DO NOT REGENERATE WHAT IS IN YOUR CONTEXT: INSTANT CONTEXT REUSE VIA A POINTER HEAD

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models increasingly process long-context inputs containing content that may reappear in their outputs, making redundant token-by-token generation a source of decoding overhead. Existing efficient decoding methods, including multi-token prediction and speculative decoding, reduce sequential computation through parallel prediction or draft verification but do not explicitly learn to locate and reuse such content. We propose the Pointer-Generator Large Language Model (PG-LLM), which turns context reuse into a model-native generation action. PG-LLM emits a learned control token when reuse is appropriate and uses an attention-based pointer mechanism to predict the start and end positions of a relevant source span. The selected span is inserted into the output before autoregressive generation resumes. We train PG-LLM at 4B and 30B scales. Across three code editing benchmarks, PG-LLM 30B uses $0.71\times$ as many autoregressively decoded tokens as its base model on average and improves editing accuracy by 1.64 percentage points, while preserving comparable overall performance across three code generation benchmarks. With a vLLM serving implementation, PG-LLM improves request throughput over the corresponding base model by 1.12–$1.28\times$ at 4B and 1.13–$1.26\times$ at 30B. These results demonstrate that model-native context localization and reuse can reduce autoregressive decoding and deliver practical serving speedups while maintaining comparable overall code generation performance.

1 INTRODUCTION

Large language models (LLMs) exhibit strong general capabilities in code generation, complex reasoning, and agentic problem solving (Chen et al., 2021; Wei et al., 2022; Yao et al., 2023). As these capabilities expand, LLMs are increasingly deployed in long-context and long-horizon settings, where instructions, interaction histories, retrieved documents, tool outputs, and intermediate results accumulate throughout a task. Efficient decoding therefore becomes increasingly important because autoregressive LLMs still produce outputs one token at a time. At the same time, these accumulated inputs contain task-relevant content—such as retrieved evidence, prior tool outputs, and existing source code—that may later need to appear in the output. Localizing and directly reusing such content offers an opportunity to avoid redundant token-by-token generation.

Existing efficient decoding methods primarily reduce sequential computation by predicting or verifying multiple future tokens at once. Multi-token prediction uses additional heads to predict several future positions, early-exit and self-speculative methods draft with intermediate layers, and speculative decoding verifies sequences proposed by a smaller draft model (Gloeckle et al., 2024; Cai et al., 2024; Elhoushi et al., 2024; Leviathan et al., 2023; Chen et al., 2023). These methods improve general decoding efficiency but do not fully exploit task-relevant content already available in long-context inputs, as they do not explicitly teach the model to localize and reuse relevant spans. This leaves open a complementary direction: making learned context localization and reuse a native action of the model itself.

One setting that directly exposes the connection between context localization, reuse, and decoding efficiency is code editing. An instruction may require changing only a small part of a file while asking the model to return the complete revised code. Imports, class definitions, and unchanged program logic must then be generated again, even though they already appear in the input. Figure 1 makes this opportunity concrete: for an edit that adds an `ASeq` class, PG-LLM copies unchanged spans and

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.

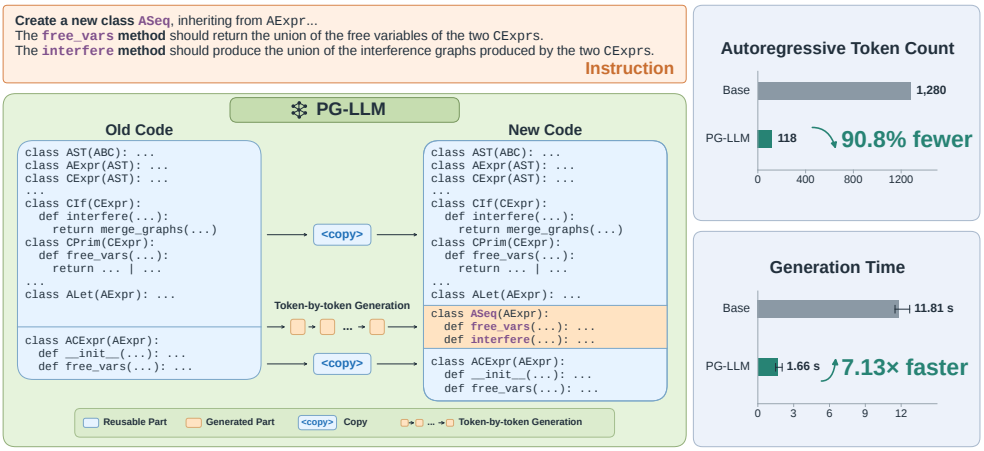


Figure 1: **Context reuse accelerates code editing.** This example asks the model to add an `ASeq` class. The left panel shows PG-LLM's Copy–Generate–Copy editing path, with copied code in blue and generated code in orange; the right reports vLLM measurements. With identical final code, PG-LLM reduces autoregressive decoding from 1,280 to 118 tokens (90.8% fewer) and median generation time from 11.81 to 1.66 s (7.13× faster). Results are measured over 10 runs, with error bars showing the 10th–90th percentiles. This example highlights a common property of code editing: substantial portions of the output can be reused to enable efficient decoding.

generates the new implementation, reducing both autoregressive token count and measured vLLM generation time while producing identical final code.

Pointer mechanisms offer a natural basis for making reuse a model decision. Pointer Networks reinterpret attention as a distribution over input positions: rather than producing an item from a fixed output vocabulary, the decoder selects an element of a variable-length input sequence (Vinyals et al., 2015). Pointer-generator models combine this positional distribution with ordinary vocabulary generation, allowing the model to alternate between generating new tokens and copying existing input content (Gu et al., 2016; See et al., 2017). This principle is particularly relevant to LLMs, whose long-context inputs provide contextualized representations of many potentially reusable spans. Recent systems such as GUI-Actor and MolmoPoint similarly use special grounding tokens and attention to select visual regions directly (Wu et al., 2025; Clark et al., 2026). Building on these ideas, we propose **PG-LLM**, a Pointer-Generator Large Language Model that equips LLMs with model-native context localization and reuse. The model learns to emit a control token when reuse is appropriate, applies pointer attention to select the start and end of a relevant source span, inserts that span into the output, and then resumes autoregressive generation. We instantiate this mechanism in code editing using abstract syntax tree (AST)-aligned copy traces to supervise reuse at structural code boundaries.

To evaluate whether learned context reuse improves decoding efficiency without sacrificing task performance, we train PG-LLM variants initialized from Qwen3-4B-Instruct-2507 and Qwen3-30B-A3B-Instruct-2507 (Yang et al., 2025). Across three code editing evaluations, PG-LLM 30B after direct preference optimization (DPO) uses $0.71\times$ as many autoregressively decoded tokens as its base model on average and improves editing accuracy by 1.64 percentage points. Both variants preserve comparable overall performance on three code generation benchmarks. With a vLLM serving implementation, the 4B and 30B variants improve request throughput over their corresponding base models by 1.12–$1.28\times$ and 1.13–$1.26\times$, respectively. Ablations on the 4B model show that AST-aligned traces offer a better correctness–efficiency balance than line-level copying, while preference optimization reduces inappropriate copying on generation tasks for both backbones.

Beyond its immediate decoding benefits, PG-LLM introduces a model-native operation for locating and reusing content already present in the context. Long-context models and agents currently organize accumulated task content largely through external retrieval-augmented generation, memory, context-processing tools, and skill libraries (Packer et al., 2024; Liu et al., 2025; Wang et al., 2023; Su

et al., 2026). PG-LLM complements these systems by making context localization and reuse explicit actions within generation. This operation provides a starting point for richer model-native context management.

Our contributions are:

- We propose PG-LLM, which introduces a model-native context-reuse operation that reduces token-by-token generation and enables efficient decoding.
- We revisit pointer mechanisms for modern LLMs and design an attention-based mechanism that localizes relevant source spans and reuses them during generation.
- PG-LLM reduces autoregressive decoding on code editing tasks while preserving comparable overall code generation performance; PG-LLM 30B achieves an average token ratio of $0.71\times$ and a 1.64-percentage-point accuracy gain, while vLLM deployment yields up to $1.28\times$ and $1.26\times$ throughput speedups at 4B and 30B.

2 Related Work

PG-LLM lies at the intersection of efficient decoding and explicit context reuse, two complementary ways to reduce redundant autoregressive generation. We review efficient decoding approaches and pointer mechanisms for selecting contextual content.

2.1 Efficient Decoding and Context Reuse

Autoregressive language models generate one token per step, motivating methods that reduce sequential decoding. Multi-token prediction uses auxiliary heads to predict several future positions from a shared representation; Medusa constructs candidate trees with multiple decoding heads for parallel verification (Gloeckle et al., 2024; Cai et al., 2024). Early-exit and self-speculative decoding use intermediate layers as an internal drafter and later layers as a verifier, avoiding a separate model; LayerSkip improves intermediate predictions through layer dropout and an early-exit loss (Elhoushi et al., 2024). Speculative decoding instead uses a cheaper draft model to propose sequences that the target model verifies in parallel, with acceptance and correction preserving the target distribution (Leviathan et al., 2023; Chen et al., 2023).

