SWE-Pruner Pro: The Coder LLM Already Knows What to Prune

Yuhang Wang[1,*], **Yuling Shi**[1,*], **Shaoqiu Zhang**[1], **Jialiang Liang**[1], **Shilin He**[2]
Siyu Ye[2], **Yuting Chen**[1], **Kai Cai**[2], **Xiaodong Gu**[1,†]

[1]LLM4SE Lab, Shanghai Jiao Tong University
[2]Douyin Group

Abstract

Pruning long context for coding agents has been a vital technology for efficient context management. While existing context pruning methods such as SWE-Pruner realize this by attaching a separate code classifier, we find the agent itself encodes internal representations indicating the relevance of code context when reading tool output. Based on this finding, we propose SWE-Pruner Pro, which prunes tool outputs directly inside the agent. Concretely, a small head turns the agent's own internal representations into a keep-or-prune label for each line, with a length-aware embedding keyed to each tool output's line count. Across two open-weight backbones and four multi-turn benchmarks, SWE-Pruner Pro saves up to 39% of prompt and completion tokens while preserving task quality, with bounded inference overhead. Notably, on MiMo-V2-Flash SWE-Pruner Pro additionally raises the SWE-Bench Verified resolve rate by +3.8% and the long-context Oolong accuracy by +2.2 points.

Project Page: https://github.com/Ayanami1314/swe-pruner-pro

1 Introduction

Pruning long context for coding agents has been a vital technology for efficient context management Cheng et al. (2024); Shi et al. (2025a); Wang et al. (2026a). During programming tasks, coding agents accumulate very long tool outputs full of redundant information during multi-turn interaction with environments. They solve repository-level tasks by interleaving reasoning with environment interactions: they call tools like `cat`, `grep`, `ls`, and `python`, observe the raw textual output, and continue. In practice, the bulk of the per-trajectory token budget is spent on tool outputs, much of it repetitive or never re-referenced (Wang et al., 2026b). This redundant content accumulates across turns, inflating cost and triggering well-documented long-context degradation (Liu et al., 2023; Laban et al., 2025; Li et al., 2023a). As such, pruning the long context has been a natural way to keep the context manageable.

Two lines of prior work address this. The first, general-purpose prompt or code compression (Jiang et al., 2023; Li et al., 2023b; Shi et al., 2025a), scores tokens with a fixed metric such as perplexity or syntactic structure, and so cannot adapt to the agent's evolving focus during the task. The second, task-specific pruning, exemplified by SWE-Pruner (Wang et al., 2026b), does condition on the agent's intent, but at the cost of a second scoring model and an explicit goal-hint query the agent must write every turn.

Both approaches obtain the pruning signal from outside the agent, even though the agent's backbone model has already processed the tool output. We ask whether the information needed to decide which lines to keep or prune is already present in the model's own representations, motivating us to read the pruning signal directly from them rather than eliciting an additional goal-hint query (Wang et al., 2026b), as illustrated in

*Equal contribution.
†Corresponding author: xiaodong.gu@sjtu.edu.cn.

