. URL https://arxiv.org/abs/2605.07024.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida I. Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and M. Lewis. Incoder: A generative model for code infilling and synthesis. 2022. doi: 10.48550/arXiv.2204.05999.

Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. Evaluation of LLMs on syntax-aware code fill-in-the-middle tasks. 2024. doi: 10.48550/arXiv.2403.04814.

Linyuan Gong, Alvin Cheung, Mostafa Elhoushi, and Sida Wang. Structure-aware fill-in-the-middle pretraining for code. 2025. doi: 10.48550/arXiv.2506.00204.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, et al. A survey on LLM-as-a-judge. volume 7, pp. 101253. Elsevier BV, 2024. doi: 10.1016/j.xinn.2025.101253.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, Y. K. Li, et al. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. 2024. doi: 10.48550/arXiv.2401.14196.

Jiwoo Hong, Noah Lee, and James Thorne. ORPO: Monolithic preference optimization without reference model. pp. 11170–11189. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.626.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, K. Dang, et al. Qwen2.5-coder technical report. 2024.

Harrison Lee, Samrat Phatale, H. Mansoor, Kellie Lu, Thomas Mesnard, Colton Bishop, Victor Carbune, and Abhinav Rastogi. RLAIF vs. RLHF: Scaling reinforcement learning from AI feedback with human feedback. 2023.

Fang Liu, Yang Liu, Lin Shi, Zhen Yang, Li Zhang, Xiaoli Lian, Zhongqi Li, and Yuchi Ma. Beyond functional correctness: Exploring hallucinations in LLM-generated code. *IEEE Transactions on Software Engineering*, 52(3):1037–1055, 2026. doi: 10.1109/TSE.2026.3657432. URL https://github.com/Lorien1128/code_hallucination.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, J. Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. 2024. doi: 10.48550/arXiv.2402.19173.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. WizardCoder: Empowering code large language models with Evol-Instruct. 2023.

Hussein Mozannar, Valerie Chen, Mohammed Alsobay, Subhro Das, Sebastian Zhao, Dennis Wei, Manish Nagireddy, P. Sattigeri, Ameet Talwalkar, and David A. Sontag. The RealHumanEval: Evaluating large language models' abilities to support programmers. In *Trans. Mach. Learn. Res.*, 2024. doi: 10.48550/arXiv.2404.02806.

Rafael Rafailov, Archit Sharma, E. Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. pp. 53728–53741. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2023. doi: 10.52202/075280-2338.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: A method for automatic evaluation of code synthesis. *arXiv.org*, 2020.

Joshua Robinson, Ching-Yao Chuang, Suvrit Sra, and Stefanie Jegelka. Contrastive learning with hard negative samples. In *International Conference on Learning Representations*, 2020.

Florian Schroff, Dmitry Kalenichenko, and James Philbin. FaceNet: A unified embedding for face recognition and clustering. In *Computer Vision and Pattern Recognition*, pp. 815–823. IEEE, 2015. doi: 10.1109/CVPR.2015.7298682.

P. Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based code generation using deep reinforcement learning. 2023. doi: 10.48550/arXiv.2301.13816.

Yuchen Tian, Weixiang Yan, Qian Yang, Xuandong Zhao, Qian Chen, Wen Wang, Ziyang Luo, and Lei Ma. Codehalu: Investigating code hallucinations in llms via execution-based verification. In *AAAI Conference on Artificial Intelligence*, volume 39, pp. 25300–25308. Association for the Advancement of Artificial Intelligence (AAAI), 2024. doi: 10.1609/aaai.v39i24.34717. URL https://api.semanticscholar.org/CorpusID:269484644.

Pat Verga, Sebastian Hofstätter, Sophia Althammer, Yixuan Su, Aleksandra Piktus, Arkady Arkhangorodsky, Minjie Xu, Naomi White, and Patrick Lewis. Replacing judges with juries: Evaluating LLM generations with a panel of diverse models. 2024. doi: 10.48550/arXiv.2404.18796.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. pp. 13484–13508. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.acl-long.754.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with OSS-Instruct. 2023.

Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul Bennett, Junaid Ahmed, and Arnold Overwijk. Approximate nearest neighbor negative contrastive learning for dense text retrieval. In *International Conference on Learning Representations*, 2020.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-edit: Fault-aware code editor for code generation. pp. 769–787. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.acl-long.45.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, E. Xing, et al. Judging LLM-as-a-judge with MT-Bench and chatbot arena. pp. 46595–46623. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2023. doi: 10.52202/075280-2020.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, and Yongqiang Ma. LlamaFactory: Unified efficient fine-tuning of 100+ language models. pp. 400–410. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-demos.38.

Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *MAPS@PLDI*, pp. 21–29. ACM, 2022. doi: 10.1145/3520312.3534864.

A Hole mining details

Source corpus. All code comes from public, permissively licensed repositories with at least 50 stars, mined in March 2026. For each of eight languages we select target third-party packages that are both widely adopted and actively maintained, using adoption statistics, package-registry rankings (PyPI, npm, Maven Central, NuGet, crates.io, Go modules, Packagist, RubyGems), and language-expert priority lists. Each package is pinned to a version released within six months of the mining date, and we keep only repositories that depend on that version or later. Files outside the 2nd–98th percentile of size or line count are removed. These filters are deterministic.

Candidate enumeration. Each file is parsed with its Tree-sitter grammar, and every call expression becomes a candidate with its byte span. A candidate can become a hole only if the answer starts after `.`, `->`, `::`, `?->`, or `new`, which places the cursor where an editor would request a completion while the call is being typed.

API classification. GPT-4.1-mini receives the whole file, an indexed list of candidate spans, and the closed list of target APIs for the language. It returns a JSON object that maps each candidate to one API from the list or to "abstain". Decoding is greedy and constrained to the schema. The instructions forbid names outside the list and require abstention when the evidence is weak or the called function is defined in the same file. Abstentions are discarded. The classifier thus only decides which library a call belongs to; it never scores holes or places the cursor.

Split rules and checks. The prefix, answer, and suffix are computed from byte offsets. Member calls are cut right after the access token, constructors after `new`, and a trailing semicolon joins the answer. Nested calls keep the surrounding expression in the prefix and suffix, and a call chain yields one hole per link. For example, `x = df.groupby('col').mean()` gives holes after `df.` and after `df.groupby('col').`. A hole is kept only if prefix, answer, and suffix reproduce the file byte for byte. Unit tests pin the expected split for member calls, chains, nesting, static access, and constructors in every supported grammar.

Random-hole baseline. For the comparison in Table 3, we sample the same number of holes from the same files but place the cursor at a uniformly random token boundary and take an answer span

whose length is drawn from the length distribution of our API-call holes. All other settings, including the number of updates, are identical to the 5K SFT run. The *all-call-site* baseline uses the same 5K files and keeps their import holes; every other hole moves to a call drawn uniformly from all calls in the file, including calls to functions defined in the same file and to the standard library (a median of 61 calls per file). The cursor and answer follow our split rules, which recover 82% of the original API holes exactly.

Decontamination and API overlap. No Delulu, HumanEval, SAFIM, or Real-FIM-Eval repository appears in the training pool, and a MinHash search (word 5-gram shingles, Jaccard > 0.8) finds no Delulu file that nearly duplicates any file in the 5K training set. Overlap at the API level is expected, because developers use the same popular libraries: 29% of Delulu holes call a method whose name, in the same language, is also a training answer. SFT 5K improves EM by 21.6 points on these holes and by 14.8 on the other 71%, so most of the gain does not come from recalling APIs seen in training.

Table 7: Holes per language in the mined pool, the curated 100K training set, and the Delulu evaluation set. C++ and TypeScript are evaluation-only.

Language	Mined	Curated	Delulu
C#	71,832	15,938	246
Java	55,122	14,793	243
PHP	53,897	14,756	–
Go	51,008	14,832	291
JavaScript	49,710	14,079	–
Python	47,241	12,353	374
Ruby	35,406	7,117	–
Rust	35,340	6,018	251
C++	–	–	125
TypeScript	–	–	420
Total	**399,556**	**99,886**	**1,950**

Curation. From the generated pool we draw a balanced 100K-row training set with at most 4,000 rows per (language, category) cell and at most 50 rows per repository, and remove every file and repository that appears in Delulu. The smaller budgets are nested prefixes of one fixed ordering. Answers are short (median 38 characters), while prefixes (median 1,931 characters) and suffixes (median 2,792) give rich context. Figure 5 shows the category mix.

