SYNTHCODE: NOT ALL SYNTHETIC CODE IS WORTH THE SAME TOKENS

Anonymous authors
Paper under double-blind review

ABSTRACT

Synthetic data now underpins code pretraining, yet the decisions that produce it are made largely by intuition: which transformation to apply to a source file, whether to train on the generated artifact alone or pack it with its source, and whether a fixed token budget is better spent on new documents or on new views of the same ones. Because these choices are rarely compared at equal cost, their relative value is unknown. We introduce SynthCode, a 5.1M-document corpus distilled from 6.4M Python commits with gpt-oss-120b, together with a token-matched protocol for isolating the value of each generation decision. The protocol covers sixteen recipes: single-transformation distillation, five context-packing formats, equal-token mixtures, and a three-way source-overlap study that prices a second view of one document against a first view of two. Every recipe is trained for the same number of tokens under identical hyperparameters, so differences are attributable to the data rather than to compute or tuning. We then train the best-performing recipe into six base models spanning two families and 4B–31B parameters (Qwen3-4B/8B/14B, Gemma4-12B/26B/31B). We release the corpus, the generation and filtering pipeline, and all checkpoints.

1 INTRODUCTION

Early open code models were trained on large but largely unstructured corpora: deduplicated source files scraped from GitHub, supplemented with documentation and web text (Kocetkov et al., 2022; Li et al., 2023a; Lozhkov et al., 2024). Recent open-weights models take a more deliberate approach. Technical reports for Qwen (Yang et al., 2025), GLM (GLM-4.5 Team et al., 2025), and Kimi (Team et al., 2026) describe mid-training stages that rewrite raw code with a teacher model and repack it alongside repository-level signals such as commit messages, pull requests, and file-system structure, so that training sequences look more like the edit-driven tasks these models later perform as coding agents. However, these descriptions are brief, the data is not released, and the design choices are never ablated in public. A practitioner building a mid-training mixture is left to guess at two questions: *how should raw code be cleaned or rewritten*, and *how should the result be packed with its surrounding context into a training sequence*?

This paper builds a controlled testbed for both questions and reports a six-model, ten-strategy ablation. Every strategy is derived from the same commits in CommitPack (Muennighoff et al., 2024), each of which supplies a pre-commit file, a post-commit file, and a human commit message, so strategies differ only in how that material is presented. The testbed has four components:

1. **Single-artifact rewriting.** We consider modifications to individual code documents. We train on either the raw file itself, which serves as the baseline; a cleaned up version with a descriptive header and inline comments; a Jupyter notebook rendering that explains each code cell; or a blog-style article about what the code does.
2. **Multi-artifact packing.** We pack training documents to place context before the post-commit file, so that it is structured like an edit with human-written prompt. The context combines the pre-commit file with a human or teacher-written commit message, a teacher-written planning document, or a teacher-written code review of the pre-commit file. Adjacent strategies differ in a single element, giving one-variable comparisons such as human versus generated commit messages and plans versus reviews.

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.

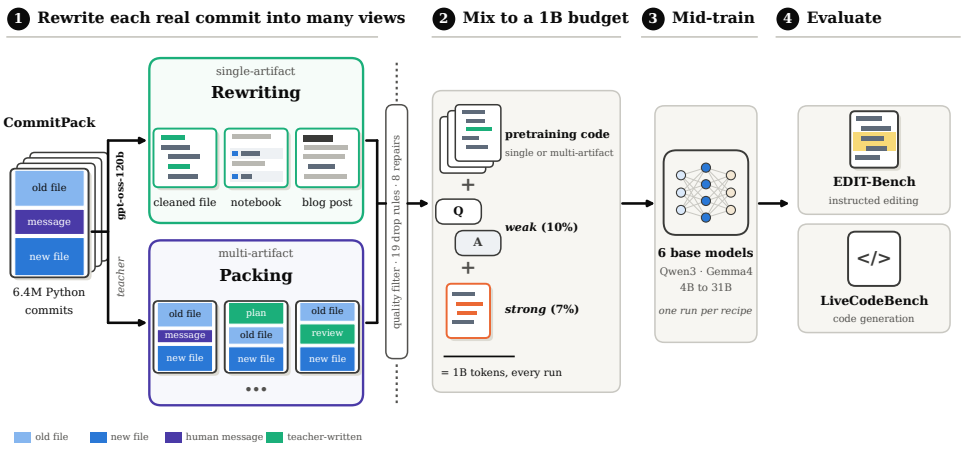

Figure 1: **Overview of the study pipeline architecture.** We rewrite GitHub commits into different views and test what data format yields the best results for pretraining coding models. We train strong coding models (Qwen3 and Gemma4 base models) and evaluate each trained strategy on coding benchmarks.

3. **Token-matched training runs.** Every strategy is trained for the same budget of 1B tokens, with a single set of hyperparameters and with the same slice of instruction data. Differences between strategies can therefore be attributed to the data rather than to compute or tuning.
4. **Unified evaluation protocol.** We train using each strategy with six base models from two families and evaluate every checkpoint on instructed code editing (EDIT-Bench) and code generation (LiveCodeBench). For two of these models, we also evaluate annealed checkpoints at 250M and 500M tokens to test whether strategy rankings are stable over the course of training.

Overall, we find that how the context is packed matters more than how the file is rewritten. Without *strong* edit instructions, mid-training on raw commit files degrades EDIT-Bench below a model instruction-tuned with no mid-training code at all, and each single-file transformation recovers most of that gap; once the instructions are present, rewriting the file stops helping but packing the pre-commit file with its commit message still does. Rewriting the human commit message into a concise synthetic one (OLD+TERSECM) is the only strategy that beats single-file training in both settings, heading a cluster of commit-message strategies that take the top three EDIT-Bench ranks across all six models, with the cleaned-file rewrite (CLEAN) the one single-file transformation to join them. The ordering shifts on LiveCodeBench, where the leaders are strategies that carry no commit message at all (CLEAN, BLOG, PLAN, OLD+REVIEW) and commit-message packing falls to the middle, while raw code and plan-with-source packing sit at the bottom of both benchmarks.

2 RELATED WORK

Synthetic data for code models. The "textbooks are all you need" line of work argues that filtering and regenerating code into pedagogical form yields disproportionate capability per token (Gunasekar et al., 2023; Li et al., 2023b). Instruction-tuning recipes built on the same premise such as Self-Instruct (Wang et al., 2023), Evol-Instruct (Luo et al., 2025), and OSS-Instruct (Wei et al., 2024), generate instruction/response pairs from seed code and report large HumanEval gains. These recipes are validated end-to-end: a mixture is proposed and a benchmark number is reported. What is rarely reported is a per-transform ablation at fixed token budget, which is what this work provides on six base models spanning two families.

Closer to our setting is the rewriting line, in which a teacher transforms existing documents rather than generating new ones. WRAP (Maini et al., 2024) paraphrases web documents into target styles for a pretraining speedup, and SwallowCode (Fujii et al., 2026) applies style-guided rewriting to 16.1B tokens of Stack-v2 Python, reporting +17.0 HumanEval pass@1 after continued pretraining.

A controlled study of web-text rephrasing finds that rephrased data alone does not pretrain faster than natural text and helps only in mixtures (Kang et al., 2025); we ask the same question for code, where the answer can be graded by execution rather than validation loss. Waheed et al. (2025) take a related approach, perturbing structural and semantic properties of code *instruction* data and measuring the effect on general reasoning; we ask the analogous question in the continued-pretraining setting, with code benchmarks as the target. The closest methodological relatives are data-mixture studies such as DoReMi (Xie et al., 2023), DataComp-LM (Li et al., 2025), and FinePhrase (Niklaus et al., 2026), which likewise emphasize controlled comparison but target general pretraining rather than transforms of code.

Packing and repetition. How documents are placed in a training sequence affects what is learned and which documents share a context changes pretraining outcomes (Zhao et al., 2024). Under a fixed token budget, up to about four epochs of repeated data are worth nearly as much as unique data (Muennighoff et al., 2025). We ask the analogous question for synthetic views: whether a second view of a document is worth a first view of another.

Code edit and repair data, and the scarcity of code evaluation. CommitPack and CommitPackFT (Muennighoff et al., 2024) established permissively licensed commit histories as a source of edit-shaped supervision, and CanItEdit (Cassano et al., 2024) established instructional code editing as a distinct evaluation target from synthesis, pairing each of 105 problems with an explicit before-state, a reference after-state, executable tests, and both a verbose and a terse phrasing of the same instruction. OctoPack trains on commit pairs directly; we instead treat a commit as a source that can be rendered several ways, and compare those renderings at equal cost.

This distinction matters more for our claim than the literature's coverage suggests. A recent audit (Ebrahimi & Rajbahadur, 2026) screened over 150 code benchmarks and found only two — CanItEdit (Cassano et al., 2024) and EDIT-Bench (Chi et al., 2025) — that evaluate instructed code editing with human-authored instructions and test-based grading, while estimating that instructed editing constitutes roughly 19% of real coding-assistant interactions. EDIT-Bench is the larger and more realistic of the two, drawing 540 problems and their instructions from live Copilot Arena sessions. Code benchmarks also leak into training, both directly and through synthetic data from teachers that have seen them (Matton et al., 2024; Riddell et al., 2024), which is why our edit-instruction data is filtered against every evaluated split (Section 4).

3 Synthetic corpus construction

3.1 Source and generation

We use CommitPack (Muennighoff et al., 2024), which supplies, for each commit, the pre-commit file x_{old}, the post-commit file x_{new}, and the human commit message m. This shape is convenient: a single row is both a document that a teacher can transform and an instruction pair the model can be trained on directly. We exercise both readings (Section 3.1). We generate with `gpt-oss-120b` served through vLLM (Kwon et al., 2023) on an Intel Max-series GPU cluster.

The teacher produces one of six artifacts from each row, each targeting a distinct hypothesis about what makes synthetic code data valuable. The *cleaned file* rewrites x_{new} with a descriptive header and inline comments while preserving behavior (Appendix A, Figure 4), testing whether stylistic normalization of otherwise identical code carries any signal. The *notebook* converts x_{new} into a Jupyter notebook whose markdown cells explain the reasoning behind each code cell. The *article* is the blog-post rewrite that builds intuition for what x_{new} does. The *plan* describes the change from x_{old} to x_{new} as a project planning document, with objectives and scope. The *code review* critiques bugs in x_{old} and the fixes needed to resolve them. Finally, the *synthetic commit message* $\hat{m}$ is written by the teacher from the diff. Unlike the others, it is a *condition* on the target rather a replacement, which is what lets us swap for the human message m elsewhere in the pipeline. Section 4.1 composes these artifacts into our training strategies and describes the comparisons they support.

Table 1: Properties of the kept CLEAN and NOTEBOOK data, measured on 3,000 rows per transform sampled evenly across the corpus. Structural measures are over rows whose source parses as Python 3.

CLEAN*: the program is kept, the explanation is added*		
	source x_{new}	cleaned
functions and classes kept	–	99.4%
imports kept	–	99.9%
definitions with a docstring	34%	74%
comments per 100 lines of code	9.7	23.5
median length relative to source	1.00	1.31
NOTEBOOK*: a valid notebook that covers the source*		
valid nbformat-4 JSON		99.97%
source lines present in code cells (mean)		96.8%
notebooks covering $\geq$95% of source lines		86.8%
concatenated code cells parse		92.6%
median cells (markdown / code)		6 / 5

3.2 FILTERING AND DATA QUALITY

Every generated row passes a streaming filter with a fixed set of checks; each added only after after its target defect had been measured on a sample and two whose obvious form removed good data were narrowed (Appendix B).

Every row must come from a source file of at least 200 non-whitespace characters, from a commit that is not a version bump, and must carry no PEM private key. Rows must also be *complete* (i.e., the decoder did not hit its token limit) and generated from an untruncated input. A cleaned file must further parse whenever its source parses, keep every function argument name, reference no name the source does not define, and not be flagged as an abridgment; copyleft license headers are restored to the source's exact wording. This keeps 1.29M cleaned files (86.3%) and 1.19M notebooks (79.3%) from 1.49M generations respectively.