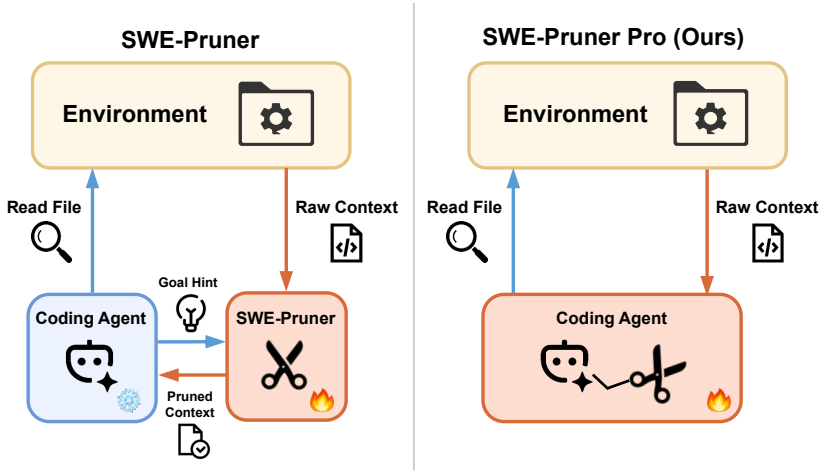

Figure 1 (Left) Prior work (Wang et al., 2026b) uses a separate Pruner model conditioned on an explicit goal-hint query the agent must write at every turn. (Right) SWE-Pruner Pro prunes directly from the agent's own internal representations, with no external scoring model.

Figure 1. Our probing study shows that kept and pruned lines are distinguishable in this representation space, with even a simple linear probe providing evidence that per-line pruning information is already there (§2).

We instantiate this idea as SWE-Pruner Pro, which turns the agent's own representations into line-level pruning decisions. When the agent reads a tool response, its frozen backbone already produces token representations during the normal prefill; SWE-Pruner Pro reuses these representations to predict which lines should be kept or pruned. A lightweight head makes these predictions, with two additions that matter in practice: length awareness, so the head can behave differently on short and long outputs, and a per-sample balanced focal loss, which rebalances each sample's keep and prune tokens equally, so the rarer the kept lines are, the more importance signal they carry into the objective. Because the head runs in-server on the existing prefill, it avoids an extra model call and adds only a small bounded cost.

Across two open-weight backbones and four multi-turn benchmarks, SWE-Pruner Pro is the most consistent pruner among the seven methods we evaluate. On the SWE-QA family and Oolong, it is the only method that reduces end-to-end token use in every setting while keeping task quality close to the unpruned agent. The savings reach 39% on SWE-QA-Pro and 30% on the long-context Oolong benchmark, where SWE-Pruner Pro also improves MiMo-V2-Flash accuracy by +2.2 points. On the coding-agent benchmark SWE-Bench Verified, SWE-Pruner Pro improves MiMo-V2-Flash's resolve rate by +3.8%. In our replay study, the in-server head adds only 15.0% aggregate wall time on top of the agent's total generation time, without requiring an extra model call.

Our contributions are:

- We show that an agent backbone's internal representations already encode line-level importance of tool outputs, removing the need for a separate scoring model or explicit query.
- We propose SWE-Pruner Pro, a lightweight head that reads the pruning signal directly from the backbone with a learned length-aware embedding and a per-sample balanced focal loss.
- We evaluate SWE-Pruner Pro on two open-weight backbones and four multi-turn benchmarks, where it saves up to 39% of tokens while preserving task quality.

2 Motivation

Multi-turn coding agents spend the bulk of their per-trajectory token budget on tool outputs. On SWE-Bench Verified, file-reading commands account for over 70% of the tokens consumed by Mini-SWE-Agent backed by Claude Sonnet 4.5, with a comparable pattern observed on GLM-4.6 (Wang et al., 2026b); this cost compounds across turns as earlier reads remain in the context window. Existing pruners reduce this cost by adding