These approaches accelerate the prediction or verification of newly generated tokens, but do not explicitly exploit spans already available in long contexts. PG-LLM instead makes context reuse a model-native action: it emits a reuse control token and uses an attention-based pointer mechanism to localize a relevant source span. Directly reusing the selected span reduces the number of tokens requiring autoregressive decoding, providing a complementary route to efficient decoding.

2.2 Pointer Mechanisms for Context Localization and Reuse

Pointer Networks use attention to predict positions in variable-length inputs (Vinyals et al., 2015). CopyNet and pointer-generator networks combine vocabulary generation with direct input copying (Gu et al., 2016; See et al., 2017). Later methods extend copying to spans and structured edits (Zhou et al., 2018; Panthaplackel et al., 2020; Stahlberg & Kumar, 2020; Mallinson et al., 2020). Together, they establish explicit input selection as an alternative to vocabulary-only generation.

Modern multimodal models similarly use special grounding tokens for explicit selection. GUI-Actor uses an `<ACTOR>` token and attention-based action head to select visual patches (Wu et al., 2025). MolmoPoint uses hierarchical location tokens for coarse-to-fine pointing and feeds selected information back into model representations (Clark et al., 2026). PG-LLM adapts this interface to textual reuse by predicting span boundaries in the context and continuing generation from the selected content.

3 Model-Native Context Reuse With Pointer Generator

PG-LLM augments a standard autoregressive LLM with a pointer generator for model-native context localization and reuse. The model learns when to select an existing source span and when to generate

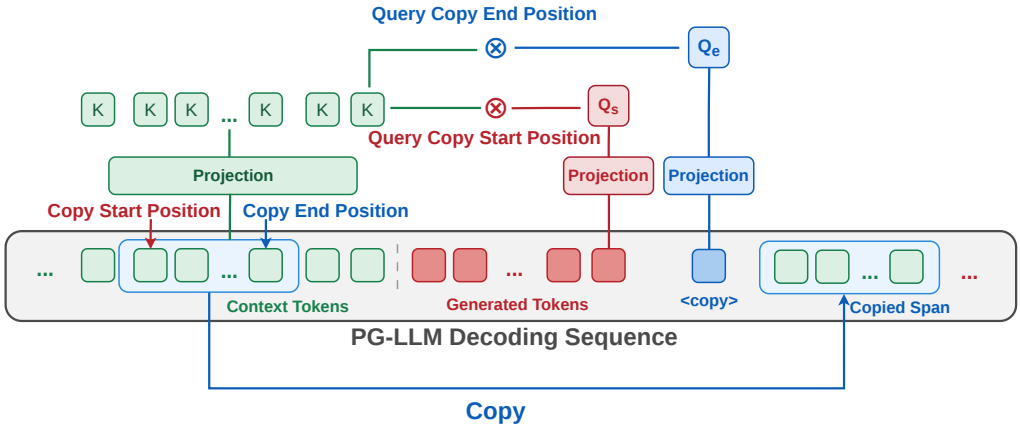


Figure 2: **The pointer generator in PG-LLM.** The current decoding state selects a copy start position. Its source representation enhances the input embedding of <copy>, whose contextualized state then selects the end position. The selected span is copied into the output and prefilled before decoding resumes.

new content, allowing reusable content to bypass token-by-token vocabulary decoding. Figure 2 presents the architecture. We describe the selection mechanism here and the training and inference procedures in Section 4.

3.1 Turning Context Reuse into a Native Decoding Action

Let $x = (x_1, \ldots, x_n)$ be the input prompt and $y_{<t}$ the current internal output prefix, including any copied spans and control tokens. The last-layer state h_t predicts the next token through the usual language-model head. We extend the vocabulary with a learned control token $c =$ <copy>:

$$p_{\mathrm{LM}}(y_t \mid x, y_{<t}) = \mathrm{softmax}(W_{\mathrm{LM}} h_t). \tag{1}$$

An ordinary token continues standard generation. Predicting c instead triggers a copy action that selects an inclusive interval $[s, e]$ in the input. The decision to copy is therefore learned through the same output interface as ordinary text, rather than imposed by a text-matching rule at inference time.

We retain contextualized source states $H^x = (h_1^x, \ldots, h_n^x)$ from the prompt. A source mask specifies eligible copying positions. Candidates come from the input, not from the generated prefix, preventing recursive copying of the model's own output. In code editing, this mechanism reuses existing source code while the model still returns a complete revised program.

3.2 Localizing Reusable Spans with Start-Conditioned Pointers

Selecting the start. The state h_t that predicts <copy> also predicts the copy start. In particular, this is the state *before* processing the control token, not the control token's own contextualized state. Learned projections map this query and the source states to a pointer space of dimension d_p:

$$q_t^s = W_Q^s \,\mathrm{RMSNorm}(h_t)\,, \qquad k_i = W_K \,\mathrm{RMSNorm}(h_i^x)\,, \tag{2}$$

$$p_s(i \mid x, y_{<t}, c) = \mathrm{softmax}_i\left(\frac{(q_t^s)^\top k_i}{\sqrt{d_p}} + M_i\right). \tag{3}$$

Here M_i is 0 for eligible source positions and $-\infty$ otherwise; the query and key normalizations have their own parameters. Training uses the labeled start s^*, whereas inference selects $\hat{s} = \arg\max_i p_s(i)$.

Conditioning on the selected start. The end predictor needs to know which source position was selected. We inject its contextualized representation into the input embedding of <copy>:

$$\widetilde{e}_c(s) = e_c + \sigma(\gamma)\, W_I \,\mathrm{RMSNorm}(h_s^x)\,, \tag{4}$$

where e_c is the control-token embedding, W_I is a learned projection, and γ is a learned scalar gate. A Transformer step with this embedding produces $h_c(s)$, conditioned on the existing prefix. This makes the selected start available within the model's representations, rather than retaining it only as an external index. We use the ground-truth start during training and the predicted start during inference.

Selecting the end. An independent query projection maps the enhanced state $h_c(s)$ into the same pointer space. We reuse the source keys and mask positions before the selected start:

$$q_c^e = W_Q^e \, \mathrm{RMSNorm}(h_c(s)) \,, \tag{5}$$

$$p_e(i \mid x, y_{<t}, c, s) = \mathrm{softmax}_i \left(\frac{(q_c^e)^\top k_i}{\sqrt{d_p}} + M_i + M_i^{\geq s} \right), \tag{6}$$

where $M_i^{\geq s}$ is 0 when $i \geq s$ and $-\infty$ otherwise. The predicted endpoint $\hat{e}$ determines the copied span $(x_{\hat{s}}, \ldots, x_{\hat{e}})$. Unlike two independent boundary predictions, this factorization explicitly conditions the end on the selected start.

3.3 Resuming Generation after Span Reuse

After selecting a span, the decoder appends the control token and source content to its internal sequence:

$$y_{\mathrm{new}} = y_{<t} \parallel c \parallel (x_{\hat{s}}, \ldots, x_{\hat{e}}). \tag{7}$$

The copied tokens are processed by prefill to update the key–value (KV) cache. Subsequent generation can therefore condition on their actual contents, including identifiers, indentation, and local program structure. The control token is removed from the delivered text, but the selected code remains in both the output and the model's context.

This operation skips vocabulary decoding for the copied tokens; it does not remove them from the context or eliminate their forward computation. The efficiency benefit is therefore first measured by the autoregressive token ratio relative to standard decoding. Because pointer scoring and copied-span prefill still incur cost, the token ratio alone is not an estimate of wall-clock speedup; Section 5.4 separately reports measured serving throughput. Appendix A gives the full decoding procedure.

4 Learning When and What to Reuse

We train PG-LLM on copy/generate trajectories to learn when and what context to reuse, followed by preference optimization to calibrate reuse behavior and preserve ordinary generation performance. We present the essential choices below; dataset statistics and training configurations are in Appendices B and C.

4.1 Structure-Aligned Supervision with AST Copy Traces

Given an editing instruction, old code x^{code}, and target code z, we construct a trajectory whose actions either copy a source span or generate new text. Realizing the trajectory must reconstruct z exactly. We obtain editing examples from CommitPackFT, InstructCoder, OCEDataFT, and NextCoderDataset (Muennighoff et al., 2024; Li et al., 2024; Zhang et al., 2025; Aggarwal* et al., 2025). Pure generation examples from OpenCodeInstruct (NVIDIA, 2025) provide trajectories without copying.

Rather than matching arbitrary fragments, we extract candidate copy units from the Python AST, including functions, classes, compound statements, and top-level simple statements. A left-to-right scan of the target selects the longest eligible matching block and generates unmatched regions. This favors copying unchanged structural units instead of repeatedly alternating between copying and generation within a block. The candidates are mapped to prompt-token boundaries and checked for exact reconstruction. AST alignment constrains the construction of training traces; it is not a hard syntax constraint on predicted spans at inference time.