Prefix-campaign transforms pass the same filter minus two rules that check an instruction only the cleanup prompt carries. It keeps 383,368 blog posts (94.8%), 478,822 plans (94.7%), and 365,518 commit messages (70.9%). The lower commit-message yield is not a teacher failure: 23.8% of rows are dropped because the commit changed two lines or fewer, and a message conditioned on such a diff carries almost no signal. The same rule removes 23.4% of quarter-campaign commit messages, so the loss is a property of CommitPack, not of either prompt. Assembly applies three final gates: a 100-character floor per file and 20 per commit message, exact deduplication on the SHA-256 of the assembled text, and a refusal-substring filter that removes 4–29 rows per strategy.

Cleaned files keep the program and add the explanation. Table 1 shows the cleaning transform preserves the program's interface almost untouched: 99.4% of cleaned files keep every function and class of the source, and 99.9% keep every import. What changes is the explanation around it: The share of definitions with a docstring more than doubles, from 34% to 74%, and comment density rises from 9.7 to 23.5 comments per 100 lines of code. The median cleaned file is 1.31$\times$ the length of its source, and the added lines are documentation rather than new code. This is what makes CLEAN a clean test of the stylistic-normalization hypothesis: whatever the arm gains or loses at training time is attributable to the added explanation, not to a change in the underlying program.

Notebooks are complete and well formed. Nearly all kept notebooks are a valid nbformat-4 document (99.97%) whose cells interleave reasoning with code (median of six markdown cells to five code cells). The generated notebooks are faithful to the file they render: on average 96.8% of the source's non-comment lines appear in the code cells, identical up to indentation, and 86.8% of notebooks carry at least 95% of them. Concatenating a notebook's code cells yields a parseable program for 92.6% of parseable sources, so the notebook rendering preserves executability and content.

4 EXPERIMENTAL DESIGN

We aim to gain a better understanding of which generation decision is worth its tokens. This requires every comparison to be made at equal cost and with identical tuning procedure.

The protocol. Every strategy is a full-parameter continued-pretraining run of the same base model for exactly $1{,}000{,}219{,}392$ tokens ($10{,}176$ steps of $98{,}292$ tokens), under one set of hyperparameters that is never tuned per strategy. The documents are drawn from the single Gemma-fitted tree so both model families train on byte-identical data. Documents are packed to $8{,}192$ tokens with per-document attention masks (no cross-document attention inside a packed sequence), carry an empty prompt so that the whole document is supervised, and end in the model's EOS token. We describe the setting in more detail in Appendix D.

4.1 AXIS 1: TRANSFORM AND PACKING FORMAT

The first axis asks how the artifacts of Section 3 should be presented to the model at training time. A transform is a single teacher instruction applied to one CommitPack row. Our strategies are organized around two questions:

Single Artifact — What should the model read? Each sequence is a single artifact, and the strategies differ only in the form that artifact takes. Together they ask whether a token of transformed code is worth more than the source it came from.

- RAW: the post-commit file x_{new} as it appears in CommitPack. This is the baseline for the Single-Artifact strategies and the closest analogue of ordinary code pre-training.
- CLEAN: the teacher-cleaned version of x_{new}.
- NOTEBOOK: the notebook rendering of x_{new}, serialized a `.ipynb` JSON.
- BLOG: a blog-post style of x_{new}.

Multi-Artifact — How should instruction, plan, and code be packed? Each sequence is a context followed by the post-commit file x_{new}; only the context varies, and each strategy is named by the segments in its prompt, in order. We write x_{old} for the pre-commit file, m for the human commit message, $\hat{m}$ for a teacher-written one, p for a teacher-written planning document and r for a teacher-written code review. A commit message is always introduced by the `INSTRUCTIONS:` marker. When the context contains a file and a commit message, the file comes first, mirroring an agent that has already opened a file before reading its instructions; when the context contains a plan, the plan comes first, as it would if written before the work began.

- OLD+CM: $x_{\text{old}} + m$. An unmodified CommitPack row, with nothing generated.
- OLD+VERBOSECM/TERSECM: $x_{\text{old}} + \hat{m}$, where $\hat{m}$ is written from the diff. VERBOSECM prompts only for a message capturing intent; TERSECM additionally fixes its shape, yielding messages $2.1\times$ shorter in tokens (Figure 13). Together, and against OLD+CM, they isolate whether rewriting human commit messages helps. We carry both prompts into training and let evaluation choose between them.
- PLAN+OLD: $p + x_{\text{old}}$, where the teacher writes the plan p from x_{new} alone.
- PLAN+CM+OLD: $p + m + x_{\text{old}}$. Against PLAN+OLD, it isolates what the commit message adds when a plan is already present.
- PLAN: p alone, without x_{old}, so the model must produce x_{new} from the plan. Against PLAN+OLD, it tests whether the plan can replace the source file.
- OLD+REVIEW: $x_{\text{old}} + r$, where r is a defect-level review written by the teacher from x_{new}. Against PLAN+OLD, it tests whether the prose should describe what is wrong with the code rather than what to build.

Comparisons: The multi-artifact strategies form four one-variable comparisons: human vs. generated commit message (OLD+CM vs. OLD+VERBOSECM/VERBOSECM), plan with vs. without x_{old} (PLAN+OLD vs. PLAN), plan alone vs. plan plus commit message (PLAN+OLD vs. PLAN+CM+OLD), and plan vs. review as the prose (PLAN+OLD vs. OLD+REVIEW).

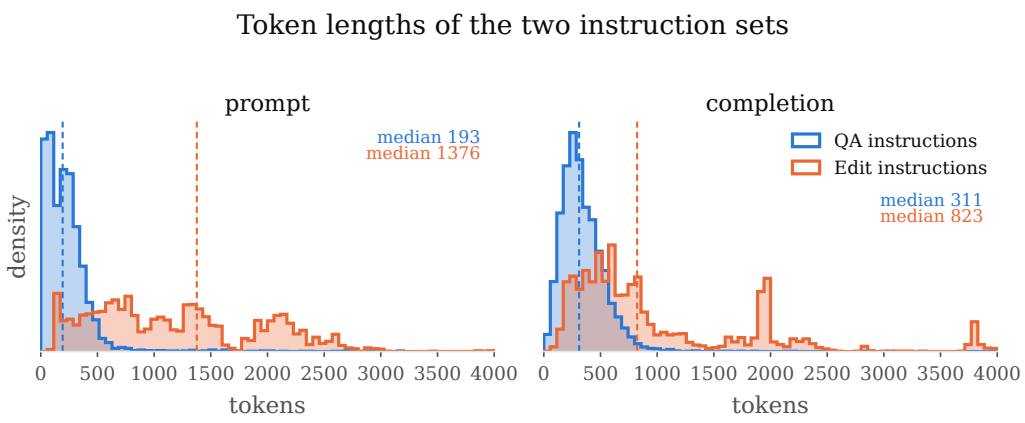

Figure 2: Length of the two instruction datasets used during training: the general QA set (*weak*) and the *strong* instruction dataset (*strong*). The *weak* dataset has shorter and simpler samples – prompts are over 1000 tokens shorter on average and completions over 500 tokens shorter.

Output form varies; source content is fixed. All transforms derive from the same x_{new}, so any difference between the strategies reflects the form the teacher was asked to produce, not the underlying code. Cleaned files and notebooks return code; plans, reviews, and blog posts return prose about code; and commit messages return a description of the edit, written with both x_{old} and x_{new} in view. This last output is the only one that acts as a condition on the target rather than a replacement for it, which is what makes the packing axis possible: a generated message can be placed before x_{old} and x_{new} to form an edit-shaped document, or swapped for the human commit message that CommitPack ships, with everything else held constant.

4.2 Axis 2: instruction data in the mixture

For benchmark scores to be meaningful, the trained checkpoints need to be able to following benchmark instructions. We inject instruction data into each experimental dataset to teach instruction following. For every strategy, 10% of training tokens come from general code-instruction data: Evol-CodeAlpaca ($111{,}272$ rows) and Magicoder OSS-Instruct ($75{,}197$ rows). These are rendered with the model family's chat template, and loss is computed on the response tokens only. The interleaving probability is set per strategy from its measured tokens per document, so the token share is 10% in every strategy. This ensures that a benchmark prompt is not the first chat-formatted text the model has seen since its base checkpoint. Because it is present in every strategy, it cannot account for differences between them. Its responses are short (Figure 2) and simple, so we do not expect it to teach a new skill; we call this set the *weak* instruction data.

We additionally include a *strong* instruction set of $18{,}103$ instruction-conditioned whole-file edits derived from EDIT-Bench's training material, with a mean completion length of 828 tokens using the Qwen3-4B-Instruct-2507 model for generating the instructions, and Qwen2.5-Coder-7B-Instruct to generate the code. Because this set shares provenance with a benchmark we evaluate on, we filter it for leakage against *every* evaluated split. We match on the SHA-256 hash of whitespace-normalized code rather than on problem identifiers, since ID spaces differ between the source and the benchmark. The filter removed $14{,}039$ of $32{,}142$ source rows (43.7%). Without it, gains from the treatment could reflect memorization of benchmark problems rather than learning.

4.3 Axis 3: training budget, via annealed milestones

A single run can be benchmarked at multiple intermediate checkpoints if each are annealed to a low learning rate. If a mid-run checkpoint sits at a higher learning rate than the final one, comparing them would confound "saw fewer tokens" with "was not annealed." On one strategy, annealing alone improved EDIT-Bench pass rate by roughly 0.06 (absolute) , confirming that this confound can obscure budget trends.

We therefore adopt a warmup-stable-decay schedule with branched decays (Hägele et al., 2024). The parent run holds a constant peak learning rate after warmup. At three token milestones (250M, 500M, and 1B), a separate job branches from the parent checkpoint and linearly decays the learning rate to 10^{-7} over a tail equal to 10% of the tokens seen at that point. Each branch resumes with the optimizer state and dataloader position intact, so it neither re-estimates Adam moments nor replays seen data. Each branch is thus a standalone run at its own budget, annealed identically to the others, so the three milestones of one strategy can be compared directly with the three of another. We produce all three milestones for every strategy on Qwen3-8B and Gemma4-12B.

4.4 EVALUATION

The ablation sweep is run with Qwen3-8B and Gemma4-12B. To ask whether the strategy ranking is a property of the data or of one model, the full set of 22 configurations (11 strategies $\times$ {weak, strong}) is then trained into six base models: Qwen3-4B, 8B and 14B, and Gemma4-12B, 26B and 31B. We evaluate on benchmarks targeting instructed code editing, code generation and code understanding under a single decoding protocol.

EDIT-Bench (Chi et al., 2025) is our primary benchmark, spanning tasks such as bug fixes, feature additions, and refactoring. Each problem provides a source file, a highlighted region within it, and a natural-language editing instruction drawn from a live Copilot Arena session. Given these inputs, the model must return the complete edited file, which is then evaluated against the problem's unit tests inside an isolated Apptainer sandbox: each problem gets its own sandbox with a fresh dependency install, via uv for the 97 Python problems and npm for the 11 JavaScript/React problems. We evaluate on the full 108-problem `test` split, drawing one sample per problem as specified by the benchmark, and report `pass_rate`, the fraction of problems on which all tests pass.

LiveCodeBench (Jain et al., 2024) measures competitive code generation. To avoid contamination from the base models' training data, we use the code-generation scenario of release v6 dated February–April 2025 which consist of 131 problems. We sample five samples per problem and report pass@1.

We fix decoding across benchmarks and model families: temperature: 0.6, top-p: 0.95, top-k: 20, and 8,192 new tokens. Two family-specific choices address base-model behavior: for Qwen, we stop generation at the closing code fence the prompt requires, since the continued-pretrained checkpoints otherwise never emit EOS and our extractor would select a later hallucinated block over the correct first one. This stop is not viable for Gemma, whose prose-heavy checkpoints often open with a sentence before the first fence and would be truncated at it;[1] we instead wrap Gemma prompts in the model's turn markers, since its base ships without a chat template (0.204 with markers vs. 0.046 without).

5 RESULTS

We report every strategy's 1B-token checkpoint on EDIT-Bench and LiveCodeBench in Tables 2 and 3 for all six base models, with and wihout the *strong* instructions, next to the untrained base and to a model instruction-tuned on the same instruction data with no code.