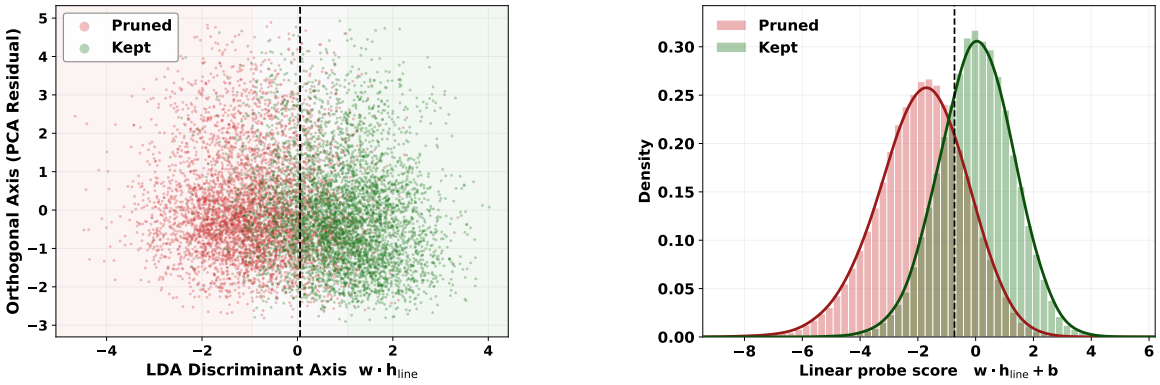

Figure 2 Linear probe on the agent backbone's frozen last-layer hidden states. (Left) Two-dimensional projection along the LDA discriminant axis and an orthogonal direction. (Right) Distribution of probe scores. The two classes have visibly different means but overlap in the middle band.

machinery on top of the agent. General-purpose compressors (Jiang et al., 2023; Li et al., 2023b; Shi et al., 2025a) score tokens by a fixed surrogate such as perplexity, with no view of the agent's task; task-specific pruners (Wang et al., 2026b) run a separate scoring model conditioned on an explicit goal-hint query the agent writes at every turn. Both routes treat the agent's current information need as a quantity to be reconstructed externally, rather than read directly from the backbone that is already processing the tool output.

Reading a tool output is itself an attention-weighted forward pass over its tokens, so the backbone must already encode which lines matter for the next action and which it can ignore. If so, the keep-or-prune signal should already be visible in its last-layer hidden states. To check this, we collect ≈ 2,260 multi-turn tool responses (≈ 155k lines) from publicly released SWE-Bench-style and terminal-task datasets, and label each line as keep or prune with Claude Sonnet 4.6; trajectory sources and the full labelling protocol are in Appendix A. We then freeze Qwen3-Coder-Next, mean-pool the last-layer hidden states of each line, and fit a logistic regression to see whether the two classes are separable.

The two classes have visibly different means in the hidden-state space: along the LDA discriminant axis their distributions are offset (Figure 2 left), and the per-line scores along the same direction confirm a clear mean gap with substantial overlap in the middle band (Figure 2 right). On the held-out trajectories, the probe reaches AUC 0.83 and best-F_1 0.63, well above the majority-class F_1 upper bound of 0.46 at the empirical positive rate of ≈30%. The keep-or-prune signal is therefore already inside the backbone's representation. A logistic regression alone cannot resolve the overlap in the mid-score region of Figure 2 (right) or exploit the length-dependent structure of tool outputs; both motivate SWE-Pruner Pro's non-linear head.

3 Method

SWE-Pruner Pro makes per-line keep-or-prune decisions on each tool response directly from the agent backbone's last-layer hidden states (Figure 3). The remainder of this section covers the per-turn pipeline (§3.1), the pruning head (§3.2), and training (§3.3).

3.1 Pipeline overview

At each turn t, the agent issues a tool call c_t and the environment returns a raw tool response r_t, often hundreds of lines long and persisting in the agent's context for the rest of the trajectory. Before the agent generates its next move, the backbone prefills $[H_{t-1}, c_t, r_t]$ into its KV cache (Figure 3 top), where H_{t-1} is the context history accumulated before turn t. The prefix $[H_{t-1}, c_t]$ is already in the KV cache, so only the new r_t tokens are forwarded, producing last-layer hidden states $h_1, \ldots, h_L$ over the r_t span, where $L = |r_t|$ in tokens.