4.2 Learning Copy-or-Generate Decisions Through Supervised Fine-Tuning

The 4B and 30B models share the same tokenizer and SFT dataset, containing 22,614 examples and 3,999,871 supervised tokens. Supervision covers ordinary generated tokens and `<copy>`, together with the labeled start and end of each copy event:

$$\mathcal{L}_{\mathrm{SFT}} = \mathcal{L}_{\mathrm{LM}} + 0.50\,\mathcal{L}_{\mathrm{start}} + 0.50\,\mathcal{L}_{\mathrm{end}}. \tag{8}$$

For the 30B mixture-of-experts backbone, the SFT objective additionally includes a router auxiliary loss with weight 0.001. Copied tokens are present in the training context but masked out of the LM loss, since the vocabulary head does not generate them individually. We adapt both Qwen3 backbones with LoRA (Hu et al., 2022), while training the new control-token and pointer parameters. The 4B SFT run uses one epoch, a learning rate of 2×10^{-4}, and an effective batch size of 16. The 30B SFT run also uses one epoch and an effective batch size of 16, with learning rates of 2×10^{-5} for attention and expert adapters and 2×10^{-6} for the router adapter. Mixing in pure generation data also gives the model examples in which it should not invoke copying.

4.3 Calibrating Selective Reuse with Preference Optimization

SFT teaches the model to reuse context but can also cause inappropriate copying on ordinary generation tasks. We subsequently apply DPO (Rafailov et al., 2023) with each backbone's SFT model as its fixed reference. A trajectory's log probability includes the LM probabilities of generated tokens and control tokens, as well as the start- and end-pointer probabilities. Copied content is determined by the selected spans and contributes no additional vocabulary-generation probability. We supplement the DPO objective with a small supervised loss on preferred trajectories. The 30B DPO run uses one epoch, an effective batch size of 16, a learning rate of 1×10^{-5}, $\beta = 0.10$, and an auxiliary chosen-trajectory SFT coefficient of $\alpha = 0.05$.

Preference data contain 1,000 editing pairs and 1,500 generation pairs for PG-LLM 4B, and 2,000 editing pairs and 3,000 generation pairs for PG-LLM 30B. Both variants use the same data sources. Editing prompts come from the four editing sources, excluding SFT examples; sampled candidates are judged by DeepSeek V4 Flash (DeepSeek-AI, 2026). Generation prompts also come from OpenCodeInstruct, from examples disjoint from its SFT subset. For these prompts, a passing base-model response is preferred over a failing SFT-model response. Thus, the generation preference prompts are drawn from a training-data source, not constructed from the reported generation benchmarks. Appendix C.2 describes the selection procedure and objective.

4.4 Decoding with Learned Context Reuse

Within each benchmark, both base models and their PG-LLM variants use the same prompt and chat templates and the same evaluation decoding settings. PG-LLM additionally retains source representations for pointer selection. When copying is triggered, it selects the highest-scoring start and then the highest-scoring valid end, inserts the span, and prefills it before resuming ordinary decoding. In contrast to an external patch format, this procedure returns the full revised code. Copying remains a model decision: a generation-only response follows the ordinary vocabulary path throughout.

5 Does Learned Context Reuse Deliver Practical Decoding Gains?

Our evaluation asks whether learned context localization and reuse can reduce autoregressively decoded tokens in code editing, preserve comparable overall code generation performance, translate into serving speedups, and benefit from structure-aligned rather than line-level copy supervision.

5.1 Experimental Setup

Models and benchmarks. We initialize PG-LLM 4B and PG-LLM 30B from Qwen3-4B-Instruct-2507 and Qwen3-30B-A3B-Instruct-2507, respectively, and compare each variant's SFT and DPO

Table 1: **Code editing results.** Accuracy (Acc.) is a percentage, and Token is the number of autoregressively decoded tokens relative to each variant's own base model. Arrows in column headers indicate the preferred direction. CommitPackFT uses an LLM judge; the other datasets use functional tests. Avg. is the unweighted mean of the three dataset scores. Bold marks the best value in each column within each backbone group.

	CommitPackFT		CanItEdit		EditEval		Avg.	
Model	Acc. ↑	Token ↓	Acc. ↑	Token ↓	Acc. ↑	Token ↓	Acc. ↑	Token ↓
Qwen3-4B-Instruct-2507	70.42	1.00×	52.38	1.00×	60.31	1.00×	61.04	1.00×
PG-LLM 4B SFT	**71.83**	**0.61**×	54.29	**0.62**×	57.22	**0.80**×	61.11	**0.68**×
PG-LLM 4B DPO	71.36	0.63×	**56.19**	0.65×	**62.89**	0.83×	**63.48**	0.70×
Qwen3-30B-A3B-Instruct-2507	82.16	1.00×	64.76	1.00×	73.20	1.00×	73.37	1.00×
PG-LLM 30B SFT	**83.57**	**0.63**×	**67.62**	**0.65**×	70.62	**0.79**×	73.94	**0.69**×
PG-LLM 30B DPO	83.10	0.65×	66.67	0.66×	**75.26**	0.82×	**75.01**	0.71×

checkpoints with its own unmodified base model. Editing evaluations comprise 213 held-out CommitPackFT examples, 105 CanItEdit examples, and 194 EditEval examples (Muennighoff et al., 2024; Cassano et al., 2024; Li et al., 2024). CommitPackFT uses an LLM judge; CanItEdit and EditEval use functional tests. We separately evaluate generation on MBPP+, LiveCodeBench, and BigCodeBench-Hard (Liu et al., 2023; Jain et al., 2025; Zhuo et al., 2025). All six models use the same evaluation pipeline and subsets within a benchmark. Our comparisons evaluate complete trained systems, including the pointer architecture and post-training, against their corresponding unmodified base models. Appendix D details the evaluation subsets and protocols.

Metrics. We report judge accuracy for CommitPackFT and single-response functional correctness for the execution-based evaluations. For each editing dataset, we compute the corpus-level autoregressive token ratio

$$R_{\text{tok}} = \frac{\sum_i D_i^{\text{PG}}}{\sum_i D_i^{\text{base}}}, \tag{9}$$

where D_i^{base} counts the corresponding backbone's base-model autoregressively decoded code tokens and D_i^{PG} counts PG-LLM's ordinary generated tokens plus copy control tokens, excluding copied tokens. For each model, we first aggregate counts within each dataset and then take the arithmetic mean of dataset scores. The resulting average is not a pooled token ratio over all benchmarks or backbones. Generation evaluations additionally report the fraction of examples with at least one copy trigger. Percentages, ratios, and benchmark averages are reported to two decimal places.

5.2 PG-LLM Reduces Autoregressive Decoding without Sacrificing Editing Quality

Table 1 shows that SFT already enables substantial context reuse on both backbones, using $0.68\times$ and $0.69\times$ as many autoregressively decoded tokens as the 4B and 30B base models on average. Average editing accuracy is 61.11% and 73.94%, respectively, compared with base-model scores of 61.04% and 73.37%. Dataset-level changes are mixed: both SFT models improve on CommitPackFT and CanItEdit but decline on EditEval. These dataset-level differences motivate the preference-optimization stage.

After DPO, PG-LLM 30B reaches 75.01% average editing accuracy, a gain of 1.64 percentage points over its base model, while using $0.71\times$ as many autoregressively decoded tokens. Its EditEval accuracy rises from 70.62% after SFT to 75.26%, although CommitPackFT and CanItEdit accuracy decrease slightly relative to SFT. The 4B model likewise improves in aggregate, reaching 63.48% accuracy, 2.44 percentage points above its base model, with an average token ratio of $0.70\times$. DPO increases the token ratios slightly relative to SFT but improves average correctness while still avoiding substantial sequential vocabulary decoding. Appendix G provides a concrete 4B DPO editing example, with reusable and regenerated regions distinguished by colored code blocks.

Table 2: **Code generation performance and copy triggers.** MBPP+ contains 378 examples and uses base tests for both backbones, not the augmented tests. Pass@1 and trigger rates are percentages. Arrows in column headers indicate the preferred direction. A trigger means at least one `<copy>` is emitted on a generation task. Base models have no copy action. Avg. averages Pass@1, not trigger rates. Bold marks the best value within each backbone group.