At equal cost, rewritten code beats raw source. On EDIT-Bench under *weak*, the raw file (RAW) declines with tokens on both selection models (Qwen3-8B: .389 $\rightarrow$ .241; Gemma4-12B: .204 $\rightarrow$ .167 across 250M/500M/1B), while every rewrite matches or recovers (7). Under *strong* the effect is starker: at 1B, the commit-conditioned rewrite OLD+TERSECM reaches the top strong score on Qwen3-4B (.657) and is top-ranked across all six strong columns (mean rank 1); its shorter-message sibling OLD+VERBOSECM is second (rank 2), and OLD+CM third (rank 3). LiveCodeBench inverts the ordering: the single-artifact strategies (RAW, CLEAN, BLOG) occupy the top three ranks. Strategies specialize to the shape of the benchmark — edit-shaped training data wins on edit tasks, completion-shaped training data wins on completion tasks — so no single strategy dominates both, and the choice of strategy should follow the deployment target.

[1] Gemma generations therefore run to EOS or the cap.

	Qwen3-4B		Qwen3-8B		Qwen3-14B		Gemma4-12B		Gemma4-26B		Gemma4-31B		rank
	weak	strong	weak	strong	weak	strong	weak	strong	weak	strong	weak	strong	
untrained base	.250		.389		.398		.204		.241		.407		
instruction tuned	.277	.537	.259	.583	.277	.583	.194	.472	.296	.556	.370	.519	
Single-artifact													
Raw	.194	.537	.241	.546	.250	.639	.167	.546	.278	.565	.250	.583	8
Clean	.250	.509	.250	.593	.231	.611	.222	.509	.287	.602	.333	.602	6
Nb	.222	.593	.306	.583	.231	.583	.204	.528	.250	.574	.269	.546	10
Blog	.259	.546	.306	.574	.269	.611	.241	.546	.324	.565	.361	.556	11
Multi-artifact													
OldCM	.287	.620	.278	.583	.407	.583	.250	.556	.306	.583	.333	.611	2
OldFormatCM	.306	.657	.343	.620	.278	.611	.176	.565	.259	.620	.287	.602	1
OldTerseCM	.269	.556	.278	.592	.361	.611	.167	.528	.333	.611	.250	.620	3
PlanOld	.194	.555	.278	.546	.259	.592	.139	.518	.278	.630	.370	.583	7
PlanCMOld	.259	.546	.269	.537	.250	.602	.157	.565	.269	.574	.380	.583	9
Plan	.250	.565	.278	.630	.315	.611	.213	.500	.324	.583	.231	.602	4
OldReview	.157	.574	.213	.564	.296	.620	.204	.556	.380	.583	.278	.565	5

Table 2: EDIT-Bench `pass_rate`: every strategy after training on 1B tokens, for 6 models. *weak* is the arm with QA instruction replay only; *strong* adds the leakage-filtered edit instructions at the same total budget. *instruction tuned* trains on standard QA instructions (weak) or edit instructions (strong) alone with no arm data. Blue: top value per group exceeding the weak baseline. Red: top two values per group exceeding the strong baseline. *rank*: integer rank across the six strong columns.



Figure 3: **What the model learns to fix beyond the instruction depends on the type of synthetic code it was trained on.** Every Gemma4-26B checkpoint trained with *strong* evaluated on EDIT-Bench #50 and grouped by what it changed; all code shown is verbatim, and chips report the problem's six unit tests re-run on it. The instruction asks for one thing, but the file has a second, unmentioned bug: `is_int` is defined outside `IntegerToken` while `__main__` calls it as a method. With *strong* alone, RAW, OLD+CM and PLAN+OLD do only what was asked and score 1 of 6; the remaining seven arms move `is_int` into the class and pass all six.

Plan-based strategies split by family. Under *strong*, PLAN (plan without source) is a strong multi-artifact choice on Qwen3 and mid-to-bottom on Gemma4, while PLAN+OLD (plan with source) shows the opposite direction. LiveCodeBench replicates this: PLAN places rank 3 over-

	Qwen3-4B		Qwen3-8B		Qwen3-14B		Gemma4-12B		Gemma4-26B		Gemma4-31B		rank
arm	weak	+strong	weak	+strong	weak	+strong	weak	+strong	weak	+strong	weak	+strong	
untrained base	.137		.160		.189		.119		.180		.209		
instruction tuned	.151	.128	.179	.163	.191	.179	.049	.005	.139	.139	.119	.113	
Single-artifact													
Raw	.137	.145	.163	.177	.186	.202	.104	.116	.179	.176	.160	.160	4
Clean	.127	.153	.185	.180	.174	.192	.096	.105	.163	.186	.169	.186	1
Nb	.148	.136	.163	.186	.180	.211	.093	.104	.160	.166	.144	.160	6
Blog	.128	.147	.154	.185	.191	.208	.104	.101	.177	.182	.185	.179	2
Multi-artifact													
OldCM	.131	.144	.156	.177	.189	.200	.098	.082	.177	.173	.163	.180	7
OldFormatCM	.133	.142	.153	.157	.174	.197	.096	.107	.176	.174	.174	.156	9
OldTerseCM	.130	.118	.159	.145	.186	—	.089	.067	.160	.195	.160	.157	11
PlanOld	.131	.134	.159	.162	.192	.195	.014	.101	.102	.171	.046	.157	10
PlanCMOld	.128	.127	.154	.169	.202	.203	.101	.095	.162	.182	.160	.157	8
Plan	.118	.160	.153	.182	.189	.194	.079	.093	.128	.176	.115	.174	3
OldReview	.119	.145	.159	.168	.180	.191	.095	.104	.169	.174	.159	.182	5

Table 3: LiveCodeBench pass@1 (131 problems spanning February-April 2025 window) at the 1B-token checkpoint. The untrained base row uses the few-shot prompt, since the bases never saw the wrapper. *instruction tuned* is the untrained base fine-tuned on each data source; it was not re-generated, and its values are the earlier run with the same truncation applied after the fact. Blue: top value per group exceeding the instruction-tuned weak baseline. Red: top value per group exceeding the instruction-tuned +strong baseline. *rank*: overall rank ordered by mean score across the six +strong columns; arms with a missing +strong cell are unranked.

all, driven by Qwen3-8B (.182, the group's top +strong value) and Qwen3-14B (.194), while PLAN+OLD is stronger on Gemma.

Commit-message conditioning transfers across families. On EDIT-Bench, the three commit-message-conditioned strategies take the top three ranks (OLD+TERSECM, OLD+VERBOSECM, OLD+CM), and each is strong on both families rather than driven by one. The synthetic-message variants (OLD+TERSECM, OLD+VERBOSECM) beat the human-message baseline (OLD+CM) on average, and OLD+TERSECM— the shape-constrained variant that produces messages 2.1× shorter wins outright. On LiveCodeBench these arms drop to the middle of the field, so the transfer is EDIT-Bench-specific: commit-message conditioning is the default choice for edit-shaped deployment.

6 CONCLUSION

We built SynthCode, a corpus of teacher-written artifacts distilled from Python commits using *gpt-oss-120b* and a token-matched protocol for asking what a code model should read during mid-training: eleven strategies, each trained for 1B tokens under one set of hyperparameters and into six base models from two families.

Three findings hold across the six models. Mid-training on raw commit files costs editing accuracy relative to instruction tuning alone, and transforming the code recovers most of that cost. Which artifact accompanies the code matters more than how the file is rewritten: once *strong* edit instructions are present, packing the pre-commit file with its commit message is the only group that still beats raw code and single-artifact rewrites, while plan-first packing adds nothing on EDIT-Bench and lowers LiveCodeBench without edit instructions. Mid-training strategies often lift LiveCodeBench over instruction tuning alone across all six models, especially benefiting from cleaned up files and blog posts.

LIMITATIONS

To gain a better understanding of the strategies that help pre- and mid-training regimes, scaling to more tokens and longer training runs is important. More importantly, it is difficult to mid-train models that already demonstrate strong capabilities. We list further limitations below:

Python only. Generalization to other languages and source distributions is untested. We use only the Python subset of the CommitPack dataset. Future work could extend our framework to other languages and benchmarks covering them. Furthermore, training data is derived from CommitPack, which is built from permissively licensed public repositories. Synthetic annotations were generated by a LLM and may contain errors; they are intended for training and are not presented as documentation of the underlying projects.

Data defects and filtering. The teacher's output is not clean by default. The filter removes 13.7% of cleaned files and 20.7% of notebooks (Appendix B). About 4.3% of rows are lost to a failed lookup of the pre-commit file, and 10.5% of notebooks stop at the generation limit, the largest single loss. Rarer defects are harder to see: the teacher occasionally returns an abridged file that parses and ends as if complete, and it rewrites the grant clause of copyleft license headers, which we restore. Because removals differ by transform, the kept source files are not identical across strategies: truncation removes the longest outputs, which likely skews the NOTEBOOK strategy toward shorter sources than the CLEAN strategy.

AI USE STATEMENT

In this work, we utilize an LLM to rewrite GitHub commit style data into synthetic data of various forms. We also utilize LLMs to prepare some of the tables and help in drafting an outline of our paper and proof-read sections of the paper.

REFERENCES

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. Can it edit? evaluating the ability of large language models to follow code editing instructions, 2024. URL https://arxiv.org/abs/2312.12450.

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. Edit-bench: Evaluating llm abilities to perform real-world instructed code edits, 2025. URL https://arxiv.org/abs/2511.04486.

Amir M. Ebrahimi and Gopi Krishnan Rajbahadur. Edit, but verify: An empirical audit of instructed code-editing benchmarks, 2026. URL https://arxiv.org/abs/2604.05100.

Kazuki Fujii, Yukito Tajima, Sakae Mizuki, Masaki Kawamura, Hinari Shimada, Taihei Shiotani, Koshiro Saito, Masanari Oi, Taishi Nakamura, Takumi Okamoto, Shigeki Ishida, Kakeru Hattori, Youmi Ma, Hiroya Takamura, Rio Yokota, Jun Sakuma, and Naoaki Okazaki. Rewriting pre-training data boosts llm performance in math and code, 2026. URL https://arxiv.org/abs/2505.02881.

GLM-4.5 Team, Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, Kedong Wang, Lucen Zhong, Mingdao Liu, Rui Lu, Shulin Cao, Xiaohan Zhang, Xuancheng Huang, Yao Wei, Yean Cheng, Yifan An, Yilin Niu, Yuanhao Wen, Yushi Bai, Zhengxiao Du, Zihan Wang, Zilin Zhu, Bohan Zhang, Bosi Wen, Bowen Wu, Bowen Xu, Can Huang, Casey Zhao, Changpeng Cai, Chao Yu, Chen Li, Chendi Ge, Chenghua Huang, Chenhui Zhang, Chenxi Xu, Chenzheng Zhu, Chuang Li, Congfeng Yin, Daoyan Lin, Dayong Yang, Dazhi Jiang, Ding Ai, Erle Zhu, Fei Wang, Gengzheng Pan, Guo Wang, Hailong Sun, Haitao Li, Haiyang Li, Haiyi Hu, Hanyu Zhang, Hao Peng, Hao Tai, Haoke Zhang, Haoran Wang, Haoyu Yang, He Liu, He Zhao, Hongwei Liu, Hongxi Yan, Huan Liu, Huilong Chen, Ji Li, Jiajing Zhao, Jiamin Ren, Jian Jiao, Jiani Zhao, Jianyang Yan, Jiaqi Wang, Jiayi Gui, Jiayue Zhao, Jie Liu, Jijie Li, Jing Li, Jing Lu, Jingsen Wang, Jingwei Yuan, Jingxuan Li, Jingzhao Du, Jinhua Du, Jinxin Liu, Junkai Zhi, Junli Gao, Ke Wang, Lekang Yang, Liang Xu, Lin Fan, Lindong Wu, Lintao Ding, Lu Wang, Man Zhang, Minghao Li, Minghuan Xu, Mingming Zhao, Mingshu Zhai, Pengfan Du, Qian Dong, Shangde Lei, Shangqing Tu, Shangtong Yang, Shaoyou Lu, Shijie Li, Shuang Li, Shuang-Li, Shuxun Yang, Sibo Yi, Tianshu Yu, Wei Tian, Weihan Wang, Wenbo Yu, Weng Lam Tam, Wenjie Liang, Wentao Liu, Xiao Wang, Xiaohan Jia, Xiaotao Gu, Xiaoying Ling, Xin Wang, Xing Fan, Xingru Pan, Xinyuan Zhang, Xinze Zhang, Xiuqing Fu, Xunkai Zhang, Yabo Xu, Yandong Wu, Yida Lu, Yidong Wang, Yilin Zhou, Yiming Pan, Ying Zhang, Yingli Wang, Yingru Li, Yinpei Su, Yipeng Geng, Yitong Zhu, Yongkun Yang, Yuhang Li, Yuhao Wu, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yuxuan Zhang, Zezhen Liu, Zhen Yang, Zhengda Zhou, Zhongpei Qiao, Zhuoer Feng, Zhuorui Liu, Zichen Zhang, Zihan Wang, Zijun Yao, Zikang Wang, Ziqiang Liu, Ziwei Chai, Zixuan Li, Zuodong Zhao, Wenguang Chen, Jidong Zhai, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models, 2025. URL https://arxiv.org/abs/2508.06471.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need, 2023. URL https://arxiv.org/abs/2306.11644.