SWE-Pruner Pro attaches at this prefill. The pruning head turns each h_i into a per-token keep-or-prune logit and aggregates them into per-line decisions (Figure 3 bottom). Line-level granularity preserves the

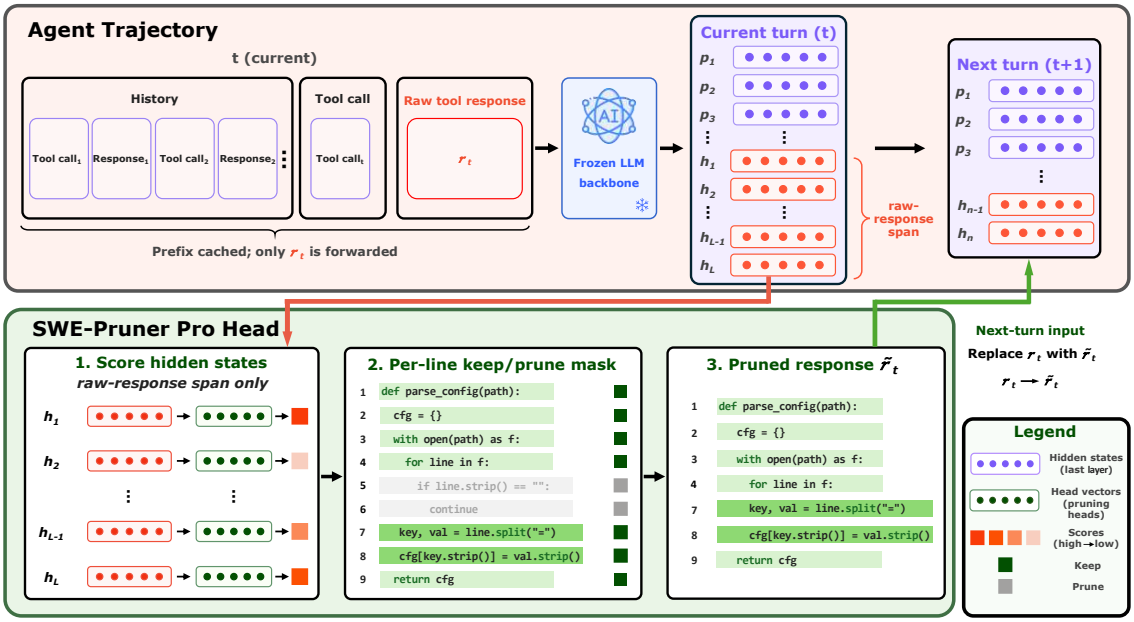

Figure 3 Overview of SWE-Pruner Pro. **Top (Agent Trajectory):** at each turn the agent backbone prefills $[\text{history}, c_t, r_t]$; only the new tool-response tokens are forwarded, and SWE-Pruner Pro reads their last-layer hidden states $\{h_i\}$ off this prefill at no extra forward on r_t. **Bottom (SWE-Pruner Pro head):** the head scores each h_i, aggregates the binarised scores into per-line keep-or-prune decisions, and the pruned response $\tilde{r}_t$ replaces r_t in the next turn.

syntactic structure of the kept code while still allowing fine-grained pruning, consistent with prior code-aware pruners (Shi et al., 2025a; Wang et al., 2026b). Pruning is applied between turns: the agent's own generation at turn t still attends to the full r_t, while $\tilde{r}_t$ replaces r_t when the trajectory continues into turn $t{+}1$. The head reads $\{h_i\}$ off the prefill the backbone already performs on r_t. The only added backbone work is a single re-forward of $\tilde{r}_t$ at the next turn; since $\tilde{r}_t$ is typically much shorter than r_t (mean labelled keep-rate $\approx$30%, Appendix A), this overhead is more than offset by the shorter context all subsequent generation must attend to. The full per-turn loop is given as pseudocode in Algorithm 1; engineering details for the in-server integration are in Appendix E.

3.2 Pruning head

Let N be the number of lines in r_t and let d denote the dimension of h_i. The head has two components: a length-aware embedding $\mathbf{e}(N) \in \mathbb{R}^d$ that conditions the keep decision on response length, and a per-token feed-forward classifier f_θ that maps each augmented hidden state to a keep logit.