	LiveCodeBench		BigCodeBench-Hard		MBPP+		Avg.
Model	Pass@1 ↑	Trigger ↓	Pass@1 ↑	Trigger ↓	Pass@1 ↑	Trigger ↓	Pass@1 ↑
Qwen3-4B-Instruct-2507	**25.19**	–	22.97	–	78.04	–	42.07
PG-LLM 4B SFT	21.37	17.56	18.24	16.22	74.60	**0.00**	38.07
PG-LLM 4B DPO	24.43	**0.76**	**23.65**	**0.68**	**79.10**	**0.00**	**42.39**
Qwen3-30B-A3B-Instruct-2507	**43.51**	–	41.22	–	**88.89**	–	**57.87**
PG-LLM 30B SFT	38.93	18.32	37.16	14.86	85.71	**0.00**	53.93
PG-LLM 30B DPO	41.98	**1.53**	**42.57**	**2.03**	88.36	**0.00**	57.64

Table 3: **Serving efficiency of PG-LLM at 4B and 30B scales.** Req/s is loaded-engine throughput, Token is the relative autoregressive token count, and Speedup is normalized to the corresponding vLLM base model. Arrows in column headers indicate the preferred direction. EditEval pointer runs use the repeated-span guard described in Appendix E.

	CommitPackFT			CanItEdit			EditEval		
Model	Req/s↑	Token↓	Speedup↑	Req/s↑	Token↓	Speedup↑	Req/s↑	Token↓	Speedup↑
Qwen3-4B (vLLM)	5.01	1.00×	1.00×	1.47	1.00×	1.00×	9.63	1.00×	1.00×
PG-LLM 4B (Transformers)	0.38	0.63×	0.08×	0.27	0.65×	0.18×	0.28	0.83×	0.03×
PG-LLM 4B (vLLM)	**6.06**	0.63×	**1.21**×	**1.88**	0.65×	**1.28**×	**10.79**	0.83×	**1.12**×
Qwen3-30B-A3B (vLLM)	2.94	1.00×	1.00×	0.88	1.00×	1.00×	5.73	1.00×	1.00×
PG-LLM 30B (Transformers)	0.15	0.65×	0.05×	0.14	0.66×	0.16×	0.07	0.82×	0.01×
PG-LLM 30B (vLLM)	**3.47**	0.65×	**1.18**×	**1.11**	0.66×	**1.26**×	**6.47**	0.82×	**1.13**×

5.3 Preference Optimization Makes Context Reuse Selective

Effective context reuse also requires the model to recognize when no available span should be reused. Table 2 shows that SFT reduces average generation Pass@1 from 42.07% to 38.07% for 4B and from 57.87% to 53.93% for 30B. SFT also produces frequent copy triggers: 17.56% and 18.32% on LiveCodeBench, and 16.22% and 14.86% on BigCodeBench-Hard, for 4B and 30B respectively. Generation prompts can contain signatures, interface descriptions, or code templates that resemble reusable source context. Copying these fragments inappropriately is one possible failure mode. However, both SFT models decline on MBPP+ despite 0.00% trigger rates, so spurious copying does not explain every generation error.

DPO reduces LiveCodeBench and BigCodeBench-Hard trigger rates to 0.76% and 0.68% for 4B, and to 1.53% and 2.03% for 30B; both retain 0.00% on MBPP+. Average Pass@1 recovers to 42.39% for 4B and 57.64% for 30B, compared with their base models' 42.07% and 57.87%. DPO thus restores near-baseline generation performance on average, with benchmark-specific variation: the 30B average remains 0.23 percentage points below its base model, with a 1.53-point gap on LiveCodeBench. Taken together with the editing results, these findings show that both trained models can reuse context where useful while largely avoiding copy actions on generation-only tasks.

5.4 Token Savings Translate into Real Serving Speedups

To measure serving efficiency including pointer scoring, request-state updates, and span prefill, we integrate both the 4B and 30B DPO models into the vLLM V1 EngineCore (Kwon et al., 2023) and compare each with its native vLLM base model and our Transformers reference implementation. Table 3 reports throughput averaged over three runs; model loading and initialization are excluded. Appendix E gives implementation, hardware, and measurement details.

Table 4: **Copy-granularity ablation on the 4B backbone.** Accuracy and Pass@1 are percentage averages across three datasets; Token is the average relative autoregressive token count. Arrows in column headers indicate the preferred direction. Both pointer variants use SFT without DPO. Appendix F.1 gives per-dataset results and trigger rates.

Model	Editing accuracy ↑	Token ↓	Generation Pass@1 ↑
Qwen3-4B-Instruct-2507	61.04	1.00×	**42.07**
PG-LLM 4B SFT (line-level)	58.08	**0.57**×	33.76
PG-LLM 4B SFT (block-level)	**61.11**	0.68×	38.07

For the 4B model, the vLLM implementation raises throughput over native vLLM by $1.21\times$, $1.28\times$, and $1.12\times$ on CommitPackFT, CanItEdit, and EditEval, respectively. The corresponding 30B speedups are $1.18\times$, $1.26\times$, and $1.13\times$. These gains are smaller than the reductions indicated by the token ratios because pointer operations and copied-span prefill remain computational costs. Nevertheless, the consistent gains across both scales show that reduced sequential decoding yields practical serving improvements when paired with a batched inference engine.

5.5 Effective Reuse Requires Structure, Not Maximum Copying

On the 4B backbone, we compare AST-aligned block-level traces against line-level traces that permit finer-grained reuse. Table 4 shows that line-level copying achieves a lower average token ratio, $0.57\times$ versus $0.68\times$, but lower editing accuracy, 58.08% versus 61.11%. Its generation Pass@1 is also lower on average, 33.76% versus 38.07%. Minimizing the token ratio is therefore not the same as learning an accurate and effective context reuse policy.

The two 4B SFT variants have similar generation trigger rates: 19.08% versus 17.56% on LiveCodeBench and 16.89% versus 16.22% on BigCodeBench-Hard. Trigger frequency alone cannot account for their performance gap. Fragmented traces may make coherent transitions between copied and generated code harder to learn. This 4B ablation favors structure-aligned supervision over maximizing the amount copied; both main PG-LLM variants therefore use block-level traces.

6 Conclusion

We propose PG-LLM, a pointer-generator-based approach to efficient decoding that equips large language models with model-native context localization and reuse capabilities. By learning when to invoke reuse and which source span to select, PG-LLM avoids token-by-token regeneration of content already available in the context. We instantiate the method on Qwen3-4B-Instruct-2507 and Qwen3-30B-A3B-Instruct-2507. Across three code editing benchmarks, PG-LLM 30B achieves an average autoregressive token ratio of $0.71\times$ together with a 1.64-percentage-point improvement in editing accuracy, while both variants preserve comparable overall code generation performance. Under vLLM, the 4B and 30B variants improve request throughput over their corresponding base models by 1.12–$1.28\times$ and 1.13–$1.26\times$, respectively. These results demonstrate that learned context localization and reuse can yield both fewer autoregressive decoding steps and practical serving speedups, while providing a foundation for richer model-native context operations.

Limitations. Our evaluation covers two Qwen3 backbones on code tasks with Python AST-aligned traces, with serving measurements on three code editing workloads and one GPU type. Granularity ablations and training diagnostics use the 4B model, where end prediction remains a bottleneck. Comparisons with base models measure the combined effects of architecture and post-training. Evaluating long-context scaling, long-horizon agent performance, other text domains, and additional serving configurations remains future work.

AI Use Statement

For manuscript preparation, generative AI assistants were used only for English translation, language polishing, and LaTeX formatting. They were not used to generate experimental results. The use of DeepSeek V4 Flash as an experimental judge is documented separately as part of the methodology in

Appendices C.2 and D. The authors reviewed all AI-assisted manuscript edits and take responsibility for the final content and claims of the paper.

Ethics Statement

This work studies decoding efficiency using public code and instruction datasets and does not involve human-subject experiments. It inherits potential licensing, provenance, bias, and code-security risks from the underlying datasets and base models. DeepSeek V4 Flash judges editing preference candidates and CommitPackFT responses; the remaining benchmarks use functional tests.

Reproducibility Statement

Appendix A specifies the pointer factorization and decoding procedure. Appendices B and C describe the shared SFT data, model-specific preference-data sizes, copy-trace construction, LoRA targets, and 4B and 30B optimization settings. Evaluation subsets and metric definitions appear in Appendix D, and Appendix E documents the 4B and 30B vLLM integration, hardware, and serving protocol. Per-dataset ablations and training diagnostics appear in Appendix F. The 4B DPO editing case in Appendix G illustrates the output format and distinguishes its source-normalized token ratio from the baseline-relative benchmark ratio.

References

Tushar Aggarwal*, Swayam Singh*, Abhijeet Awasthi, Aditya Kanade, and Nagarajan Natarajan. Nextcoder: Robust adaptation of code lms to diverse code edits. In *ICML 2025*, July 2025.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Anton Lozhkov, Carolyn Jane Anderson, and Arjun Guha. Can It Edit? Evaluating the Ability of Large Language Models to Follow Code Editing Instructions. In *Conference on Language Modelling (COLM)*, 2024.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling, 2023.