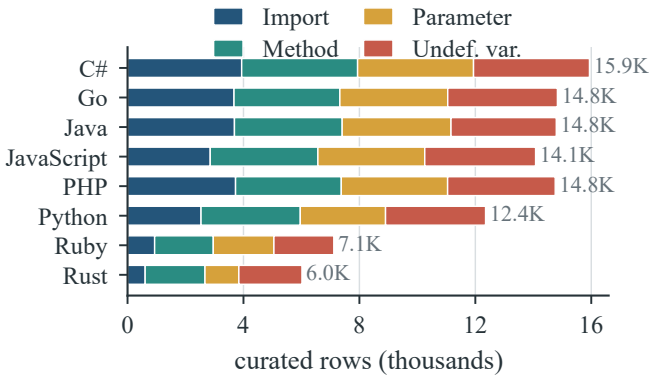


Figure 5: Curated rows by language and hallucination category.

B VALIDATING THE GENERATED NEGATIVES

Structural contract. Every generator must return a single line in a fixed format; the parser rebuilds the full triple and, for imports, checks that the file is reconstructed exactly (Appendix B.1). Of 2,484,731 candidates, 2,473,312 pass (99.5%). The rejects are 9,580 format failures, 1,816 missing

fields, and 23 transient API errors. Import has the lowest acceptance (98.3%) because some sampled files contain no import to change; the other categories exceed 99.9%.

Category audit. For each of the 12 (generator, category) cells we sample 3,000 accepted rows (36K in total) and check that the edit realises the requested category: a called name changes for METHOD; the call is kept but a new argument appears for PARAMETER; a new non-call name appears for UNDEFINED VARIABLE; the imported module or symbol changes for IMPORT. Conformance is 99.8% (import), 87.7% (method), 91.6% (parameter), and 90.3% (undefined variable), 92.3% overall and 90.6–94.2% per generator (Table 8). The check is conservative. Manual inspection shows that most flagged method and parameter rows are real matches that the detector misses on generic calls such as `M<T>(args)`, and most flagged imports are reordered path segments that are still valid fabrications.

Table 8: Acceptance over the full pool and category conformance on a balanced 36K sample.

Category	Accepted	Rejected	Accept. %	Conform. %
IMPORT	574,507	9,965	98.3	99.8
METHOD	645,809	471	99.9	87.7
PARAMETER	612,076	453	99.9	91.6
UNDEF.	640,920	530	99.9	90.3
Overall	2,473,312	11,419	99.5	92.3

Are the negatives really wrong? The category audit checks the kind of edit, not whether the result is invalid. For imports, validity can be checked without building the project: a fabricated package either exists in its public registry or it does not. We take all 2,181 import negatives written for the aligned 5K holes and keep the 789 whose edit changes the package itself and whose original package resolves in PyPI, npm, crates.io, the Go module proxy, RubyGems, or NuGet; the rest change a module or namespace inside a real package, or use a language without a simple package mapping, and cannot be checked this way. Only 8.1% of the fabricated packages exist under that name, and the rate falls as the negatives get harder (Table 9). Convincing negatives are therefore not convincing because they name real packages; developers and judges accept them because they look right.

Table 9: Import negatives whose fabricated package exists in its public registry, by fool-rate stratum (aligned 5K holes).

Stratum	Checked	Package exists	Rate (%)
Easy (no judge fooled)	440	42	9.5
Medium (one fooled)	154	13	8.4
Hard (two or three)	195	9	4.6
All	789	64	8.1

Pool size. GPT-5.4 and GPT-5.5 cover the whole corpus and contribute 56.1% and 40.0% of curated rows; GPT-5.2-Codex covers a subset and contributes 3.9% (Table 10).

Table 10: Valid generated rows (thousands) per generator and category.

Generator	IMPORT	METHOD	PARAM.	UNDEF.
GPT-5.2-Codex	26.3	31.3	31.3	27.2
GPT-5.4	315.5	357.3	335.9	354.9
GPT-5.5	232.7	257.2	244.8	258.8

B.1 IMPORT RESTRUCTURING

An import mistake must appear at file scope, not at the original hole. We give the generator the whole file and ask for one real import line and a fabricated replacement. The parser finds the original line, splits the file there into a new prefix and suffix, and checks exact reconstruction. Evaluation uses the

same restructured prefix and suffix. An early version left the hole at the call site, which placed the import after its first use and leaked the answer through the suffix; moving the hole fixed this.

C PROMPTS

Each generator prompt ends with a strict one-line output contract. The user message gives the prefix, the developer code, and the suffix (the whole file for IMPORT).

> **method**: Replace the method name in the completion with an invented method name that does NOT appear in prefix or suffix. The new name must be plausible; do not signal that it is invented. Output exactly: "Generated completion: <. . . >".
>
> **parameter**: Inject a non-existent keyword or positional argument into the call. The new parameter must be plausible. Keep the function name and the other parameters unchanged.
>
> **undefined variable**: Introduce a reference to an identifier that is NOT defined in prefix or suffix. It must be plausible for the language but fail at runtime.
>
> **import**: Given the full file, pick one real import line and replace it with a plausible but fictitious package or symbol. Output "Original import: <. . . >" and "Hallucinated import: <. . . >".