Length-aware embedding. $\mathbf{e}(N)$ is a learned function of the line count N, broadcast-added to every hidden state before f_θ and zero-initialised so f_θ reduces to its length-agnostic limit at the start of training:

$$\tilde{h}_i = h_i + \mathbf{e}(N). \tag{1}$$

The embedding gives the head explicit access to N alongside the per-token hidden state, so its keep-or-prune mapping can vary with response length. This matters because the cost of mis-pruning is highly non-uniform in N: a few lines wrongly stripped from a 5-line response is catastrophic, while the same on a 300-line response is negligible. The length-aware embedding lets the head absorb this dependence rather than relying on a single length-agnostic mapping for all lengths.

Per-token classifier. f_θ is a small non-linear network rather than a single linear layer: the linear-probe evidence in §2 already shows the signal is largely decodable from h_i, but a linear classifier alone cannot close the mid-score overlap (Figure 2) or interact with the length-aware embedding of Eq. 1; both gaps motivate

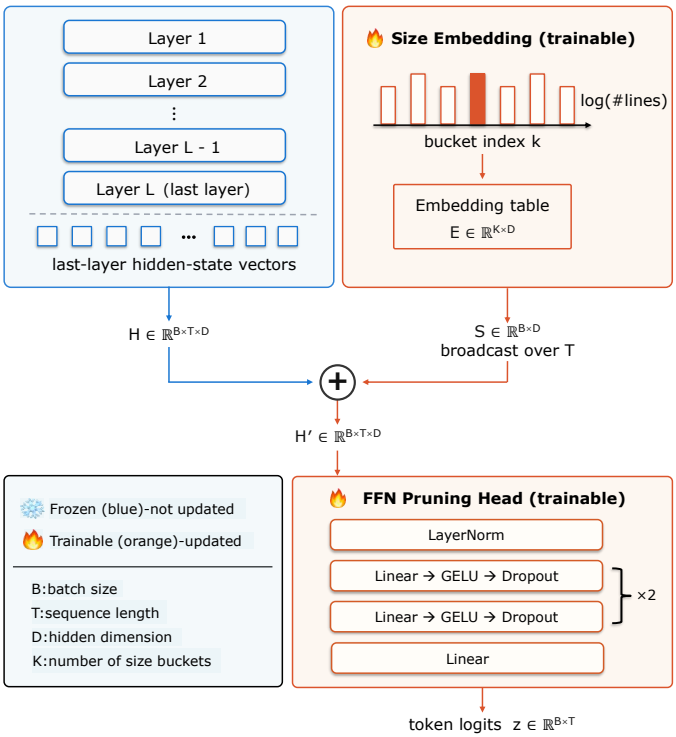

Figure 4 Head architecture. The frozen backbone's hidden states h_i receive an additive length-aware embedding indexed by the line count N, then pass through a two-block feed-forward stack with LayerNorm and dropout to a per-token keep logit. Line decisions are produced by majority vote within each line at inference.

the small non-linear head. Concretely, f_θ is a per-token feed-forward classifier: LayerNorm, two hidden Linear-GELU-Dropout blocks with hidden width d matching the backbone's hidden size, and a final Linear projection to a single logit. Applied to every augmented hidden state, it produces keep probabilities:

$$z_i = f_\theta(\tilde{h}_i), \qquad p_i = \sigma(z_i). \tag{2}$$

Line-level decision. With per-token sigmoid threshold τ (we use $\tau = 0.5$), per-line decisions $\hat{y}_\ell$ are obtained by majority vote of the binarized token decisions within each line ℓ:

$$\hat{y}_\ell = \mathbb{1}\left[\frac{1}{|\ell|}\sum_{i\in\ell}\mathbb{1}[p_i > \tau] > \frac{1}{2}\right]. \tag{3}$$

Deferring aggregation to inference time allows the head to be trained from per-token labels expanded directly from per-line annotations. Lines with $\hat{y}_\ell = 0$ are removed from the tool response before it is appended to the agent's history.