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

Christopher Clark, Yue Yang, Jae Sung Park, Zixian Ma, Jieyu Zhang, Rohun Tripathi, Mohammadreza Salehi, Sangho Lee, Taira Anderson, Winson Han, and Ranjay Krishna. Molmopoint: Better pointing for vlms with grounding tokens, 2026.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, Ahmed Aly, Beidi Chen, and Carole-Jean Wu. LayerSkip: Enabling early exit inference and self-speculative decoding. 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. 12622–12642, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.681.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. Better & faster large language models via multi-token prediction. *arXiv preprint arXiv:2404.19737*, 2024.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. Incorporating copying mechanism in sequence-to-sequence learning. In Katrin Erk and Noah A. Smith (eds.), *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1631–1640, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1154.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

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. In *The Thirteenth International Conference on Learning Representations*, 2025.

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. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding, 2023.

Kaixin Li, Qisheng Hu, James Xu Zhao, Hui Chen, Yuxi Xie, Tiedong Liu, Michael Shieh, and Junxian He. InstructCoder: Instruction tuning large language models for code editing. In Xiyan Fu and Eve Fleisig (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pp. 473–493, Bangkok, Thailand, August 2024. Association for Computational Linguistics. ISBN 979-8-89176-097-4.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 21558–21572. Curran Associates, Inc., 2023.

Shukai Liu, Jian Yang, Bo Jiang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents, 2025.

Jonathan Mallinson, Aliaksei Severyn, Eric Malmi, and Guillermo Garrido. FELIX: Flexible text editing through tagging and insertion. In Trevor Cohn, Yulan He, and Yang Liu (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1244–1255, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.111.

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.

NVIDIA. OpenCodeInstruct. Dataset, 2025.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024.

Sheena Panthaplackel, Miltiadis Allamanis, and Marc Brockschmidt. Copy that! editing sequences by copying spans, 2020.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Abigail See, Peter J. Liu, and Christopher D. Manning. Get to the point: Summarization with pointer-generator networks. In Regina Barzilay and Min-Yen Kan (eds.), *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1073–1083, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1099.

Felix Stahlberg and Shankar Kumar. Seq2Edits: Sequence transduction using span-level edit operations. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 5147–5159, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.418.

Weihang Su, Jianming Long, Qingyao Ai, Yichen Tang, Changyue Wang, Yiteng Tu, and Yiqun Liu. Skill retrieval augmentation for agentic ai, 2026.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer Networks. In *Advances in Neural Information Processing Systems*, volume 28, 2015.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Qianhui Wu, Kanzhi Cheng, Rui Yang, Chaoyun Zhang, Jianwei Yang, Huiqiang Jiang, Jian Mu, Baolin Peng, Bo Qiao, Reuben Tan, Si Qin, Lars Liden, Qingwei Lin, Huan Zhang, Tong Zhang, Jianbing Zhang, Dongmei Zhang, and Jianfeng Gao. Gui-actor: Coordinate-free visual grounding for gui agents, 2025.

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.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Zekai Zhang, Mingwei Liu, Zhenxi Chen, Linxi Liang, Yuxuan Chen, Guangsheng Ou, Yanlin Wang, Dan Li, Xin Peng, and Zibin Zheng. Generating high-quality datasets for code editing via open-source language models, 2025.

Qingyu Zhou, Nan Yang, Furu Wei, and Ming Zhou. Sequential copying networks, 2018.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*, 2025.

A Architecture and Decoding Details

A.1 Representations and Conditional Selection

The prompt $x = (x_1, \ldots, x_n)$ includes the instruction and any source material. The source states $h_i^x \in \mathbb{R}^d$ are the last-layer Transformer representations of prompt tokens. The decoding state $h_t \in \mathbb{R}^d$ is the final state of the currently processed prefix, used to predict the next token. In a copy event, $W_Q^s, W_Q^e, W_K \in \mathbb{R}^{d_p \times d}$ project the two queries and source keys to the pointer space. The injection projection is $W_I \in \mathbb{R}^{d \times d}$. Query, key, and injection normalization are learned RMSNorm operations; the start and end pointers have separate query projections and share source keys.

Writing the binary source mask as $m_i \in \{0, 1\}$, the additive mask in Equations 3 and 6 is $M_i = \log m_i$, with $\log 0 = -\infty$. The end mask additionally imposes $i \geq s$. Thus, the action probability factorizes as

$$p(c, s, e \mid x, y_{<t}) = p_{\mathrm{LM}}(c \mid x, y_{<t})\, p_s(s \mid x, y_{<t}, c)\, p_e(e \mid x, y_{<t}, c, s). \tag{10}$$

The copy trigger and the start query use the same pre-control-token decoding state. The end query instead uses

$$h_c(s) = \mathrm{TransformerStep}(e_c + \sigma(\gamma) W_I\, \mathrm{RMSNorm}(h_s^x)\,; \mathrm{KV}(x, y_{<t}))\,. \tag{11}$$

The scalar $\sigma(\gamma)$ controls the strength of the injected source representation, allowing the model to learn how strongly the selected start should condition the control-token state.

During supervised training, $s = s^*$ is the labeled source start. The endpoint is supervised under this ground-truth conditioning, with per-event losses

$$\ell_s = -\log p_s(s^* \mid x, y_{<t}, c), \qquad \ell_e = -\log p_e(e^* \mid x, y_{<t}, c, s^*). \tag{12}$$

At inference time, $s = \hat{s}$ is the predicted start. A start error can therefore affect the endpoint both through the injected representation and through the endpoint mask. This difference between training and inference is important when interpreting boundary-prediction accuracy.

A.2 Decoding Procedure

The following procedure describes a single-sequence decode. Source indices refer to prompt-token positions, and span endpoints are inclusive.

1. Prefill the prompt, retain the eligible source states, and initialize the output prefix and KV cache.
2. Use the current decoding state to predict the next vocabulary token. If it is an end-of-sequence token, stop.
3. If it is an ordinary token, append and process it as in standard autoregressive decoding, then return to step 2.
4. If it is <copy>, compute p_s from the current state and select $\hat{s} = \arg\max_i p_s(i)$.
5. Process the control token with the enhanced embedding $\widetilde{e}_c(\hat{s})$, obtaining $h_c(\hat{s})$ and its cache update.
6. Compute the endpoint distribution, applying the source and ordering masks, and select $\hat{e} = \arg\max_i p_e(i)$.
7. Append $(x_{\hat{s}}, \ldots, x_{\hat{e}})$, prefill these tokens to update the cache, and return to step 2 with the resulting decoding state.
8. Remove control tokens when converting the internal sequence into the final output text.

Source-only copying rules out selecting spans from the generated prefix. It does not remove the source or copied content from the context. Repeated use of a source region is distinct from recursively using generated output as the copy source; the source mask excludes the latter. The added pointer computations, source-state storage, enhanced-token step, and span prefill are all part of inference and are included in the serving measurements reported in Section 5.4 and Appendix E.

B Training Data and Copy-Trace Construction

B.1 Data Composition

Both backbones use the same tokenizer and share the SFT dataset, which combines four code editing sources with generation examples from OpenCodeInstruct. Table 5 reports the retained mixture. Supervised-token counts include ordinary generated tokens and copy control tokens, but exclude tokens inserted by copying. Therefore, they measure LM-supervised positions rather than the total number of tokens processed by the Transformer.

Table 5: **Shared SFT data composition for 4B and 30B.** Shares are percentages. Counts are integers; the token columns count LM-supervised positions only.

Source	Examples	Share	Supervised tokens	Token share
CommitPackFT	1,432	6.33	207,828	5.20
InstructCoder	9,589	42.40	986,765	24.67
OCEDataFT	5,546	24.52	1,184,502	29.61
NextCoderDataset	3,243	14.34	1,051,439	26.29
OpenCodeInstruct	2,804	12.40	569,337	14.23
Total	22,614	100.00	3,999,871	100.00

Editing examples with overly long sequences, low copying ratios, or excessively complex trajectories are filtered. For NextCoderDataset, filtering additionally limits the total number of actions, copy actions, and generated tokens to avoid highly fragmented traces. OpenCodeInstruct samples are selected for quality while approximately preserving the balance between algorithmic and general coding tasks and between self-instruct and evol-instruct examples.

Table 6 summarizes the retained editing traces. The copy ratio is the corpus-level fraction of target tokens supplied by copy actions; it is not the baseline-relative autoregressive token ratio in Equation 9. NextCoderDataset provides the longest source examples and the most complex traces among these sources.

Table 6: **Shared editing-trace statistics for 4B and 30B.** Copy ratio is a percentage. Action counts include both copying and generation; P90 is the 90th percentile of the total action count.

Source	Mean source length	Copy ratio	Mean actions	P90	Mean copies
CommitPackFT	216.16	38.36	5.75	10	2.72
InstructCoder	109.03	31.23	4.22	8	1.77
OCEDataFT	206.19	28.37	6.05	10	2.86
NextCoderDataset	476.45	40.68	10.92	18	5.32
Overall	204.12	34.01	5.94	11	2.73

B.2 AST-Aligned Candidates and Matching

Let x^{code} denote the old code and z the target code. A trajectory $\tau = (a_1, \ldots, a_m)$ consists of actions

$$a_j \in \{\text{copy}(s_j, e_j), \text{generate}(u_j)\}, \qquad \text{realize}(\tau, x^{\text{code}}) = z. \tag{13}$$

At the text-matching stage, candidate spans are represented by character offsets; the final labels use token positions in the complete prompt.