Judge prompt. Each judge sees the prefix, the candidate, and the suffix, and is not told that the candidate may be wrong:

> You are evaluating a code completion drawn from the middle of a source file. Decide whether the candidate is CORRECT (1) or NOT CORRECT (0). Score 1 only when the completion keeps the file valid, is consistent with the surrounding context, advances the apparent task, and does not hallucinate APIs, parameters, identifiers, modules, or behaviour. Output "Detailed Reasoning: <. . . >" and "Final Score: <0 or 1>".

D TRAINING AND EVALUATION SETTINGS

We train with LLaMA-Factory (Zheng et al., 2024), full fine-tuning, DeepSpeed ZeRO-3, bf16, and FlashAttention-2 on one node with eight H100 GPUs. Table 11 lists the shared settings. DPO keeps a frozen copy of the base model as reference and uses a per-device batch of one with more gradient accumulation, so the global batch stays at 256. Evaluation uses vLLM with greedy decoding and at most 256 new tokens. The negative writers are never used at evaluation time. The whole project, including discarded preliminary runs, used about 1,800 H100-hours: 1,250 for training and 550 for evaluation.

Table 11: Shared training settings.

Setting	Value
Learning rate	5×10^{-6}, cosine schedule
Global batch	256 (per-device 4, accumulation 8)
Context length	8,000 tokens
Epochs	7 at 5K, 2 at 20K, 1 at 35K and above (137–156 updates up to 35K)
ORPO	$\beta \in \{0.05, 0.1, 0.5, 1.0\}$, no reference model
DPO+NLL	$\beta = 0.1$, $\lambda \in \{0.25, 0.5, 1.0, 2.0\}$, frozen reference

D.1 METRICS

Exact match (EM) counts predictions equal to the developer code up to whitespace. Edit similarity (ES) is one minus normalised Levenshtein distance. CodeBLEU (Ren et al., 2020) uses weights $(1/3, 1/3, 1/3, 0)$; we drop data-flow matching because short completions rarely yield a data-flow

graph, compute it per language, and report the row-weighted mean. Real-FIM-Eval has no language labels, so we report no CodeBLEU for it.

The *verified hallucination rate* uses the per-item Delulu containers. Each prediction is spliced into its file and passed to the container's `verify patch` command, which compiles or type-checks the file in the pinned environment; a prediction counts as a hallucination if the check fails. Predictions that equal the developer code always pass, so the rate is at most $100 - \text{EM}$.

Stopping. To separate wrong content from failure to stop, we also score each prediction after cutting it to the number of lines in the reference. This oracle never changes decoding. The difference between the two scores measures over-generation.

Table 12: Delulu and SAFIM control flow with all three metrics. Each cell is untruncated / truncated to the reference length.

	Delulu			SAFIM control flow		
Model	EM	ES	CB	EM	ES	CB
Base	42.9/48.8	.628/.770	.397/.466	4.8/7.7	.099/.510	.298/.341
SFT 5K	59.7/59.8	.843/.841	.502/.505	28.8/28.8	.674/.679	.368/.394
ORPO 5K	59.6/59.5	.835/.834	.494/.495	31.6/31.6	.685/.688	.381/.392
DPO+NLL 5K	58.0/57.9	.824/.822	.492/.494	32.3/32.3	.683/.689	.379/.394
SFT 20K	60.0/60.0	.846/.846	.507/.509	31.4/31.4	.689/.694	.378/.410
ORPO 20K	60.1/60.2	.840/.839	.497/.500	32.9/32.9	.700/.702	.391/.404
DPO+NLL 20K	59.2/59.3	.837/.836	.491/.493	33.2/33.2	.699/.701	.395/.406

Table 13: Exact match (%) untruncated / truncated for the aligned 7B runs, including Real-FIM-Eval.