3.3 Training

We train the head on 22,609 samples from real multi-turn agent trajectories, each annotated by Claude Sonnet 4.6 with per-line keep-or-prune labels. We expand line labels to per-token labels for the loss, retaining uncertain rows as positives following the labelling protocol in Appendix A.

The loss is a per-sample balanced focal loss. Per-line keep/prune labels from an LLM annotator are inherently fuzzy, but the *ratio* the annotator settles on is itself informative: when only 3 of 100 lines are kept, those 3 lines carry the strongest signal about what is irreplaceable; when 90 of 100 are kept, the 10 pruned lines carry the strongest signal about what is safely removable. Cross-entropy and batch-level focal loss see only the global keep rate and treat every token equally, so they overfit the typical keep rate (≈30%) and dilute exactly the extreme-ratio samples whose labels carry the most importance. Where focal sharpens the loss around

Table 1 End-to-end quality and token consumption on the read-only multi-turn benchmarks.

	SWE-QA		SWE-QA-Pro		Oolong	
Method	**Score**	**Tokens**	**Score**	**Tokens**	**Acc.**	**Tokens**
Qwen3-Coder-Next						
No Pruning	7.71	590K	7.60	607K	81.7	3.6K
LLMLingua2	7.06 ↓0.65	**363K** ↓38.5%	7.02 ↓0.58	381K ↓37.3%	74.6 ↓7.1	9.5K ↑163.9%
Selective Context	7.57 ↓0.14	449K ↓23.9%	7.32 ↓0.28	463K ↓23.7%	**81.0** ↓0.7	12.0K ↑233.3%
RAG	**7.78** ↑0.07	549K ↓6.9%	7.73 ↑0.13	590K ↓2.8%	79.1 ↓2.6	4.5K ↑25.0%
Self-Prune	7.16 ↓0.55	496K ↓15.9%	6.96 ↓0.64	536K ↓11.7%	79.6 ↓2.1	3.8K ↑5.6%
LongCodeZip	7.45 ↓0.26	531K ↓10.0%	7.34 ↓0.26	593K ↓2.3%	80.1 ↓1.6	3.7K ↑2.8%
SWE-Pruner	7.33 ↓0.38	397K ↓32.7%	7.36 ↓0.24	433K ↓28.7%	79.7 ↓2.0	3.6K
SWE-Pruner Pro	7.73 ↑0.02	**385K** ↓34.7%	**7.84** ↑0.24	**368K** ↓39.4%	80.3 ↓1.4	**3.1K** ↓13.9%
MiMo-V2-Flash						
No Pruning	8.02	321K	7.97	438K	92.4	58.9K
LLMLingua2	7.73 ↓0.29	422K ↑31.5%	7.47 ↓0.50	423K ↓3.4%	87.1 ↓5.3	170.7K ↑189.8%
Selective Context	8.06 ↑0.04	360K ↑12.1%	7.94 ↓0.03	493K ↑12.6%	91.8 ↓0.6	47.5K ↓19.4%
RAG	8.05 ↑0.03	326K ↑1.6%	**7.97**	482K ↑10.0%	92.5 ↑0.1	47.9K ↓18.7%
Self-Prune	7.75 ↓0.27	330K ↑2.8%	7.29 ↓0.68	466K ↑6.4%	91.1 ↓1.3	48.6K ↓17.5%
LongCodeZip	8.13 ↑0.11	364K ↑13.4%	7.93 ↓0.04	488K ↑11.4%	90.7 ↓1.7	46.8K ↓20.5%
SWE-Pruner	**8.20** ↑0.18	303K ↓5.6%	7.94 ↓0.03	417K ↓4.8%	92.1 ↓0.3	52.5K ↓10.9%
SWE-Pruner Pro	7.98 ↓0.04	**299K** ↓6.9%	7.86 ↓0.11	**339K** ↓22.6%	**94.6** ↑2.2	**41.2K** ↓30.1%