We parse the old code with the Python AST and extract candidate spans for functions, asynchronous functions, classes, and compound statements such as `if`, `for`, `while`, `with`, `try`, and `match`. Top-level simple statements, including imports and assignments, are also candidates. Very short spans are excluded to discourage fragmented copying. Each candidate retains its exact source text, rather than a reformatted AST rendering.

Candidates are ordered by text length. Scanning the target from left to right, we choose the longest eligible candidate matching the current position. Text before the next eligible match is assigned

to generation. Adjacent actions are consolidated where doing so preserves the trajectory's exact realization; a single copy action must still refer to a contiguous source span. Blocks containing changes are generated rather than forced into an unchanged-block match. This procedure uses the target only when synthesizing supervision, not when running the trained model.

B.3 Token Alignment and Loss Masking

Character-level matches are mapped to the model tokenizer's positions in the full prompt, accounting for the instruction and source-code placement. Each copy event records the source start and end labels. Its control token and generated segments receive LM supervision, whereas copied tokens receive no LM loss. The copied tokens nevertheless remain in the training context so that later predictions observe the same realized content available during inference.

Every retained trajectory undergoes an exact reconstruction check: removing the control tokens from the realized sequence must reproduce the target code. This check is essential because code correctness depends on details such as indentation and string contents, and because text matches must remain valid after token alignment. Line-level traces used in the 4B ablation replace the AST candidate units with code lines; both main model variants use the shared block-level traces.

C Training Configuration and Preference Data

C.1 Supervised Fine-Tuning Configuration

The implementation uses PyTorch and Hugging Face Transformers. Most backbone parameters are frozen, and LoRA updates the backbone while the added control-token representations and pointer parameters are trained. The 4B SFT run uses four GPUs, a per-device batch size of 1, and four gradient-accumulation steps, giving an effective batch size of 16. It runs for one epoch with cosine learning-rate scheduling. Table 7 gives the 4B optimization settings, and Table 8 gives the 30B settings for the two training stages.

Table 7: **4B training settings.** The DPO stage starts from the 4B block-level SFT model.

Setting	SFT	DPO
Learning rate	2×10^{-4}	1×10^{-5}
Per-device batch size	1	1
Gradient accumulation	4	8
LoRA rank	16	16
LoRA scaling factor	32	32
LoRA dropout	0.05	0.05
Warmup ratio	0.05	0.05
Precision	bf16	bf16
Gradient checkpointing	Enabled	Enabled

For the 4B backbone, LoRA targets `q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, and `down_proj`, with rank 16, scaling factor 32, and dropout 0.05. The 30B backbone uses three LoRA parameter groups: the attention projections `q_proj`, `k_proj`, `v_proj`, and `o_proj` use rank 16 and scaling factor 32; the packed expert parameters `mlp.experts.gate_up_proj` and `mlp.experts.down_proj` use rank 2 and scaling factor 4; and the router `mlp.gate` uses rank 4, scaling factor 8, and zero dropout.

The 30B SFT run uses two GPUs with one example per device and eight gradient-accumulation steps, yielding the effective batch size of 16. It uses BF16, FSDP2 sharding, non-reentrant gradient checkpointing, 5% warmup followed by cosine decay, zero weight decay, and gradient clipping at 1.0. The trainer applies no additional sequence truncation; the longest tokenized sequence in the retained SFT data contains 2,043 tokens. Its SFT objective adds a router auxiliary loss with weight 0.001 to the language-model and pointer losses in Equation 8.

The language-model term supervises ordinary generated tokens and `<copy>`. The start and end losses are applied only at copy events, with weights $\lambda_s = \lambda_e = 0.50$. Pure generation trajectories

Table 8: **30B training settings.** Differential learning rates apply to SFT.

Setting	SFT	DPO
Epochs	1	1
Per-device microbatch	1	1
Effective batch size	16	16
Learning rate	2×10^{-5} (attention/expert) 2×10^{-6} (router)	1×10^{-5}
DPO β	–	0.10
Chosen-SFT coefficient α	–	0.05

have no pointer events and train only the generation path. Token masking reduces the number of LM-supervised targets but does not remove copied positions from the Transformer computation.

C.2 Preference Data

Editing pairs. Editing preference prompts are selected from CommitPackFT, InstructCoder, OCE-DataFT, and NextCoderDataset, excluding examples used in SFT. We sample multiple candidate responses with temperature 0.80 and retain prompts for which the SFT model produces both correct and incorrect candidates. Since the four sources do not share a unified execution evaluator, DeepSeek V4 Flash judges whether a candidate satisfies the edit instruction. The resulting preference datasets contain 1,000 editing pairs for PG-LLM 4B and 2,000 for PG-LLM 30B, preferring judged-correct trajectories over judged-incorrect ones. Errors may arise from incorrect copy spans or from the newly generated edits, so the preference objective can affect both selection and generation.

Generation pairs. Generation preference prompts come from OpenCodeInstruct, the same source as the SFT generation data, but from a non-overlapping set of examples. We select prompts for which the base model produces a response that passes the available tests while the SFT model's response fails. The base response becomes the preferred pure-generation trajectory and the SFT response the rejected trajectory, yielding 1,500 generation pairs for PG-LLM 4B and 3,000 for PG-LLM 30B. These prompts are not selected from LiveCodeBench, BigCodeBench-Hard, or MBPP+. The construction supervises recovery of useful generation behavior without requiring a copy action in the preferred response.

C.3 Trajectory Probability and DPO Objective

Let $\mathcal{G}(\tau)$ include vocabulary-decoded positions, both ordinary tokens and copy control tokens, and let $\mathcal{C}(\tau)$ denote copy events. The log probability of a trajectory is

$$\log \pi_\theta(\tau \mid x) = \sum_{t \in \mathcal{G}(\tau)} \log p_{\mathrm{LM},\theta}(y_t \mid x, y_{<t}) + \sum_{j \in \mathcal{C}(\tau)} \left[\log p_{s,\theta}(s_j \mid x, y_{<t_j}, c) + \log p_{e,\theta}(e_j \mid x, y_{<t_j}, c, s_j)\right]. \tag{14}$$

The copied tokens are a deterministic consequence of the selected span and are not counted again as vocabulary predictions. The start used to evaluate each endpoint is the start specified by that trajectory.

With preferred and rejected trajectories τ^+ and τ^-, the fixed reference policy π_{ref} is the corresponding backbone's SFT model. We minimize

$$\Delta_\theta(x, \tau^+, \tau^-) = \log \frac{\pi_\theta(\tau^+ \mid x)}{\pi_\theta(\tau^- \mid x)} - \log \frac{\pi_{\mathrm{ref}}(\tau^+ \mid x)}{\pi_{\mathrm{ref}}(\tau^- \mid x)}, \tag{15}$$

$$\mathcal{L}_{\mathrm{DPO}} = -\log \sigma\big(\beta \Delta_\theta(x, \tau^+, \tau^-)\big), \tag{16}$$

$$\mathcal{L}_{\mathrm{preference}} = \mathcal{L}_{\mathrm{DPO}} + \alpha \mathcal{L}_{\mathrm{SFT}}(\tau^+). \tag{17}$$

Both the 4B and 30B DPO runs use $\beta = 0.10$ and $\alpha = 0.05$. The 30B DPO run trains for one epoch with a per-device microbatch of 1, an effective batch size of 16, and a learning rate of 1×10^{-5}.

Reference log probabilities for both trajectories are computed offline before training, avoiding an additional reference-model copy in training memory. The 4B DPO run uses a maximum sequence length of 4,096 and at most 256 copy events per sequence, together with the settings in Table 7.

D Evaluation Protocols and Metrics

D.1 Tasks and Evaluation Subsets

The editing task takes a natural-language instruction and old code and requires complete revised code. None of the evaluated models is asked to produce a patch or a difference file. Within each benchmark, both base models and all four PG-LLM checkpoints share the input prompt, chat template, decoding configuration, and evaluation script. Table 9 records the evaluated subset sizes and evaluation methods.

Table 9: **Evaluation subsets shared by both backbones.** Counts denote the number of evaluated examples in each benchmark. Generation metrics are single-response Pass@1.

Task	Dataset	Examples	Evaluation
Editing	CommitPackFT held-out	213	DeepSeek V4 Flash judge
Editing	CanItEdit	105	Official functional tests
Editing	EditEval	194	Local functional test harness
Generation	MBPP+	378	EvalPlus
Generation	LiveCodeBench v6	131	Official execution evaluator
Generation	BigCodeBench-Hard	148	Functional tests

The CommitPackFT subset is held out from the training data and shares its source distribution. We report judge accuracy for this subset, functional correctness under the official CanItEdit tests, and functional correctness under the local EditEval test harness.