Alexander Hägele, Elie Bakouch, Atli Kosson, Loubna Ben Allal, Leandro Von Werra, and Martin Jaggi. Scaling laws and compute-optimal training beyond fixed training durations, 2024. URL https://arxiv.org/abs/2405.18392.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Feiyang Kang, Newsha Ardalani, Michael Kuchnik, Youssef Emad, Mostafa Elhoushi, Shubhabrata Sengupta, Shang-Wen Li, Ramya Raghavendra, Ruoxi Jia, and Carole-Jean Wu. Demystifying synthetic data in llm pre-training: A systematic study of scaling laws, benefits, and pitfalls, 2025. URL https://arxiv.org/abs/2510.01631.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The Stack: 3 TB of permissively licensed source code, 2022. URL https://arxiv.org/abs/2211.15533.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, Kushal Arora, Saurabh Garg, Rui Xin, Niklas Muennighoff, Reinhard Heckel, Jean Mercat, Mayee Chen, Suchin Gururangan, Mitchell Wortsman, Alon Albalak, Yonatan Bitton, Marianna Nezhurina, Amro Abbas, Cheng-Yu Hsieh, Dhruba Ghosh, Josh Gardner, Maciej Kilian, Hanlin Zhang, Rulin Shao, Sarah Pratt, Sunny Sanyal, Gabriel Ilharco, Giannis Daras, Kalyani Marathe, Aaron Gokaslan, Jieyu Zhang, Khyathi Chandu, Thao Nguyen, Igor Vasiljevic, Sham Kakade, Shuran Song, Sujay Sanghavi, Fartash Faghri, Sewoong Oh, Luke Zettlemoyer, Kyle Lo, Alaaeldin El-Nouby, Hadi Pouransari, Alexander Toshev, Stephanie Wang, Dirk Groeneveld, Luca Soldaini, Pang Wei Koh, Jenia Jitsev, Thomas Kollar, Alexandros G. Dimakis, Yair Carmon, Achal Dave, Ludwig Schmidt, and Vaishaal Shankar. Datacomp-lm: In search of the next generation of training sets for language models, 2025. URL https://arxiv.org/abs/2406.11794.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder: may the source be with you!, 2023a. URL https://arxiv.org/abs/2305.06161.

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. Textbooks are all you need ii: phi-1.5 technical report, 2023b. URL https://arxiv.org/abs/2309.05463.

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, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder 2 and The Stack v2: The next generation, 2024. URL https://arxiv.org/abs/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, 2025. URL https://arxiv.org/abs/2306.08568.

Pratyush Maini, Skyler Seto, He Bai, David Grangier, Yizhe Zhang, and Navdeep Jaitly. Rephrasing the web: A recipe for compute and data-efficient language modeling, 2024. URL https://arxiv.org/abs/2401.16380.

Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, and Matthias Gallé. On leakage of code generation evaluation datasets. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 13215–13223, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.772. URL https://aclanthology.org/2024.findings-emnlp.772/.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models, 2024. URL https://arxiv.org/abs/2308.07124.

Niklas Muennighoff, Alexander M. Rush, Boaz Barak, Teven Le Scao, Aleksandra Piktus, Nouamane Tazi, Sampo Pyysalo, Thomas Wolf, and Colin Raffel. Scaling data-constrained language models, 2025. URL https://arxiv.org/abs/2305.16264.

Joel Niklaus, Atsuki Yamaguchi, Michal Štefánik, Guilherme Penedo, Hynek Kydlíček, Elie Bakouch, Lewis Tunstall, Edward Emanuel Beeching, Thibaud Frere, Colin Raffel, Leandro von Werra, and Thomas Wolf. How can we synthesize high-quality pretraining data? a systematic study of prompt design, generator model, and source data, 2026. URL https://arxiv.org/abs/2604.13977.

Martin Riddell, Ansong Ni, and Arman Cohan. Quantifying contamination in evaluating code generation capabilities of language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14116–14137, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.761. URL https://aclanthology.org/2024.acl-long.761/.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, M. C., Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y. Charles, H. S. Che, Guanduo Chen, Guangyu Chen, Guanzheng Chen, Huarong Chen, Jia Chen, Jianlong Chen, Jun Chen, Kexin Chen, Peng Chen, Ruijue Chen, Wentao Chen, Xin Chen, Yang Chen, Yanru Chen, Yifei Chen, Yingjiang Chen, Yuankun Chen, Yujie Chen, Yutian Chen, Zhirong Chen, Dazhi Cheng, Yean Cheng, Jialei Cui, Jingbing Cui, Anqi Dai, Jiaqi Deng, Hao Ding, Rui Ding, Shaofeng Ding, Mengfan Dong, Mengnan Dong, Yuhao Dong, Yuxin Dong, Angang Du, Chenzhuang Du, Dikang Du, Jusen Du, Yulun Du, Yu Fan, Jing Feng, Qiulin Feng, Yichen Feng, Kelin Fu, Qiang Fu, Fuxuan Gao, Hongcheng Gao, Jingyue Gao, Tong Gao, Weijia Gao, Shangyi Geng, Jie Gong, Linhu Gong, Shengao Gong, Xiaochen Gong, Qizheng Gu, Yicheng Gu, Shuhao Guan, Haiqing Guo, Shiqi Guo, Xiang Guo, Zhengyan Guo, Beixi Hao, Wenxin Hao, Xiaoru Hao, Dailan He, Haotian He, Lehan He, Qi He, Weiran He, Xinran He, Xinyi He, Yibo He, Yunjia He, Chao Hong, Tiange Hong, Hao Hu, Jiaxi Hu, Ruikun Hu, Weiming Hu, Yangyang Hu, Zhenxing Hu, Liang Hua, Jinbin Huang, Ke Huang, Ruiyuan Huang, Siying Huang, Weixiao Huang, Yan Huang, Zhengjie Huang, Zhiqi Huang, Yulong Hui, Chaobo Jia, Yutong Jiang, Zhejun Jiang, Zuoyou Jiang, Wenyi Jin, Xinyi Jin, Yu Jing, Huanjun Kong, Guokun Lai, Aidi Li, Cheng Li, Chengyuan Li, Cong Li, Fang Li, Guanyu Li, Haoyang Li, Jia Li, Junxiong Li, Lei Li, Letian Li, Lincan Li, Weihong Li, Wentao Li, Xintong Li, Yang Li, Yishen Li, Yiwei Li, Yuxiao Li, Zhaowei Li, Zhaoxi Li, Zheming Li, Zhengxiao Li, Zhiyuan Li, Jiawei Lin, Xiaohan Lin, Yibo Lin, Zichao Lin, Ziyan Lin, Bill Liu, Boxiao Liu, Chuan Liu, Liang Liu, Shaowei Liu, Shudong Liu, Shuran Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yanming Liu, Yibo Liu, Yipeng Liu, Zhengying Liu, Zhiheng Liu, Enzhe Lu, Haoyu Lu, Linqiang Lu, Tingzhan Lu, Zhiyuan Lu, Aotian Luo, G. Luo, Junyu Luo, Yifan Luo, B. Lyu, Wenzhou Lyu, Shaoguang Mao, Yuan Mei, Xin Men, Minqing Ni, Yixuan Niu, Siyuan Pan, Shujun Peng, Zhangyang Qi, Ruoyu Qin, ZeChao Qin, Zeyu Qin, Haiquan Qiu, Jianxin Qiu, Jiezhong Qiu, Bowen Qu, Yuhao Qu, Zeyu Shang, Youbo Shao, Han Shen, Jincheng Shi, Juanfeng Shi, Lidong Shi, Shengyuan Shi, Wingchun Siu, Pengwei Song, Xiaoxi Song, Jianlin Su, Yunfeng Su, Zhaochen Su, Lin Sui, Jingsong Sun, Junyao Sun, Shaoning Sun, Shuzhe Sun, Tongyu Sun, Yujun Sun, Yunpeng Tai, Chuning Tang, Heyi Tang, Sirui Tang, Zecheng Tang, Chaoran Tian,

Rongpeng Tian, Yu Tian, Wei Tu, Chensi Wang, Chuang Wang, Chunjie Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Hao Wang, Hao Wang, Huaqing Wang, Hui Wang, Jiayi Wang, Jinglong Wang, Jinhong Wang, Jiuzheng Wang, Linian Wang, Shaobo Wang, Shenzhi Wang, Shuyi Wang, Si Wang, Siyuan Wang, Tianfu Wang, Wenjue Wang, Xingran Wang, Xinmei Wang, Xinyuan Wang, Xusheng Wang, Yalin Wang, Yangkun Wang, Yao Wang, Yaoyu Wang, Yejie Wang, Yiqin Wang, Yucheng Wang, Yuzhi Wang, Zhaoji Wang, Zhaowei Wang, Zhengtao Wang, Zhenhao Wang, Zhongsheng Wang, Zifan Wang, Chu Wei, Ming Wei, Shouxin Wei, Zichen Wen, Fan Wu, Haoning Wu, Rucong Wu, Wenhao Wu, Xiaoxue Wu, Yingcong Wu, Yongqi Wu, Yuxin Wu, Zijian Wu, Xinglang Xian, Chenxuan Xiang, Yuye Xiang, Bocheng Xiao, Chenjun Xiao, Xin Xiao, Jin Xie, Xiaotong Xie, Yifeng Xie, Zhe Xie, Bowei Xing, Yiming Xiong, Baosheng Xu, Boyu Xu, Jiale Xu, Jianfan Xu, Jing Xu, Jinjing Xu, L. H. Xu, Qingtao Xu, Shuyao Xu, Suting Xu, Tiantian Xu, Tianxiang Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ye Xu, Yueni Xu, Ziyao Xu, Haonan Xue, Junjie Yan, Yaoyao Yan, Fan Yang, Guangyao Yang, Hao Yang, Junwei Yang, Ruoyu Yang, Wenjie Yang, Xiaofei Yang, Xinyu Yang, Yi Yang, Yiling Yang, Ying Yang, Yuchen Yang, Zhen Yang, Zhilin Yang, Zian Yang, Zuhao Yang, Haotian Yao, Dan Ye, Haoran Ye, Wenjie Ye, Zhanbo Ye, Bohong Yin, Haoxiang Yin, Xietong Yin, Chengzhen Yu, Haozhen Yu, Longhui Yu, Shengnan Yu, Shuying Yu, Tianxiang Yu, Enming Yuan, Mengjie Yuan, Tongtian Yue, Wei Yue, Yang Yue, Dunyuan Zha, Haobing Zhan, B. H. Zhang, Dehao Zhang, Fei Zhang, Hao Zhang, Haoyuan Zhang, Huanyu Zhang, Jiapei Zhang, Jiaxuan Zhang, Jin Zhang, Kaiyi Zhang, Miaozhen Zhang, Puqi Zhang, Qinglei Zhang, Rong Zhang, Rui Zhang, Shaoshuai Zhang, Shiyi Zhang, Xiaobin Zhang, Xiaoyun Zhang, Y. Zhang, Yangkun Zhang, Ye Zhang, Yichi Zhang, Yikun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Zijing Zhang, Bin Zhao, Chenguang Zhao, Feifan Zhao, Jinglun Zhao, Jinxiang Zhao, Shuai Zhao, Wenshuo Zhao, Xiangyu Zhao, Xuanle Zhao, Yikai Zhao, Zijia Zhao, Haozhi Zheng, Huabin Zheng, Ruihan Zheng, Shaojie Zheng, Tengyang Zheng, Haofeng Zhong, Lei Zhong, Longguang Zhong, M. Zhou, Qiankang Zhou, Runjie Zhou, Ruozhang Zhou, Xinyu Zhou, Yiqiao Zhou, Zaida Zhou, Jinguo Zhu, Liya Zhu, Xinhao Zhu, Yangjunfeng Zhu, Yuxuan Zhu, Zhen Zhu, Chen Zhuang, Weiyu Zhuang, and Xinxing Zu. Kimi k3: Open frontier intelligence, 2026. URL https://arxiv.org/abs/2607.24653.