Arrows report changes relative to the corresponding *No Pruning* baseline. Green denotes a favorable change and red an unfavorable change. Bold values are best among pruning methods; shaded rows denote SWE-Pruner Pro.

hard decision boundaries, our per-sample balancing instead protects the recall of each sample's minority class: we compute the loss separately over keep and prune tokens within a sample and average the two with equal weight, so each sample contributes equally from each of its classes regardless of its own keep rate.

For each token i in sample s, let $p_{t,i} = p_i$ if $y_i = 1$ and $p_{t,i} = 1 - p_i$ otherwise; then:

$$\mathcal{L}_i^{\text{tok}} = (1 - p_{t,i})^{\gamma} \cdot \text{BCE}(p_i, y_i), \quad \gamma = 2. \tag{4}$$

We average separately over keep and prune tokens within each sample:

$$\mathcal{L}_s^{\text{keep}} = \frac{\sum_i y_i \, \mathcal{L}_i^{\text{tok}}}{\sum_i y_i}, \quad \mathcal{L}_s^{\text{prune}} = \frac{\sum_i (1 - y_i) \, \mathcal{L}_i^{\text{tok}}}{\sum_i (1 - y_i)}, \tag{5}$$

and combine with equal weights:

$$\mathcal{L}_s = \tfrac{1}{2} \, \mathcal{L}_s^{\text{keep}} + \tfrac{1}{2} \, \mathcal{L}_s^{\text{prune}}. \tag{6}$$

The final objective is $\mathcal{L} = \frac{1}{S} \sum_s \mathcal{L}_s$.

The backbone is fully frozen, so adding the head does not alter the agent's general behaviour and requires no retraining or fine-tuning of the backbone itself; we train the head from cached features, with the full training configuration in Appendix B.

4 Experiments

Benchmarks. We evaluate on four benchmarks: *SWE-Bench Verified* (Jimenez et al., 2024) (500 patch-generation issues), *SWE-QA* (Peng et al., 2026) (144 multi-turn QA questions), *SWE-QA-Pro* (Cai et al., 2026) (260 QA questions with executable environments), and *Oolong* (Bertsch et al., 2025) (280 long-context aggregation instances, run as a multi-turn agent variant; Appendix C). SWE-Bench Verified uses the standard Mini-SWE-Agent harness;[1] the other three use a minimal bash-only agent (Table 1). SWE-Pruner Pro prunes tool outputs throughout.

[1] https://github.com/SWE-agent/mini-swe-agent

Agent backbones. We use two open-weight Mixture-of-Experts LLMs designed for long-horizon coding-agent workloads. MiMo-V2-Flash (Xiaomi LLM-Core Team, 2026) is a 309B-parameter MoE (15B active per token) with a hybrid sliding-window plus global attention architecture and a 256K context length. Qwen3-Coder-Next (Qwen Team, 2026) is a more compact 80B-parameter MoE (3B active) built on Qwen3-Next, agent-specialized via large-scale execution-feedback training, with the same 256K context length. Both backbones are served on our patched SGLang stack (Appendix E.1).

Baselines. The unpruned agent (*No Pruning*) is the headline reference. For pruner-vs-pruner comparison on the SWE-QA family and Oolong (Table 1), we run six prior pruners under the matched configuration, swapping only the pruning module: *LLMLingua2* (Pan et al., 2024), *Selective Context* (Li et al., 2023b), *RAG* (sliding-window retrieval with bge-reranker-v2-m3[2]), *Self-Prune* (the same agent backbone re-prompted on each r_t with our line-keep labelling prompt), *LongCodeZip* (Shi et al., 2025a) (coarse-only function-level perplexity ranking), and *SWE-Pruner* (Wang et al., 2026b), the closest prior task-specific pruner.