LiveCodeBench evaluation covers 131 tasks from February through May 2025 in the v6 evaluation selection. BigCodeBench-Hard is the difficult subset used in the experiments, and MBPP+ is evaluated with EvalPlus under the test protocol specified in Table 2. One response is generated per example. We report benchmark scores as descriptive point estimates for the evaluated checkpoints.

D.2 Token Accounting and Aggregation

For editing example i, write G_i for the number of ordinary vocabulary-generated code tokens and C_i for the number of copy control tokens. Under the one-control-token design,

$$D_i^{\mathrm{PG}} = G_i + C_i, \qquad C_i = \text{number of copy actions.} \tag{18}$$

Copied-span tokens do not enter D_i^{PG}, although they are included in the model's realized context and output. The baseline-relative autoregressive token ratio is computed from the summed counts in Equation 9, using the base model of the same backbone as the denominator. This avoids giving very short examples disproportionate influence through an average of per-example ratios.

For each model, the editing average in Table 1 is the arithmetic mean of the three separately computed dataset scores:

$$\overline{R}_{\mathrm{tok}} = \frac{1}{3}\sum_{b=1}^{3} R_{\mathrm{tok},b}. \tag{19}$$

The same unweighted averaging is used for accuracy and generation Pass@1, without pooling the two backbones. For PG-LLM 30B DPO, averaging the displayed editing ratios gives $(0.65 + 0.66 + 0.82)/3 = 0.71\times$ after rounding. Editing accuracy improves from 73.37% to 75.01%, an absolute increase of 1.64 percentage points, not a relative increase of 1.64%. The corresponding 4B DPO results are an average token ratio of $0.70\times$ and a 2.44-percentage-point accuracy gain.

The ratio compares the autoregressive token counts of each model's own outputs, whose lengths and contents can differ. We report it alongside editing correctness to characterize the accuracy–efficiency tradeoff.

D.3 Copy Triggers on Generation Tasks

For a generation benchmark with N examples, the trigger rate is

$$R_{\text{trigger}} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}[\text{at least one copy action occurs on example } i]. \tag{20}$$

This example-level rate measures how often the model invokes copying on generation tasks. We report it alongside functional correctness to characterize reuse behavior and task performance.

E vLLM Integration and Serving Evaluation

E.1 Engine Integration

We export the 4B and 30B DPO checkpoints, including their LoRA and pointer parameters, as custom Qwen3 models for vLLM 0.11.0. During prompt prefill, the implementation retains eligible source states and their pointer keys for each request. When the LM head emits `<copy>`, the runner uses the preceding decode state to select the start, processes the control token with the start-conditioned embedding in Equation 4, and selects the end. It then writes the selected source token IDs into the request sequence; vLLM prefills these tokens into the KV cache before ordinary decoding resumes. Pointer state and LM-decode budgets are maintained per request so that this procedure remains compatible with continuous batching. Model registration and runner hooks are loaded in the V1 EngineCore subprocess.

E.2 Benchmark Configuration

All routes in Table 3 use the same prompts, greedy decoding settings, generation limits, and NVIDIA RTX PRO 5000 72GB Blackwell GPUs. The 4B routes run on one GPU, while the 30B routes run on two GPUs because the model does not fit in the memory of a single device; all other reported serving settings are shared across scales. CommitPackFT, CanItEdit, and EditEval contain 213, 105, and 194 requests, respectively. Results are means over three runs and exclude model loading and initialization. vLLM uses BF16, the PyTorch sampler, `max_num_seqs=64`, and `max_num_batched_tokens=8192`. The native base models use CUDA Graphs, whereas the custom pointer paths currently run eagerly. Prefix caching is disabled because source hidden states are required from the beginning of each prompt. The Transformers routes are sequential reference implementations; the vLLM routes use batched EngineCore scheduling.

For serving robustness, EditEval pointer runs stop a request if it selects the same source span eight consecutive times and attempts to copy it again. This guard addresses a pathological repeated-copy loop and is not part of the learned architecture. Throughput is the number of completed requests divided by generation wall time, and speedup is normalized to the corresponding backbone's vLLM base throughput on the same dataset. Token denotes the autoregressive token ratio in Equation 9; copied tokens are prefilled into the cache but are not counted as autoregressive LM decisions. Because the two scales use different GPU counts, their absolute throughput values should not be interpreted as a direct cross-scale comparison.

E.3 Figure 1 Single-Example Measurement

Figure 1 uses an edit that adds an `ASeq` class. The base model and PG-LLM produce the same 1,280-token final code, while PG-LLM supplies 1,164 tokens through two copy actions and generates the remaining 116 code tokens. Including the two copy-control tokens, its autoregressive count is 118 tokens, a 90.8% reduction from the base model's 1,280 tokens. Table 10 reports the complete token and timing measurements.

Both systems are measured with batch size 1 on the same NVIDIA RTX PRO 5000 72GB GPU using vLLM 0.11.0, BF16, and an 80% GPU-memory budget. Each model is loaded twice; each load performs two warm-up calls followed by five timed calls, yielding ten measurements per model. The base model uses CUDA Graphs, whereas PG-LLM runs eagerly. Each timing covers one `LLM.generate` call, including prompt prefill, autoregressive decoding, and pointer-based copy

Table 10: **Token and latency measurements for Figure 1.** Both systems produce identical final code. Timing values are seconds.

Metric	Base	PG-LLM
Final code tokens	1,280	1,280
Copied code tokens	0	1,164
Generated code tokens	1,280	116
Copy-control tokens	0	2
Autoregressive tokens	1,280	118
Median generation time	11.8079	1.6571
Generation time P10–P90	11.4871–12.7157	1.4864–2.0362

processing, while excluding model loading and warm-up. The $7.13\times$ speedup is the ratio of the unrounded median times, $11.8079/1.6571$.

E.4 Implementation Scope

The serving study evaluates two model scales on three code editing workloads using one GPU type. The pointer path uses eager execution with prefix caching disabled. Its throughput gains depend on copy frequency, span length, and the number of copy actions. BF16 batch scheduling can also produce small greedy-decoding variations. The integration targets vLLM 0.11.0 through version-specific EngineCore hooks; extending it to future vLLM releases requires compatibility and correctness testing.

F Additional Results and Training Diagnostics

F.1 Per-Dataset Copy-Granularity Ablations

Tables 11 and 12 expand the averages in Table 4. We compare line-level and AST-aligned block-level traces using 4B SFT models.

Table 11: **4B editing ablations by dataset.** Accuracy is a percentage, and Token is the relative autoregressive token count. Arrows in column headers indicate the preferred direction. Bold indicates the best score within this comparison, including the base model.

	CommitPackFT		CanItEdit		EditEval		Avg.	
Model	Acc. ↑	Token ↓	Acc. ↑	Token ↓	Acc. ↑	Token ↓	Acc. ↑	Token ↓
Qwen3-4B-Instruct-2507	70.42	1.00×	52.38	1.00×	**60.31**	1.00×	61.04	1.00×
PG-LLM 4B SFT (line-level)	67.14	**0.47**×	51.43	**0.52**×	55.67	**0.73**×	58.08	**0.57**×
PG-LLM 4B SFT (block-level)	**71.83**	0.61×	**54.29**	0.62×	57.22	0.80×	**61.11**	0.68×

Table 12: **4B generation ablations by dataset.** Pass@1 and copy-trigger rates are percentages. Arrows in column headers indicate the preferred direction. Avg. averages Pass@1. Bold marks the highest Pass@1 in each column.

	LiveCodeBench		BigCodeBench-Hard		MBPP+		Avg.
Model	Pass@1 ↑	Trigger ↓	Pass@1 ↑	Trigger ↓	Pass@1 ↑	Trigger ↓	Pass@1 ↑
Qwen3-4B-Instruct-2507	**25.19**	–	**22.97**	–	**78.04**	–	**42.07**
PG-LLM 4B SFT (line-level)	12.21	19.08	14.19	16.89	74.87	0.00	33.76
PG-LLM 4B SFT (block-level)	21.37	17.56	18.24	16.22	74.60	0.00	38.07

On the 4B backbone, line-level traces yield lower token ratios on all three editing datasets, but lower editing accuracy than block-level traces. The generation difference is largest on LiveCodeBench. These results support choosing structurally aligned traces for a better correctness–efficiency balance.

F.2 Supervised Training Dynamics

The SFT and DPO curves below report training-set metrics.

Figure 3 shows the 4B SFT loss and gradient norm. Loss decreases over training, and the early gradient fluctuations settle as optimization progresses.