Abdul Waheed, Zhen Wu, Carolyn Rosé, and Daphne Ippolito. On code-induced reasoning in llms, 2025. URL https://arxiv.org/abs/2509.21499.

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, 2023. URL https://arxiv.org/abs/2212.10560.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct, 2024. URL https://arxiv.org/abs/2312.02120.

Sang Michael Xie, Hieu Pham, Xuanyi Dong, Nan Du, Hanxiao Liu, Yifeng Lu, Percy Liang, Quoc V. Le, Tengyu Ma, and Adams Wei Yu. Doremi: Optimizing data mixtures speeds up language model pretraining, 2023. URL https://arxiv.org/abs/2305.10429.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Yu Zhao, Yuanbin Qu, Konrad Staniszewski, Szymon Tworkowski, Wei Liu, Piotr Miłoś, Yuxiang Wu, and Pasquale Minervini. Analysing the impact of sequence composition on language model pre-training. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7897–7912. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.427. URL http://dx.doi.org/10.18653/v1/2024.acl-long.427.

A INSTRUCTIONS AND EXAMPLES

One subsection per transform, each following one CommitPack row from source to SynthCode document: the CommitPack file or commit, the instruction prompt we sent to `gpt-oss-120b` (verbatim), and the resulting SynthCode training document. Diffs, colour, and abbreviated files are ours; the model always received complete, unmarked files. RAW and OLD+CM use no prompt: they are the CommitPack file and its human commit message as shipped.

A.1 CLEANED FILE

Cleaned file

CommitPack file

```
from __future__ import print_function
import sys
from coverage import coverage


def missing_lines(cov_file, cov_config, fname):
  cov = coverage(data_file=cov_file, config_file=cov_config)
  cov.load()
  if cov.omit_match and cov.omit_match.match(fname):
    sys.stderr.write('Omitted file.\n')
    sys.exit(1)

  missing_lines = cov.analysis(fname)[2]
  print(*missing_lines, sep='\n', end='')

if __name__ == '__main__':
  missing_lines(*sys.argv[1:])
```

Instruction prompt

You are a senior software engineer. Your task is to cleanup the code in the document. Make sure to include comments to explain the code and add descriptive headers to the code blocks to make it easier to understand. Output only the cleaned code, nothing else. Do not use markdown code fences. Preserve every function, class, and import exactly as written: do not add, remove, rename, or restructure any of them, and do not change any logic. If the document begins with a copyright or licence header, reproduce it verbatim at the top of your output.

SynthCode cleaned file

```
# Script to print missing coverage lines for a given file using
  coverage.py

# ------------------------------------------------------------
# Imports
# ------------------------------------------------------------
from __future__ import print_function
import sys
from coverage import coverage
# ------------------------------------------------------------
# Function: missing_lines
# ------------------------------------------------------------
def missing_lines(cov_file, cov_config, fname):
```

```
    """
    Print missing line numbers for the specified source file.

    Args:
        cov_file (str): Path to the coverage data file.
        cov_config (str): Path to the .coveragerc configuration
    file.
        fname (str): Path to the source file to analyse.
    """
    cov = coverage(data_file=cov_file, config_file=cov_config)
    cov.load()
    if cov.omit_match and cov.omit_match.match(fname):
      sys.stderr.write('Omitted file.\n')
      sys.exit(1)

    missing_lines = cov.analysis(fname)[2]
    print(*missing_lines, sep='\n', end='')


# ------------------------------------------------------------
# Entry point
# ------------------------------------------------------------
if __name__ == '__main__':
    # Pass command-line arguments: coverage data file, config, and
    target file.
    missing_lines(*sys.argv[1:])
```

Figure 4: Cleaned file. The teacher adds comments and docstrings to a file without changing its code. Yellow: lines the teacher wrote; white: lines unchanged from CommitPack. Table 1 measures this in aggregate.

A.2 NOTEBOOK

Notebook

CommitPack file

```
# flake8: noqa

from .utils import (HARD_DELETE, SOFT_DELETE, HARD_DELETE_NOCASCADE,
                    DELETED_INVISIBLE, DELETED_VISIBLE_BY_PK)
from .models import safedelete_mixin_factory
from .managers import safedelete_manager_factory


__all__ = ['safedelete_manager_factory',
           'safedelete_mixin_factory',
           'HARD_DELETE', 'SOFT_DELETE',
           'HARD_DELETE_NOCASCADE',
           'DELETED_INVISIBLE',
           'DELETED_VISIBLE_BY_PK']
```

Instruction prompt

You are a software engineer. Convert the code in the document into a Jupyter notebook with interleaved markdown and code cells, where the markdown explains what each code cell does. Output a SINGLE valid .ipynb JSON document and NOTHING else: no prose, no markdown fences. It must be a JSON object with keys "cells", "metadata", "nbformat" (4), and "nbformat_minor" (5). Each cell needs "cell_type" ("markdown" or "code"), "metadata": {}, and "source" (a list of strings); code cells also need "execution_count": null and "outputs": []. If the document begins with a copyright or licence header, reproduce it verbatim in the first cell. Convert the entire document: do not omit, abbreviate, or summarise any part of it, and never write that code has been omitted for brevity.

SynthCode notebook

```
# Safe Delete Module

This notebook imports symbols needed for safe delete
  functionality.
```

```
# flake8: noqa

from .utils import (HARD_DELETE, SOFT_DELETE,
  HARD_DELETE_NOCASCADE,
                    DELETED_INVISIBLE, DELETED_VISIBLE_BY_PK)
from .models import safedelete_mixin_factory
from .managers import safedelete_manager_factory
```

```
Define the public API of this module via `__all__`.
```

```
__all__ = ['safedelete_manager_factory',
           'safedelete_mixin_factory',
           'HARD_DELETE', 'SOFT_DELETE',
           'HARD_DELETE_NOCASCADE',
           'DELETED_INVISIBLE',
           'DELETED_VISIBLE_BY_PK']
```

Figure 5: Notebook. The teacher turns a file into a notebook that explains each part. Yellow: markdown cells the teacher wrote; white: code cells, unchanged from CommitPack. The notebook is stored as `.ipynb` JSON and shown here as its cells.

A.3 PLAN

Plan

CommitPack commit

```
   #
   # Copyright (C) 2013 - 2015 Satoru SATOH <ssato @ redhat.com>
   # License: MIT
   #
   """anyconfig globals.
   """
+ import anyconfig.init
+
+
   AUTHOR = "Satoru SATOH <ssat@redhat.com>"
   VERSION = "0.0.7"
- import anyconfig.init
   LOGGER = anyconfig.init.LOGGER

   # vim:sw=4:ts=4:et:
```

Instruction prompt

You are a program manager. Your task is to create a planning document for the code in the document. Output only the planning document, nothing else.

SynthCode plan

```
   # Planning Document - `anyconfig.globals` Module

   **Prepared by:** Program Management Office
   **Date:** 2026-05-14

   ---

   ## 1. Overview
   The `anyconfig.globals` module provides a single source of truth
     for package-wide constants such as author information, version,
     and a pre-configured logger. It imports the logger from `
     anyconfig.init` and exposes it as `LOGGER`. This module is used
     throughout the `anyconfig` code-base for consistent metadata and
      logging.

   ---

   ## 2. Objectives
   | # | Objective | Success Metric |
   |---|-----------|----------------|
   | 1 | Ensure the module follows current Python packaging & style
     standards (PEP-8, PEP-517). | Lint passes with `flake8` (max-
     line-length 88) and `black` formatting. |
   | 2 | Centralise version handling using `importlib.metadata` to
     avoid manual updates. | Version string derived from `pyproject.
     toml` and matches `git` tag. |

   [... output continues ...]
   [the pre-commit file, 13 lines]
   [the post-commit file, 15 lines]
```

Figure 6: Plan. The teacher writes a planning document from the post-commit file. Yellow: text the teacher wrote; white: code it quoted unchanged from CommitPack. The plan (excerpt) comes before the two files in PLAN+OLD; PLAN+CM+OLD adds `INSTRUCTIONS:` and the human commit message after the plan, and PLAN leaves out the pre-commit file.

A.4 BLOG POST

Blog post

CommitPack file

```
from .labels import LabelsPlugin
from electrum_ltc.plugins import hook

class Plugin(LabelsPlugin):

    @hook
    def load_wallet(self, wallet, window):
        self.start_wallet(wallet)

    def on_pulled(self, wallet):
        self.print_error('labels pulled from server')
```

Instruction prompt

Your task is to write a blog post about the code in the document. The blog post should be written in a way that is easy to understand and follow. Output only the blog post, nothing else.

SynthCode blog post

````
## Decoding a Simple Electrum-LTC Plugin: A Walk-through

If you've ever dabbled with Bitcoin-derived wallets, you know
  that **Electrum** (and its Litecoin sibling, **Electrum-LTC**)
  are built to be extensible. Developers can drop in tiny "plugins
  " that hook into the wallet's lifecycle and add extra behaviour-
  think automatic label syncing, custom UI tweaks, or background
  housekeeping.

Below is a minimal example of such a plugin. Even though it's
  only a handful of lines, it showcases the core patterns you'll
  see in many Electrum extensions. Let's dissect it step-by-step
  and explain what each piece does, why it matters, and how you
  could expand it.

---

### The Full Listing

```python
from .labels import LabelsPlugin
from electrum_ltc.plugins import hook

class Plugin(LabelsPlugin):

    @hook
    def load_wallet(self, wallet, window):
        self.start_wallet(wallet)

    def on_pulled(self, wallet):
[... output continues ...]
````

Figure 7: Blog post. The teacher writes a blog post explaining a file. Yellow: text the teacher wrote; white: code it quoted unchanged from CommitPack. Excerpt; the post is the whole training document.
````

A.5 VERBOSE COMMIT MESSAGE

Verbose commit message

CommitPack commit

```
   from setuptools import setup

   setup(
       name='hyperchamber',
       version='0.2',
       description='Tune and optimize your hyperparameters',
       author='Martyn Garcia',
       author_email="martyn@255bits.com",
       packages=['hyperchamber'],
       install_requires=['requests >= 2.4.2'],
       url='https://github.com/255bits/hyperchamber',
       license='MIT',
-      classifiers=['License :: MIT',
-                   'Programming Language :: Python :: 3'],
+      classifiers=[
+                   ],
   )
```

Human commit message: [fix] add classifiers

Instruction prompt

You are an expert software engineer writing concise and informative git commit messages. Given only the old and new file contents, write a high-quality commit message that captures intent.

SynthCode commit (verbose message)

```
   [the pre-commit file, 15 lines]
   INSTRUCTIONS:
   **Strip classifiers from setup configuration**
   The `classifiers` entry in `setup()` has been cleared, removing
     the
   previous metadata (`License :: MIT` and `Programming Language ::
     Python :: 3`).
   This simplifies the package metadata by eliminating now-
     unnecessary
   classifiers.
   [the post-commit file, 15 lines]
```

Figure 8: Verbose commit message. The teacher writes a new commit message for a real commit. The human message says the classifiers were added; the diff empties the list, and the generated message says so. This unconstrained prompt yields markdown and several lines. Red and green: the commit, as a diff. Yellow: the message the teacher wrote, which takes the place of the human one between the two files.