		5K			20K		
Benchmark	**Base**	SFT	ORPO	DPO	SFT	ORPO	DPO
Delulu	42.9/48.8	59.7/59.8	59.6/59.5	58.0/57.9	60.0/60.0	60.1/60.2	59.2/59.3
HE single	40.2/43.3	56.7/56.7	54.3/54.3	54.3/54.3	57.3/57.3	54.9/54.9	57.3/57.3
HE multi	16.1/19.5	30.5/30.5	24.6/24.6	28.0/28.0	28.8/28.8	23.7/23.7	26.3/26.3
HE random	34.6/40.2	47.2/48.8	43.3/44.1	45.7/45.7	48.8/50.4	45.7/46.5	48.0/48.0
SAFIM API	53.9/58.7	70.7/70.7	68.7/68.7	68.4/68.4	71.0/71.0	71.3/71.3	69.0/69.0
SAFIM block	12.4/30.1	27.9/28.1	27.1/27.1	27.4/27.5	27.9/28.1	26.1/26.2	26.9/27.0
SAFIM block-v2	13.4/37.6	34.6/34.7	33.7/33.7	34.1/34.3	35.2/35.3	32.8/32.8	34.1/34.1
SAFIM control	4.8/7.7	28.8/28.8	31.6/31.6	32.3/32.3	31.4/31.4	32.9/32.9	33.2/33.2
Real-FIM-Eval	3.4/5.0	3.6/3.7	3.5/3.6	3.4/3.5	3.9/4.0	3.6/3.6	3.6/3.6

E Objective sweeps and data budget

Figure 6 and Table 14 show that the comparison between objectives is not an artefact of one setting. ORPO changes by at most 0.31 EM at 5K and less than 0.1 at 20K across the temperature sweep. DPO+NLL improves steadily as the anchor weight grows, confirming that keeping the developer code likely is essential for FIM. Across budgets, Delulu is near 60% from 5K to 35K once updates are matched and reaches 61.7–62.1% at 100K (Table 15); the 100K runs are not update-matched, so they show scale rather than a ranking of objectives.

F Seed variance and data controls

Table 16 repeats the 5K runs with three training seeds (data order and initialisation of the random state; the hole list is fixed) and reports the baselines of Table 3 on every split. The differences discussed in the main text are larger than the seed spread, and random or all-call-site holes trail our holes on Delulu, SAFIM API, and HumanEval, while random holes are strongest on control flow.

Additional controls. For model-sampled negatives we draw eight completions per hole (temperature 0.8, top-p 0.95) from the base or the 5K SFT model, cut each to the line count of the developer

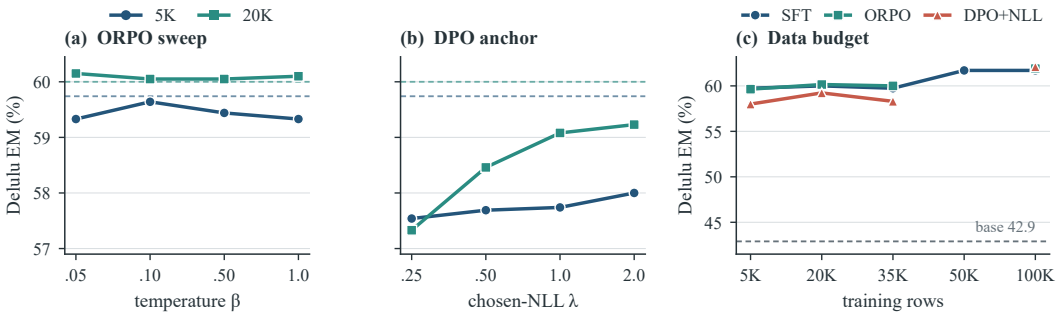


Figure 6: (a) ORPO temperature and (b) DPO anchor weight at 5K and 20K, with SFT as dashed lines. (c) Delulu EM across training budgets.

Table 14: Preference sweeps on Delulu (EM / ES / CodeBLEU).

Objective	Setting	5K	20K
ORPO	$\beta = 0.05$	59.33 / .836 / .491	60.15 / .840 / .497
	$\beta = 0.10$	59.64 / .835 / .494	60.05 / .841 / .495
	$\beta = 0.50$	59.44 / .834 / .498	60.05 / .841 / .495
	$\beta = 1.00$	59.33 / .834 / .504	60.10 / .840 / .496
DPO+NLL	$\lambda = 0.25$	57.54 / .819 / .479	57.33 / .820 / .471
	$\lambda = 0.50$	57.69 / .824 / .490	58.46 / .831 / .486
	$\lambda = 1.00$	57.74 / .826 / .492	59.08 / .834 / .487
	$\lambda = 2.00$	58.00 / .824 / .493	59.23 / .837 / .491
SFT	–	59.74 / .843 / .502	60.00 / .846 / .507