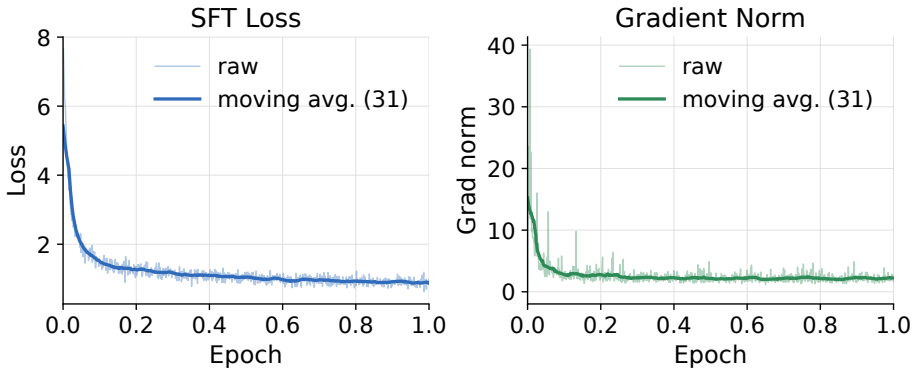


Figure 3: **4B SFT optimization diagnostics.** Training loss and gradient norm with moving-average smoothing.

Figure 4 separates 4B SFT LM-head accuracy from copy-start and copy-end accuracy. Start accuracy rises more rapidly and reaches above 90.00% in the training trace, whereas end prediction remains substantially less accurate. Endpoint selection must account for where reuse should stop, and during inference it also depends on the predicted start. This asymmetry identifies end prediction as a practical bottleneck.

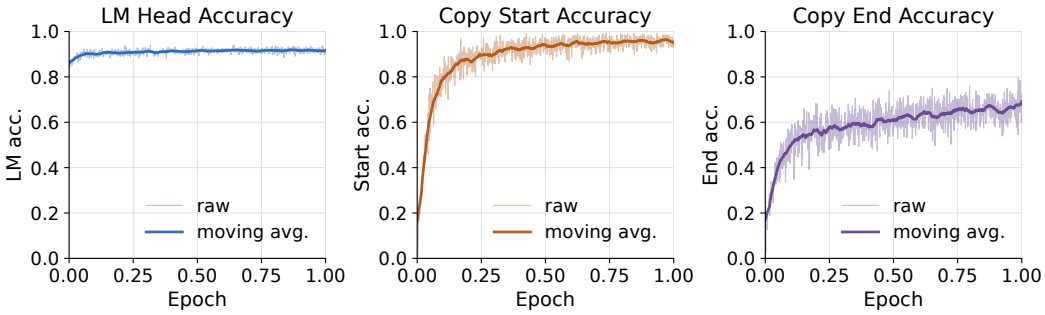


Figure 4: **4B SFT training accuracies.** LM-head accuracy, copy-start accuracy, and copy-end accuracy. Copy-end prediction is the weaker boundary component.

F.3 Preference Optimization Dynamics

Figure 5 shows the 4B DPO loss, preference accuracy, and preference margin. The plotted loss is $\mathcal{L}_{\text{DPO}}$ without the auxiliary supervised term. Preference accuracy measures the fraction of pairs for which the model improves the preferred–rejected log-probability difference relative to the reference; the margin is Δ_θ defined in Appendix C.2. Together, the preference accuracy and margin track the model's increasing separation of preferred and rejected training trajectories.

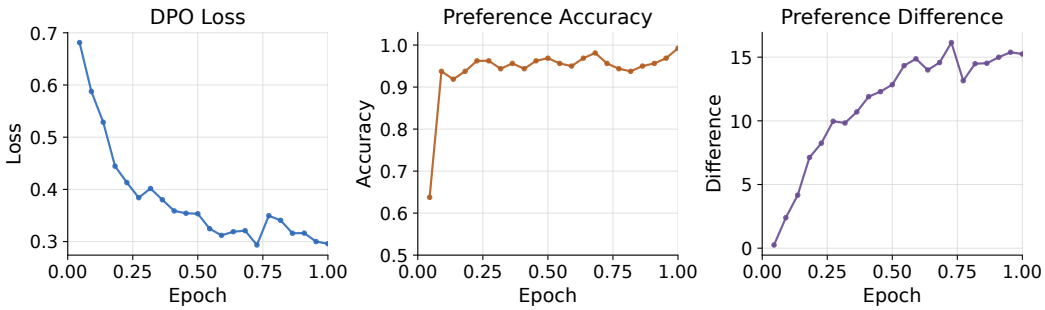


Figure 5: **4B DPO training dynamics.** Preference loss decreases as preference accuracy and the reference-relative preference margin increase. These quantities use both vocabulary and pointer probabilities in each trajectory.

G A Qualitative Example for Code Editing

G.1 Task and Original Code

We examine a successful CanItEdit example, `LLMInference`, produced by PG-LLM 4B DPO. The original program provides a Flask inference service backed by `vllm`. The edit asks for defensive validation of the `conversation` field: it must be present, be a non-empty list of strings, and contain an odd number of messages. Most of the service—imports, model loading, inference configuration, prompt construction, and routing—does not require modification.

The green boxes below show unchanged source regions available for reuse; the blue box shows the original region to edit. The replacement is shown in a separate blue box afterward. These display regions summarize reusable versus modified content and are not a one-to-one annotation of the model's three pointer events.

Original code: reusable prefix

```
from flask import Flask, request, jsonify
from threading import Lock
from vllm import LLM, SamplingParams

HUMAN_HEADER = "Question:"
AI_HEADER = "Answer:"


class Inferencer:
    def __init__(self, model_name):
        self.model_name = model_name
        self.model_lock = Lock()
        self.model = None

    def get_model(self):
        if self.model is None:
            self.model = LLM(self.model_name)
        return self.model
```

Original code: region requiring modification

```
    def predict_from_json(self, inputs):
        if inputs is None:
            return jsonify({"error": "no json provided"})

        convo = inputs['conversation']
```

Original code: reusable suffix

```
        max_tokens = inputs.get('max_tokens', (len(inputs) * 3) + 1024)
        temperature = inputs.get('temperature', 0.4)
        top_p = inputs.get('top_p', 0.9)
        n = inputs.get('n', 1)

        with self.model_lock:
            model = self.get_model()
            params = SamplingParams(
                max_tokens=max_tokens, temperature=temperature, top_p=
    ↪ top_p, stop=[
                    HUMAN_HEADER]
            )
            prompt = ""
            for i, text in enumerate(convo):
```

```
                if i % 2 == 0:
                    prompt += f"{HUMAN_HEADER}\n{text}\n"
                else:
                    prompt += f"{AI_HEADER}\n{text}\n"

            prompt += f"{AI_HEADER}\n"

            result = model.generate(
                [prompt] * n, sampling_params=params,
            )
            outs = [x.outputs[0].text for x in result]

        return jsonify(outs)


app = Flask(__name__)
inferencer = Inferencer("bigcode/starcoder")


@app.route('/predict', methods=['POST'])
def predict():
    return inferencer.predict_from_json(request.json)
```

G.2 Generated Replacement

PG-LLM 4B DPO generates the following replacement for the blue region while reusing unchanged context. The resulting edit passes the CanItEdit functional test. It first checks that the request is a dictionary, then verifies the presence and type of `conversation`, and finally checks odd message count. An empty list is also rejected by the odd-count check. The green prefix and suffix remain unchanged; the final output is the complete code, not this replacement block alone.

Revised code: newly generated input validation

```
    def predict_from_json(self, inputs):
        if not isinstance(inputs, dict):
            return jsonify({"error": "no json provided"})

        if 'conversation' not in inputs:
            return jsonify({"error": "missing conversation key"})

        convo = inputs['conversation']
        if not isinstance(convo, list) or not all(isinstance(text, str)
↪  for text in convo):
            return jsonify({"error": "conversation must be a non-empty
↪ list of strings"})

        if len(convo) % 2 != 1:
            return jsonify({"error": "conversation must have an odd
↪ number of messages"})
```

G.3 Token Accounting

The source code contains 557 tokens. PG-LLM 4B DPO decodes 106 ordinary tokens and three copy control tokens, for 109 vocabulary-decoded tokens in total. Each of the three copy actions uses one control token. Table 13 summarizes the source-token and vocabulary-decoding counts.

Using the old-code length as a reference, the example's source-normalized token ratio is

$$R_{\text{source}} = \frac{106 + 3}{557} = 0.20\times \quad \text{(rounded)}. \tag{21}$$

Table 13: **Token accounting for the 4B DPO `LLMInference` example.** The final column reports the decoded-token count normalized by the source-code length.

Example	Source tokens	Generated	Control	Decoded total	R_{source}
LLMInference	557	106	3	109	0.20×

This source-normalized case statistic illustrates the small number of vocabulary decisions needed for the edit. It is distinct from the benchmark metric in Equation 9, whose denominator is the base model's actual decoded output.

This 4B DPO example illustrates the intended context localization and reuse behavior: the model reuses unchanged source material and concentrates autoregressive decoding on the requested modification. The example complements the aggregate results by showing how reuse and generation combine in a complete, functionally correct edit.