A.6 TERSE COMMIT MESSAGE

Terse commit message

CommitPack commit

```
   import pytest
+ from poliastro.bodies import Sun
   from poliastro.twobody.mean_elements import get_mean_elements


   def test_get_mean_elements_raises_error_if_invalid_body():
-      body = "PlanetNine"
+      body = Sun

       with pytest.raises(ValueError) as excinfo:
           get_mean_elements(body)
       assert f"The input body '{body}' is invalid." in excinfo.
     exconly()
```

```
Human commit message:  Add test for error check
```

Instruction prompt

You are an expert software engineer writing a git commit message from a diff. Given only the old and new file contents, write a concise, high-quality commit message. Format: a single imperative subject line of at most 72 characters, then a blank line, then one to three short bullet lines ('- ...') explaining what changed and why. Write only as many bullets as the diff actually warrants -- if only one thing changed, write one bullet. Do not pad the list. Plain text only — no markdown, no bold, no backticks. Output only the commit message.

SynthCode commit (terse message)

```
   [the pre-commit file, 11 lines]
   INSTRUCTIONS:
   Use Sun constant in invalid-body test for get_mean_elements
   - Replace the string "PlanetNine" with the Sun constant and
     import Sun in the test.
   [the post-commit file, 12 lines]
```

Figure 9: Terse commit message. The teacher writes a new commit message for a real commit. The human message (“Add test for error check”) misdescribes the change: no test was added. Red and green: the commit, as a diff. Yellow: the message the teacher wrote, which takes the place of the human one between the two files.

Terse commit message

CommitPack commit

```
  # Copyright 2016 John Reese
  # Licensed under the MIT license
  # flake8: noqa
+ from concurrent.futures import CancelledError
+
  from .tasks import Task, PeriodicTask, TimerTask, QueueTask
  from .config import Config, JsonConfig
+ from .stats import Stats
  from .loop import Tasky

  __version__ = '0.6.2'
```

```
Human commit message:  Improve tasky imports
```

Instruction prompt

You are an expert software engineer writing a git commit message from a diff. Given only the old and new file contents, write a concise, high-quality commit message. Format: a single imperative subject line of at most 72 characters, then a blank line, then one to three short bullet lines ('- ...') explaining what changed and why. Write only as many bullets as the diff actually warrants -- if only one thing changed, write one bullet. Do not pad the list. Plain text only — no markdown, no bold, no backticks. Output only the commit message.

SynthCode commit (terse message)

```
  [the pre-commit file, 9 lines]
  INSTRUCTIONS:
  Add CancelledError and Stats imports to top-level package
  - Import CancelledError from concurrent.futures for task
    cancellation handling
  - Import Stats class to expose the statistics API at the package
    level
  [the post-commit file, 12 lines]
```

Figure 10: Terse commit message. The teacher writes a new commit message for a real commit. The human message ("Improve tasky imports") names nothing; the generated message names both imports and what each is for. Red and green: the commit, as a diff. Yellow: the message the teacher wrote, which takes the place of the human one between the two files.

A.7 CODE REVIEW

Code review

CommitPack commit

```
  Package ('xamarin-gtk-theme', 'master',
        sources = [ 'git://github.com/mono/xamarin-gtk-theme.git'
    ],
        override_properties = { 'configure':
                './autogen.sh --prefix=%{prefix}'
        },
-       revision = '8840eb554ce193d8660aeb3ae4957f49904edaae',
+       revision = '3ccf8317b6b1503c584584064e39f4643fb90069',
  )
```

Instruction prompt

You are a senior software engineer performing a code review of the file below. Write a specific, actionable review that covers: correctness and likely bugs, error handling and edge cases, readability and naming, structure/design, and security or performance concerns. Reference concrete functions, classes, or line-level details from the file — do not be generic. Do not cite line numbers; the file is given to you without them. Use plain prose with short section headings; do not restate the whole file. Output only the review.

SynthCode code review

```
[the pre-commit file, 7 lines]
```

```
**Correctness & Likely Bugs**
- The `sources` entry uses the `git://` protocol, which is
  unauthenticated and can be intercepted. If the repository is
  moved or the URL changes, the fetch will silently fail. Replace
  it with `https://github.com/mono/xamarin-gtk-theme.git` (or a
  signed tarball) and verify the checksum.
- The `override_properties` dictionary overrides the `configure`
  step, but there is no explicit `build`, `install`, or `clean`
  step defined. If the underlying build system expects separate `
  make` and `make install` phases, the package may break at build
  time. Add at least a `build: make` and `install: make install
  DESTDIR=%{buildroot}` entry, or confirm that the build system
  handles everything in the overridden configure script.
- The version string is set to `'master'`. This points to a
  mutable branch; each build could produce a different binary,
  violating reproducibility. Consider pinning to a tagged release
  (e.g., `1.0.2`) and updating `revision` accordingly.

**Error Handling & Edge Cases**
- No integrity verification (hash or signature) is provided for
  the source. Without a `sha256` (or similar) field, a corrupted
  or malicious clone will go unnoticed. Add a checksum field that
  is regenerated when the source URL changes.
- The `configure` command interpolates `%{prefix}`. If the package
   manager does not replace this placeholder correctly, the
  configure script will receive a literal `%{prefix}`, causing an
  installation path error. Validate that the placeholder syntax
  matches the manager's expectations or use the documented
  variable name (e.g., `${PREFIX}`).

[... output continues ...]
```

```
[the post-commit file, 7 lines]
```

Figure 11: Code review. The teacher reviews the post-commit file. Yellow: text the teacher wrote; white: code it quoted unchanged from CommitPack. The review (excerpt) sits between the two files.

B FILTER RULES AND DROP RATES

Table 4 lists every rule that removed rows from the CLEAN and NOTEBOOK data, as a percentage of generated rows. *Input* rules remove rows whose source was unusable; they are nearly identical across the two transforms because both read the same source rows. *Output* rules remove generation defects.

Table 4: Filter drop rates (% of generated rows). A dash means the rule does not apply.

	Rule	CLEAN	NOTEBOOK
Input	pre-commit file lookup failed	4.34	4.27
	source under 200 characters	2.11	2.16
	version-bump-only commit	1.72	1.72
	source cut by the input cap	0.02	0.07
Output	truncated at the token limit	4.21	10.53
	invalid notebook JSON	–	1.73
	syntax regression	0.68	–
	argument renamed / silent bug fix	0.38	–
	abridged ("omitted for brevity")	0.17	–
	kept	**86.3**	**79.3**

Two rules narrowed after measurement. *Truncation* is detected only from the decoder (`finish_reason = length`). An earlier character cap also removed long outputs, but those were valid: 92.5% of the cleaned files and 95.6% of the notebooks it removed parsed, against 97.9% and 98.2% of the rows kept. *Syntax* is gated conditionally: 21–26% of sources are Python 2, so a cleaned file is dropped only when its source parses as Python 3 and the output does not.

What the defects are. A non-zero return code from the pre-commit lookup predicted an empty x_{old} in all 683 rows checked; a transform reading both sides would describe a change that never happened, so these rows are removed everywhere. *Abridgement* is caught by the output-to-source character ratio and by marker phrases; a definition-count ratio would also remove legitimate consolidation, such as folding 39 near-identical wrappers into a single factory. Repairs, applied before the drop rules, strip the teacher's reasoning trace, fold injected typographic characters absent from the source back to ASCII, restore notebook cell structure (8.0% of notebooks), and restore rewritten copyleft grant clauses (1.4% of cleaned files, 2.2% of notebooks).

C STRATEGY EXAMPLES

Appendix A follows one CommitPack row per transform, from source to SynthCode document.
We also show some examples of the EditBench evaluation of one model per family, with every return of one model grouped by what it did to the file in Figure 14. With *strong* data every strategy returns the whole file; five implement the feature correctly and the other five attach `samples_per_candle` to the wrong class. The fragment failure is systematic rather than anecdotal: over all six models and 108 problems, 37% of returns without *strong* data are shorter than half the original file against 3% with it, and returning the whole file is the single largest thing the 7% *strong* data teaches.

D HYPERPARAMETERS

Table 5 lists the settings. Two of them were chosen against what a default would give, and each corrected a measured failure.
Warmup is an absolute token count, not a ratio. A warmup ratio makes the early milestones depend on the total budget: at 0.58B a 3% ratio is 178 steps, at 1B it is 305, so the 250M checkpoint of a 1B run has nearly twice the warmup baked in as the 250M checkpoint of a shorter run and the two are not comparable. We therefore fix warmup at 17.5M tokens (178 steps) regardless of budget.

Gradient clipping is per family. Gemma4's natural gradient norms on this data run 100–1,700 (measured 300–16,550 on the same batches where Qwen3-8B measures 1.07–1.75). Clipping Gemma at the conventional 1.0 rescales a norm-400 step by 0.0025, so the clip rather than the learning

setting	value
objective	full-parameter causal LM, empty prompt, loss on every document token
budget	1,000,219,392 tokens = 10,176 × 98,292 (12 ranks × 1 seq × 8,191)
sequence	8,192 cutoff, packed, per-document attention mask, EOS-terminated
optimizer	AdamW, $\beta_1 = 0.9$, $\beta_2 = 0.95$, weight decay 0.1, bf16
learning rate	2×10^{-5} peak; warmup 178 steps (17.5M tokens) then constant
anneal	branch at milestone; linear decay to 1×10^{-7} over 10% of tokens seen
grad clip	1.0 (Qwen3), 200 (Gemma4)
instruction replay	10.0% of tokens, all arms; +6.4% / 7.4% edit instructions in treatment
milestones	250M, 500M, 1B
parallelism	DeepSpeed ZeRO-2 ($\leq$ 8B Qwen) or ZeRO-3; gradient checkpointing; SDPA attention
hardware	Intel Max-series GPUs, 12 ranks per node; 1 / 4 / 8 nodes per arm by model size

Table 5: Training configuration, identical across arms. No tuning is done per arm. Gradient clipping is per family.

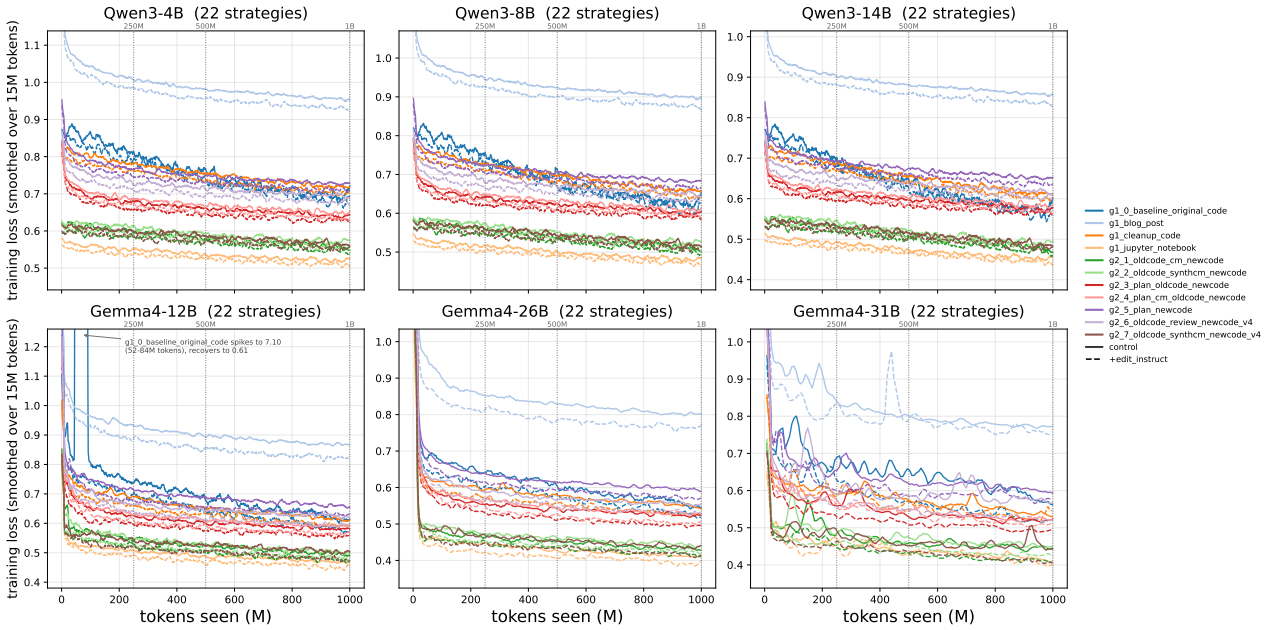

Figure 12: Training loss over the first 1B tokens for all six models. Each panel shows the eleven strategies trained with *weak* instruction only (solid) and with the leakage-filtered edit instructions added (dashed). Curves are smoothed over a 15M-token window and show the constant learning-rate run; the 1B checkpoints we evaluate come from a separate short anneal that is not shown. Loss is not comparable across strategies, since each trains on a different corpus.

rate governs every update. Gemma is clipped at 200, Qwen at 1.0; the threshold is constant across strategies within a family.