code, and keep the most frequent one that uses a name the file does not define (comments ignored); failing that, the most frequent completion that differs from the developer code. Holes where every sample reproduces the developer code keep their frontier negative, so the hole set is unchanged. For the base model, 40% of the kept negatives invent a name, 24% are other wrong completions, and 36% fall back to the frontier negative. The 5K SFT model reproduces the developer code in all eight samples for 67% of its own training holes, so only a third of its set comes from its own samples (21% invent a name, 12% are other wrong completions). SFT→ORPO continues the 5K SFT checkpoint with ORPO ($\beta = 0.1$) for 137 further updates, and is compared with SFT trained for the same total number of updates (59.5% on Delulu and 32.8% on SAFIM control flow, against 59.5% and 31.5% for SFT→ORPO). For the length-controlled strata we match easy, medium, and hard pairs on category, character edit distance between g and h (bins of 4), and negative length (bins of 16 characters) before training. For "hard pairs" we keep, for each hole, the negative with the highest fool rate among the two to three candidates the teachers wrote for that hole's category (ties keep the original negative), so the hole set and category mix are identical to the other 5K runs. This changes the negative for 814 of the 5,000 holes and raises the mean fool rate from 0.22 to 0.29.

Where the gains come from. Table 17 breaks Delulu down by category for Qwen2.5-Coder-7B-Instruct. SFT helps every category, most of all imports (+23.5 points, against about +15 for the other three). Among the verifier failures that remain after SFT, 38% are type or arity errors in an otherwise valid call, 27% invent a name, 19% import a wrong or missing module, and 16% are syntax or boundary errors.

G What the training data should contain

Table 18 varies one property of the training set at a time. These runs use a single epoch (about 125 updates for 32K rows and 62 for 16K rows) rather than the update-matched schedule of the main text, so their absolute scores are lower, especially for the 16K language runs, and only comparisons within a block are meaningful. Method, parameter, and undefined-variable data behave alike and each reach about 57% on Delulu, which suggests they teach a shared skill of not inventing names.

Table 15: Delulu EM / CodeBLEU across budgets. 5K–35K are update-matched; 50K is SFT only; 100K runs are single-epoch endpoints.

Rows	SFT	ORPO	DPO+NLL
5K	59.7 / .502	59.6 / .494	58.0 / .493
20K	60.0 / .507	60.2 / .497	59.2 / .491
35K	59.7 / .485	60.0 / .492	58.3 / .485
50K	61.7 / .504	–	–
100K	61.7 / .512	61.9 / .506	62.1 / .515

Table 16: 5K runs, mean ± standard deviation over three seeds (EM %). Tables in the main text report the first seed.

Run	Delulu	HE-SL	HE-ML	HE-RS	SA-API	SA-Ctl
SFT, random holes	54.4±0.3	53.7±0.8	28.0±1.0	42.5±0.9	64.5±0.4	33.8±0.7
SFT, all call sites	59.2±0.2	55.2±0.8	29.0±1.0	45.3±0.9	69.5±0.4	30.4±0.6
SFT, our holes	59.7±0.2	56.9±0.4	29.9±0.5	47.8±0.9	70.6±0.3	28.0±0.7
ORPO, our holes	59.8±0.2	55.1±0.7	24.9±0.5	44.6±1.2	69.6±0.8	30.5±1.1
DPO+NLL, our holes	58.0±0.0	54.7±1.3	29.4±1.3	47.5±1.6	68.9±0.5	31.9±0.6

Import-only data reaches 25.5% on Delulu and nearly fails on multi-line and control-flow completion, because import answers are short and follow a narrow syntax. Language breadth helps at a fixed budget: five languages without Python beat Python-only training even on Python (Figure 7), although the Python-only run is also the most undertrained under this schedule. At 3B, the mixed 20K set is best on Delulu and method-only data is best on the general splits; with the full 100K set, the 3B model reaches 58.7%.

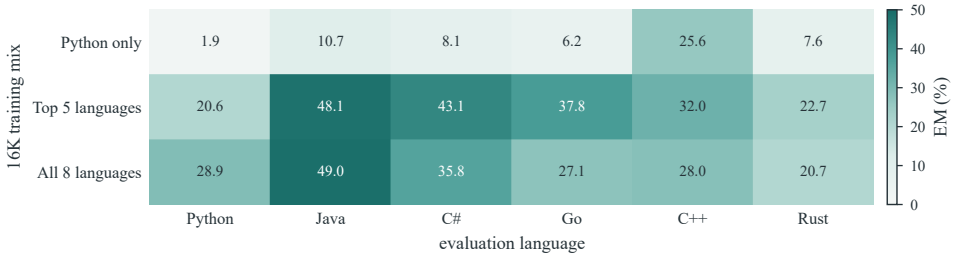


Figure 7: Per-language Delulu EM for the 16K language-coverage runs. The five-language set contains no Python.