E Additional Results

Table 7 reports EDIT-Bench pass rate for every strategy at three annealed milestones on Qwen3-8B and Gemma4-12B. Three patterns stand out. First, under *weak* instructions alone, no strategy reliably exceeds its untrained base at 1B: the best Qwen3-8B strategy reaches .343 against a base of .389, and the best Gemma4-12B strategy reaches .250 against .204. More tokens sometimes hurt — Gemma4-12B under PLAN+CM+OLD peaks at .287 (500M) and falls to .157 (1B) — indicating that continued pretraining on the transformed corpus alone drifts models away from the edit-benchmark distribution. Adding the *strong*-instruction set stabilizes the milestone trajectory and lifts every strategy well above both bases: at 1B, all Qwen3-8B *strong* strategies fall in [.537, .630] against

.389, and all Gemma4-12B *strong* strategies in [.500, .565] against .204. Notably, several *strong* strategies match or exceed the full-dataset instruction-tuning baseline (.583 on Qwen3-8B, .472 on Gemma4-12B) despite using a leakage-filtered subset. Third, commit-message conditioning is the most consistent single choice: OLD+CM and OLD+TERSECM are among the top strategies on both models under both conditions, with OLD+TERSECM achieving the best Qwen3-8B *weak* score at 1B (.343) and PLAN+CM+OLD tying for the best Gemma4-12B *strong* score (.565).
Table 8 splits the 1B checkpoints by estimated difficulty. The *strong* gains hold across tiers: on hard problems, strategies such as OLD+TERSECM on Qwen3-4B (.413) and PLAN+CM+OLD on Gemma4-12B (.391) exceed the *strong*-only baseline in their column. On easy problems the picture is similar, with OLD+TERSECM and OLD+REVIEW the most frequent leaders. The *weak* strategies rarely clear the *weak*-only reference in either tier, consistent with the milestone-level finding.

We also ask whether mixing the different data sources lead to any benefits. We run the following mixing experiments for Qwen3-8b and Gemma4-12B (Table 7).

1. Mix Single-artifacts: Mixing CLEAN, NOTEBOOK, and BLOG;
2. Mix Multi-artifacts: OLD+CM, PLAN+OLD, PLAN+CM+OLD, PLAN, REVIEW, VERBOSECM;
3. Mix Cross: CLEAN, NOTEBOOK, PLAN+CM+OLD;
4. Mix all 9: the three single-artifacts plus the six multi-artifacts.

Interestingly, across all *strong* columns, MIX-G2 achieves a mean score of .556, which would rank 3rd overall against individual arms, outperforming all single-artifact strategies.

Is a second view of a commit worth a new commit? Synthetic pipelines can spend a fixed token budget in two ways: generate views of more source commits, or generate more views of the same commits. We compare three recipes trained to the same number of tokens (Table 6). DISJOINT gives $2N$ commits one synthetic view each. SHARED gives N commits two different views each. REPEAT gives N_c commits the same view twice, with N_c chosen so the token count matches, since view types differ in length. We train each recipe into Qwen3-8B and Gemma4-12B for approximately .5B tokens. We score the final checkpoints on two instruments: EDIT-Bench `pass_rate` and bits-per-byte (BPB) on 900 decontaminated held-out CommitPack documents. EDIT-Bench cannot rank the two non-repeating recipes. SHARED and DISJOINT differ by a single question out of 108 on Qwen (.009) and tie exactly on Gemma. Both gaps are well below the benchmark's 0.03 noise floor. BPB gives the same order in both families. Relative to a fresh commit, a second view of the same commit costs $+0.0025$ (Qwen) and $+0.0022$ (Gemma) bits, and repeating the same view costs $+0.0109$ and $+0.0085$. A second, distinct view therefore keeps about three quarters of a new commit's value (77% and 74%), while plain repetition loses most of it. In practice, when source data runs short, a new transformation of an existing commit is a much better use of the budget than a second pass over existing text, and only slightly worse than a new commit.

	EDIT-Bench `pass_rate` ↑		held-out BPB ↓	
recipe	Qwen3-8B	Gemma4-12B	Qwen3-8B	Gemma4-12B
DISJOINT	0.2870	0.1667	0.3254	0.3777
SHARED	0.2778	0.1667	0.3279	0.3799
REPEAT	0.2500	0.1574	0.3363	0.3862
spread across recipes	0.0370	0.0093	0.0108	0.0085
ranking (best first)	D–S–R	D=S–R	D–S–R	D–S–R

Table 6: **A second view of a commit is worth about three quarters of a first view of a new one.** At a fixed token budget, DISJOINT covers $2N$ commits with one synthetic view each, SHARED N commits with two different views each, and REPEAT N_c commits with the same view twice (N_c token-matched, as the view types differ in length). Both instruments score the same `final` checkpoints. EDIT-Bench cannot separate SHARED from DISJOINT (.009 apart on Qwen, exactly tied on Gemma, against a 0.03 noise floor). BPB over 900 decontaminated held-out CommitPack documents orders all three identically in both families: a second view costs $+.0025$ / $+.0022$ bits against a fresh commit and repetition costs $+.0109$ / $+.0085$, so a second view retains 77% / 74% of a new commit's value.

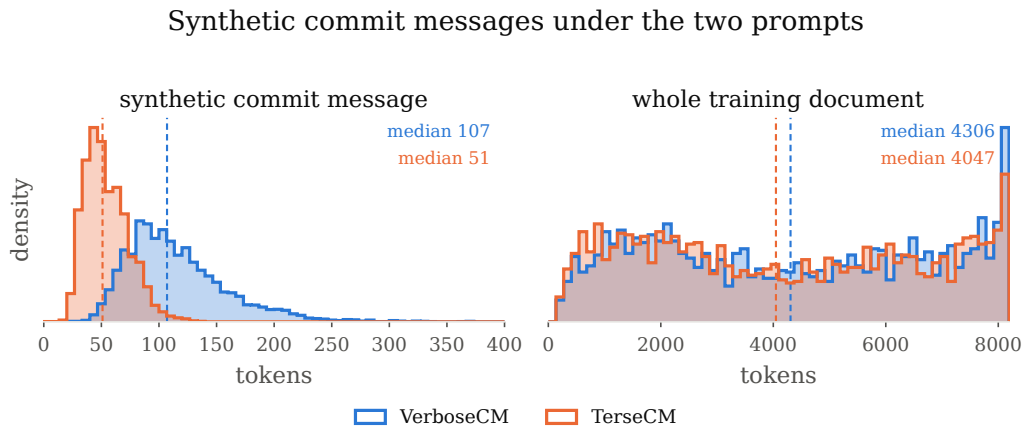

Figure 13: Length of the two synthetic commit-message rewrites, VERBOSECM and TERSECM. *Left:* the isolated generations, where VERBOSECM produces more than twice as many tokens as TERSECM. *Right:* the full training documents into which each is packed; the difference shrinks but VERBOSECM retains a longer tail.

	Qwen3-8B						Gemma4-12B						rank
	weak			strong			weak			strong			
arm	250M	500M	1B	250M	500M	1B	250M	500M	1B	250M	500M	1B	
untrained base			.389						.204				
instruction tuned		.259			.583			.194			.472		
single artifact													
RAW	.287	.259	.241	.565	.528	.546	.231	.194	.167	.500	.519	.546	8
CLEAN	.269	.222	.250	.565	.565	.593	.194	.185	.222	.519	.491	.509	2
NOTEBOOK	.259	.250	.306	.574	.593	.583	.204	.241	.204	.472	.472	.528	9
BLOG	.389	.269	.306	.519	.528	.574	.241	.213	.241	.500	.556	.546	3
multi-artifact packing													
OLD+CM	.380	.333	.278	.593	.583	.583	.231	.213	.250	.463	.500	.556	5
OLD+TERSECM	.361	.380	.343	.574	.611	.620	.194	.185	.176	.509	.491	.565	1
PLAN+OLD	.315	.296	.278	.546	.611	.546	.148	.185	.139	.500	.546	.518	6
PLAN+CM+OLD	.370	.306	.269	.574	.537	.537	.231	.287	.157	.546	.537	.565	4
PLAN	.306	.250	.278	.583	.546	.630	.213	.194	.213	.509	.463	.500	7
OLD+REVIEW	.296	.241	.213	.537	.519	.564	.222	.185	.204	.435	.528	.556	10
equal-token mixtures													
MIX-G1	.250	.269	.287	.519	.528	.593	.250	.185	.204	.472	.519	.528	—
MIX-G2	.315	.269	.333	.528	.602	.602	.241	.194	.176	.472	.565	.565	—
MIX-CROSS	.324	.315	.269	.528	.583	.528	.046	.046	.074	.491	.491	.556	—
MIX-ALL	.324	.333	.287	.528	.556	.583	.194	.222	.231	.472	.500	.509	—

Table 7: EDIT-Bench `pass_rate` in the ablation sweep: every strategy at three annealed milestones, for both model families. *instruction tuned* shows the full-dataset instruction-tuning baseline with no pretraining data. Blue: top value per strategy type exceeding the instruction-tuned weak baseline. Red: top value per strategy type exceeding the instruction-tuned strong baseline. *rank*: overall rank ordered by mean score across the six strong columns. Mixing strategies at equal token shares: MIX-G1 = CLEAN + NOTEBOOK + BLOG; MIX-G2 = the six multi-artifact strategies other than OLD+VERBOSECM; MIX-CROSS = CLEAN + NOTEBOOK + PLAN+CM+OLD; MIX-ALL = the nine arms of MIX-G1 and MIX-G2. Gemma4-12B MIX-CROSS under *weak* is low because about half its outputs run to the 8192-token cap.

	Qwen3-4B		Qwen3-8B		Qwen3-14B		Gemma4-12B		Gemma4-26B		Gemma4-31B	
strategy	easy	hard	easy	hard	easy	hard	easy	hard	easy	hard	easy	hard
untrained base	.355	.109	.581	.130	.581	.152	.323	.043	.339	.109	.548	.217
weak	.387	.130	.403	.065	.419	.087	.274	.087	.403	.152	.500	.196
strong	.661	.370	.710	.413	.742	.370	.597	.304	.710	.348	.677	.304
single artifact												
RAW	.274	.087	.339	.109	.371	.087	.258	.043	.403	.109	.371	.087
CLEAN	.355	.109	.355	.109	.339	.087	.323	.087	.371	.174	.468	.152
NOTEBOOK	.323	.087	.468	.087	.355	.065	.290	.087	.371	.087	.371	.130
BLOG	.355	.130	.419	.152	.403	.087	.355	.087	.452	.152	.484	.196
multi-artifact												
OLD+CM	.419	.109	.419	.087	.565	.196	.355	.109	.435	.130	.468	.152
OLD+TERSECM	.452	.109	.500	.130	.387	.130	.242	.087	.355	.130	.435	.087
PLAN+OLD	.290	.065	.387	.130	.387	.087	.177	.087	.387	.130	.565	.109
PLAN+CM+OLD	.371	.109	.403	.087	.387	.065	.226	.065	.355	.152	.516	.196
PLAN	.371	.087	.387	.130	.452	.130	.306	.087	.419	.196	.339	.087
OLD+REVIEW	.194	.109	.290	.109	.419	.130	.258	.130	.532	.174	.419	.087
+strong												
RAW	.694	.326	.774	.239	.839	.370	.710	.326	.742	.326	.742	.370
CLEAN	.677	.283	.758	.370	.790	.370	.661	.304	.758	.391	**.806**	.326
NOTEBOOK	.823	.283	.790	.304	.758	.348	.661	.348	.677	**.435**	.710	.326
BLOG	.758	.261	.758	.326	.806	.348	.710	.326	.774	.283	.758	.283
OLD+CM	.806	.370	.774	.326	.806	.283	**.758**	.283	.758	.348	**.806**	.348
OLD+TERSECM	**.839**	**.413**	**.839**	.326	.823	.326	.742	.326	**.806**	.370	.790	.348
PLAN+OLD	.742	.304	.726	.304	.806	.304	.694	.283	**.806**	.391	.726	**.391**
PLAN+CM+OLD	.726	.304	.758	.239	.823	.304	.694	**.391**	.742	.348	.774	.326
PLAN	.742	.326	.806	.391	.790	.370	.645	.304	.742	.370	.758	**.391**
OLD+REVIEW	.790	.283	.742	.326	**.855**	.304	.726	.326	**.806**	.283	.742	.326

Table 8: **EDIT-Bench `pass_rate` split by estimated difficulty** (1B annealed checkpoints). EDIT-Bench ships no difficulty field, so the tiers are *estimated*: each of the 108 `test` problems is ranked by how many of the 44 reference models solve it, and split 62 easy / 46 hard – the ratio implied by the EDIT-Bench paper. These tiers approximate but do not reproduce the published easy/hard partition. In each group, the best strategy for that model and tier is marked when it beats the reference row trained on the same non-arm data: underline for a strategy above *weak* only, **bold** for a +*strong* arm above *strong*.

	Qwen3-4B			Qwen3-8B			Qwen3-14B			Gemma4-12B			Gemma4-26B			Gemma4-31B		
arm	easy	med.	hard	easy	med.	hard	easy	med.	hard	easy	med.	hard	easy	med.	hard	easy	med.	hard
untrained base	.471	.077	.007	.535	.082	.020	.639	.103	.016	.432	.046	.007	.600	.103	.016	.645	.159	.020
replay-only	.490	.103	.010	.587	.123	.007	.626	.123	.013	.200	.005	.000	.477	.077	.007	.432	.056	.000
edit-only	.419	.077	.013	.568	.082	.010	.613	.092	.013	.019	.000	.000	.432	.103	.013	.361	.082	.007
Group 1 — single document																		
RAW	.503	.067	.013	.632	.103	.007	.697	.144	.007	.406	.031	.000	.535	.103	.007	.574	.082	.003
CLEAN	.432	.062	.003	.587	.097	.010	.613	.118	.016	.335	.051	.003	.542	.108	.007	.600	.103	.000
NOTEBOOK	.477	.072	.007	.587	.087	.000	.652	.144	.010	.323	.051	.010	.561	.092	.000	.510	.077	.003
BLOG	.477	.082	.003	.542	.108	.010	.710	.123	.013	.368	.051	.000	.548	.118	.016	.665	.097	.000
Group 2 — multi-artifact packing																		
OLD+CM	.535	.087	.013	.561	.087	.010	.606	.097	.020	.368	.026	.003	.613	.082	.016	.594	.092	.000
OLD+TERSECM	.445	.077	.007	.503	.077	.003	.639	.097	.010	.381	.036	.003	.626	.108	.013	.606	.113	.003
PLAN+OLD	.439	.087	.013	.594	.082	.003	.645	.103	.007	.065	.000	.000	.355	.067	.013	.174	.041	.000
PLAN+CM+OLD	.445	.097	.013	.639	.103	.007	.632	.123	.013	.387	.056	.000	.626	.113	.010	.587	.092	.010
PLAN	.465	.067	.003	.523	.082	.003	.671	.128	.020	.239	.056	.000	.497	.077	.000	.432	.082	.000
OLD+REVIEW	.452	.072	.007	.555	.082	.003	.639	.108	.010	.335	.056	.000	.626	.108	.010	.548	.087	.000
+edit condition																		
RAW+edit	**.555**	.072	.016	.613	.092	.007	.710	.144	.016	**.426**	.046	.007	.587	**.144**	.020	.555	**.128**	.003
CLEAN+edit	.523	**.103**	.013	.594	**.128**	.007	.652	**.159**	.016	.355	.051	.007	.594	.113	.013	.606	.113	.010
NOTEBOOK+edit	.452	.103	.007	.600	.077	.013	.703	.144	.013	.374	.056	.003	.542	.108	.016	.561	.103	.003
BLOG+edit	.516	.087	**.026**	.542	.092	.013	**.716**	.149	.010	.348	**.067**	.007	.594	.118	.010	.606	.123	**.016**
OLD+CM+edit	.497	.082	**.026**	**.632**	.082	.007	.677	.154	**.030**	.303	.031	.000	**.619**	.123	.016	.626	.108	.010
OLD+TERSECM+edit	.477	.072	.013	.516	**.128**	**.016**	.703	.128	.020	.387	.062	**.013**	.581	.113	**.023**	.516	.092	.010
PLAN+OLD+edit	.484	.087	.003	.574	**.128**	.007	.639	.144	.010	.381	.031	.000	.613	.138	.007	.568	.087	.000
PLAN+CM+OLD+edit	.426	.077	.003	.581	.097	.007	.703	.133	.000	.323	.046	.010	**.619**	.108	.003	.555	.092	.007
PLAN+edit	.523	.046	.016	.619	.123	.013	.658	.154	.016	.348	.051	.003	.613	.133	.013	.594	.077	.010
OLD+REVIEW+edit	.458	.072	.023	.568	.113	.003	.619	.138	.013	**.426**	.031	.007	.606	.097	.013	**.645**	.108	.007

Table 9: **LiveCodeBench pass@1 split by the benchmark's own difficulty label** (1B annealed checkpoints). The 131-problem February–May 2025 window contains 31 easy, 39 medium and 61 hard problems; the labels ship with LiveCodeBench, so no re-running is needed. *weak* and *strong* are the base trained on the *weak* instruction alone and on the *strong* data alone. In each group, the best strategy for that model and tier is marked when it beats the reference row trained on the same instruction data: underline for a strategy trained with *weak* above tuning on *weak* only, **bold** for those trained with *+strong* above tuning with *strong*. The tiers are small – 31 easy, 39 medium, 61 hard – so the binomial standard error is about .09, .04 and .01 respectively.

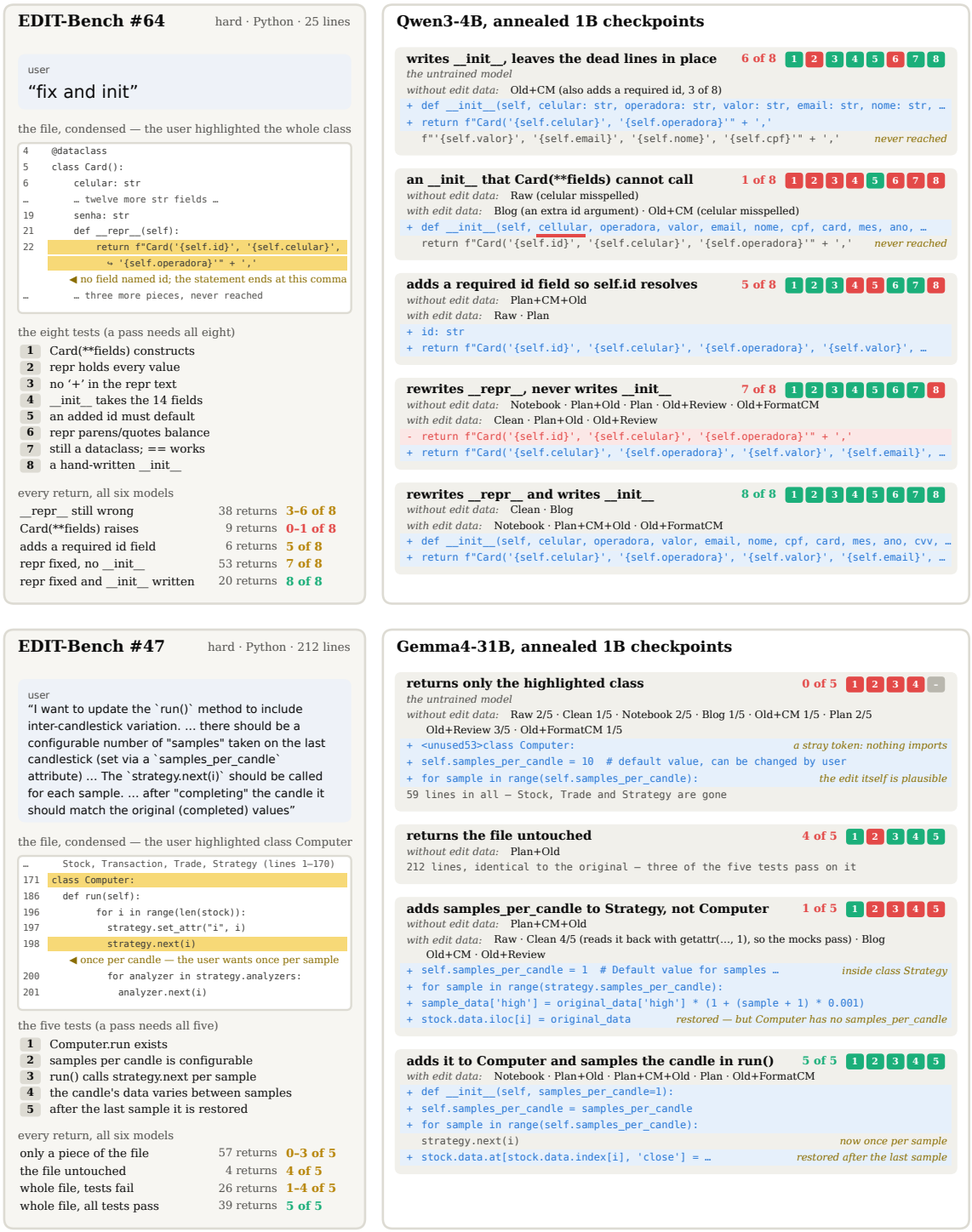

Figure 14: Two EDIT-Bench problems, one per family, grouped by what it did to the file; chips are the problem’s own tests re-run on that return, and the tally repeats the grouping over all six models. *Top, Qwen3-4B on #64:* the untrained model does half of “fix and init” (6 of 8 tests), RAW and OLD+CM make it worse (1 and 3), CLEAN and BLOG do both halves (8). *Bottom, Gemma4-31B on #47:* without *strong* data the base and eight of ten strategies return only the highlighted class, so nothing imports; with it every strategy returns the whole file and five pass. Across all six models, 37% of returns without *strong* data are fragments of the file against 3% with it.

Qwen3-8B, 1B	LiveCodeBench	EDIT-Bench
Target format: diff		
OLD+CM	−.020	+.046
PLAN+CM+OLD	−.027	−.037
OLD+TERSECM	−.029	−.046
mean $\|\Delta\|$	.025	.043
Target format: whole-file		
PLAN+OLD	+.003	.000
PLAN+CM+OLD	−.002	−.009
PLAN	−.003	.000
OLD+REVIEW	+.003	−.065
OLD+TERSECM	−.003	+.065
mean $\|\Delta\|$	.003	.028

Table 10: Which generation decision moves the score most, for Qwen3-8B trained on 1B tokens with *weak* only. Each entry is a paired difference on the same 131 LiveCodeBench problems (pass@1) or 108 EDIT-Bench questions (`pass_rate`). Highlighting marks an exact two-sided sign test $p < 0.05$ (light) and $p < 0.01$ (dark).

F LESSONS LEARNED

Target format matters more than what precedes the code, which matters more than source quality. While we aim to study the effects of pretraining code data, the benchmarks used are the biggest indicators of what the model should be trained on. Intuitively, we want a model that is able to recognize and output in diff format. However, training on such format yields to drops in accuracies when testing the model on benchmarks that require the whole file to be returned (10).

Cosine schedules make milestone snapshots incomparable. A snapshot taken mid-decay has a different effective learning rate from a run that ended at that token count. Warmup-stable-decay with a 10% decay phase keeps every intermediate snapshot in the same regime.

Gradient clipping at 1.0 destabilizes gemma-4. Gemma-4 scales embeddings by $\sqrt{d}$ and runs at gradient norms of 30 to 400. Clipping at 1.0 rescaled every step by a factor that varied $11\times$ between steps; clipping at 200 leaves the normal range untouched and removes only rare spikes.