Model families. Table 19 gives the raw numbers behind Figure 3. Each model uses its own FIM tokens on the same holes.

H FOOL RATE: DISTRIBUTION AND HUMAN STUDY

Most frontier-written negatives are caught by at least one judge. On the aligned 5K holes, where all 9,964 candidate negatives are scored by all three judges, the mean fool rate is 0.22: 60% of negatives fool no judge, 22% fool one, 11% fool two, and 8% fool all three. Imports are the hardest to catch (mean 0.33), because a fabricated package name cannot be checked without a registry lookup, followed by parameters (0.28); invented methods (0.18) and undefined variables (0.11) are easier to spot. Imports make up 45% of the negatives that fool all three judges but only 21% of the set. Rule-based negatives are caught far more often (mean 0.09; 82% fool no judge).

C# strata. For the controlled study, each stratum holds 5K C# pairs with identical category counts: easy pairs fool no judge ($\tau = 0$), medium pairs fool one ($\tau = 1/3$), and hard pairs fool two or three ($\tau \geq 2/3$). Because hard pairs are the scarcest, the per-category quota is set by the smallest cell, and the other categories fill the rest of each stratum in the same proportions. SFT and ORPO train for 7

Table 17: Delulu EM / verified hallucination rate (%) by category, Qwen2.5-Coder-7B-Instruct, 5K holes.

Model	IMPORT	METHOD	PARAM.	UNDEF.
Base	39.4/34.8	41.4/28.2	48.3/28.5	43.0/26.2
SFT	62.9/21.2	56.4/11.1	63.0/9.7	57.4/9.2
ORPO, hard pairs	63.4/20.9	58.1/10.4	64.6/8.3	58.9/8.4

Table 18: Training-set composition (7B unless noted; EM %). These are single-epoch runs with far fewer updates than the main experiments, so their absolute scores are not comparable to the main text; compare rows only within a block. Category cells use about 32K rows and language cells 16K rows.

Training subset	**Delulu**	**HE-SL**	**HE-ML**	**HE-RS**	**SA-API**	**SA-Ctrl**
Import only	25.5	55.5	0.8	34.6	15.8	0.4
Method only	57.3	52.4	24.6	40.2	71.6	24.1
Parameter only	57.0	50.6	25.4	39.4	70.3	24.0
Undefined-variable only	57.1	51.2	22.9	40.2	70.6	24.9
Python only	7.8	20.1	6.8	14.2	28.4	3.0
Five languages (no Python)	31.9	55.5	27.1	40.9	53.5	15.8
All eight	31.1	53.0	22.0	41.7	50.6	10.2
3B, 20K mixed	53.2	50.0	13.6	40.2	65.8	15.0
3B, method only	52.5	50.0	21.2	41.7	67.4	19.7
3B, five languages	48.6	42.7	11.9	40.2	64.5	12.8
3B, 100K mixed	58.7	51.8	11.0	41.7	69.0	36.4

epochs (about 137 updates). Eighteen pairs whose negative equalled the developer code are replaced within the same stratum and cell. ORPO results in Table 5 use a fixed $\beta = 1$; the best β per stratum gives the same ordering.

Human study. We sampled 400 negatives, 100 at each fool rate (0, 1/3, 2/3, 1), balanced across categories and languages. Three professional developers, each with at least five years of experience, saw the prefix, the candidate, and the suffix in an editor view, had up to 60 seconds per item, and answered whether they would accept the completion. They were not told that any candidate was wrong. Pairwise agreement was 0.61 (Krippendorff's α). Acceptance rose with fool rate: 9%, 18%, 29%, and 41% for 0, 1, 2, and 3 fooled judges.

Cross-family judges. To check that the fool rate is not a quirk of GPT-family judges, two judges from other families, Claude Haiku 4.5 and a small code-specialised model from a third provider, score 1,500 pairs from the C# strata (500 per stratum) with the same blind prompt. Their fool rate correlates with the GPT panel's across items (Spearman $\rho = 0.50$, $p < 10^{-90}$), and their mean fool rate rises from easy to medium to hard pairs (0.07, 0.24, 0.51), for each judge separately as well.

I TEACHER FAMILIES

All three teachers receive the same 5K holes, prompts, and quotas. We keep the 4,861 holes where every teacher returned a valid negative and retrain the GPT arm on this shared set, so only the author of the negative changes. Larger spreads appear only on the small HumanEval splits (118–164 items), where one completion moves EM by 0.6–0.8 points, and no teacher leads consistently (Table 20).

J TRANSFER BEYOND THE TRAINING CATEGORIES

We build a pairwise test from the human-annotated release of Liu et al. (2026). We keep every CoderEval Python and Java task with at least one annotation (345 tasks, 11 categories) without mapping them to our four categories. The canonical function body is placed in the FIM hole as the correct member of each pair, and each body that the annotators marked as hallucinated is the incorrect member, giving 952 pairs. A pair is correct when the canonical body has the higher mean token

Table 19: Three 7B families before and after SFT on the same 100K holes (EM %).

Family	Run	Delulu	HE-SL	HE-ML	HE-RS	SA-API	SA-Blk	SA-Blk2	SA-Ctl
Qwen2.5-Coder	Base	42.9	40.2	16.1	34.6	53.9	12.4	13.4	4.8
	SFT	61.7	57.3	22.9	44.9	72.6	26.3	32.9	39.2
StarCoder2	Base	43.9	51.8	24.6	38.6	57.7	26.1	31.1	36.3
	SFT	59.3	51.2	18.6	42.5	68.1	29.8	36.3	40.6
CodeLlama	Base	23.6	7.3	6.8	7.9	45.5	7.2	10.6	24.9
	SFT	58.7	49.4	11.0	34.6	61.6	18.3	22.7	26.7

Table 20: Negatives written by different teachers (EM %, same 4,861 holes).

Objective	Teacher	Delulu	HE-SL	HE-ML	HE-RS	SA-API	SA-Blk	SA-Blk2	SA-Ctl
ORPO	GPT panel	60.0	55.5	27.1	43.3	69.0	27.2	34.0	29.7
	Claude Sonnet 5	60.0	55.5	26.3	43.3	70.0	27.2	34.2	29.9
	Gemini 2.5 Pro	60.2	55.5	26.3	43.3	70.0	27.2	34.1	30.1
DPO+NLL	GPT panel	58.5	53.1	29.7	48.0	69.7	27.2	34.5	31.1
	Claude Sonnet 5	58.9	54.9	31.4	46.5	70.0	27.3	34.3	31.6
	Gemini 2.5 Pro	58.8	55.5	31.4	47.2	69.4	27.7	34.6	31.4
Evaluation items		1950	164	118	127	310	8781	4571	8629

log-probability. Intervals use 20,000 bootstrap samples over tasks. Because the source annotates whole functions, free-form generation is near zero for every model, so we report only pairwise accuracy. Absolute accuracy is low for all models (11.2% for the base model and 21.3% at best). A likely reason is that the hallucinated bodies were themselves written by code models, so any code model finds them likely; what fine-tuning changes is the relative ranking (Table 21; Figure 4 shows the per-category gains).

Table 21: Pairwise accuracy on human-annotated CoderEval hallucinations. The 100K runs are not update-matched across objectives.

Model	Accuracy (%)	Gain over base [95% CI]
Base	11.2	–
SFT 20K	18.2	+6.9 [4.9, 9.1]
ORPO 20K	16.9	+5.7 [3.7, 7.8]
DPO+NLL 20K	18.0	+6.7 [4.6, 8.9]
SFT 100K	19.6	+8.4 [6.1, 10.9]
ORPO 100K	21.3	+10.1 [7.5, 12.8]
DPO+NLL 100K	20.5	+9.2 [6.7, 11.9]

K LIMITATIONS AND DATA RELEASE

Scope of the negatives. The four categories cover API-level hallucinations. Logic errors, asynchronous misuse, and type drift need their own prompts. The CoderEval test shows transfer to other categories, but only as a ranking of given alternatives.

Real edits. On Real-FIM-Eval every model, base or trained, stays below 4% EM, because exact match must reproduce one particular free-form edit. Our method does not hurt these edits, but it does not make a model a better general programmer.

Judge family and language. The main fool-rate judges are GPT-family models; judges from other families rank pairs consistently with them (Appendix H), and the teacher study shows that the negatives themselves are not tied to one family. The controlled fool-rate study uses C# only; repeating it across languages is left to future work.

Data release. The source files are public and permissively licensed. We release the code for every pipeline stage after hole extraction: negative generation, validation, fool-rate judging, curation, FIM formatting, and training recipes. The curated holes and generated negatives are not released, but the released code rebuilds an equivalent set from any repository snapshot.