Metrics and judges. For SWE-Bench Verified we report the standard Resolve Rate (Jimenez et al., 2024). For SWE-QA and SWE-QA-Pro we report mean LLM-judge score on a 1–10 rubric, using GPT-5.4-mini as the judge. For Oolong we use the rule-based exact-match scorer and report accuracy on a 0–100 scale. The annotation and judge prompts are listed in Appendix H. Across all benchmarks we additionally report total prompt and completion tokens aggregated across all instances, and, where applicable, the number of pruning invocations triggered.

Inference configuration. Each backbone decodes under the official settings recommended on its HuggingFace model card. Agent rollouts use a maximum of 250 turns per trajectory; trajectories exceeding this are terminated. The judge GPT-5.4-mini runs at temperature 0 for determinism. All pruners on the same backbone share identical decoding, harness, and hardware, so pruning is the only varying factor; full serving details are in Appendix E.1.

5 Results

5.1 Main Results

Read-only multi-turn benchmarks. Pruning is only worthwhile if it actually shrinks the agent's token bill end-to-end. On SWE-QA, SWE-QA-Pro, and Oolong (Table 1), SWE-Pruner Pro is the only pruner that reduces tokens on every cell, with savings up to 39% on the heaviest cell (Qwen3-Coder-Next, SWE-QA-Pro) and 30% on the longest-context one (MiMo-V2-Flash, Oolong). Four of six prior pruners inflate tokens on at least one cell, with LLMLingua2 reaching +190% on Oolong. The takeaway is that the pruning signal is already in the backbone's representations, so the overhead other pruners pay (goal-hint queries, surrogate scorers, or naive sliding-window retrieval) wipes out their compression gains; reading the signal in place avoids that overhead by construction.

Token savings would be hollow if quality collapsed. With Qwen3-Coder-Next, every pruner except RAG degrades judge scores by 0.14–0.65 on SWE-QA/Pro; RAG preserves quality but yields at most 6.9% token savings—negligible given the overhead it introduces. SWE-Pruner Pro is the only method that simultaneously maintains quality (+0.02, +0.24, −1.4 pp) and delivers large reductions (34.7%, 39.4%, 13.9%). The pattern holds on MiMo-V2-Flash: methods that preserve scores (RAG, Selective Context) inflate tokens on four of six cells, while those that compress (SWE-Pruner, Self-Prune) concede quality elsewhere; SWE-Pruner Pro again achieves the lowest token count on every cell, with its only loss a marginal −0.11 on SWE-QA-Pro and a +2.2 pp gain on Oolong. Across both backbones, no prior pruner resolves this quality–compression trade-off: the signal already in the backbone is sharp enough to drive aggressive pruning without discarding what the agent needs. Appendix F illustrates this on four representative tool outputs.

Code-modification benchmark. We turn next to SWE-Bench Verified, which extends the comparison to the heavier patch-generation workload (Table 2). The picture is asymmetric across backbones. On MiMo-

[2] https://huggingface.co/BAAI/bge-reranker-v2-m3

Table 2 SWE-Bench Verified results across two agent backbones.

Method	Resolved	Input Tokens	API Calls
MiMo-V2-Flash			
No Pruning	326/500	2,971K	94.8
LongCodeZip	344/500 ↑3.6%	**3,166K** ↑6.6%	**99.8**
RAG	338/500 ↑2.4%	3,391K ↑14.1%	102.4
SWE-Pruner	**347/500** ↑4.2%	3,414K ↑14.9%	103.8
SWE-Pruner Pro	345/500 ↑3.8%	3,190K ↑7.4%	111.8
Qwen3-Coder-Next			
No Pruning	341/500	5,307K	131.9
LongCodeZip	288/500 ↓10.6%	4,718K ↓11.1%	**124.7**
RAG	330/500 ↓2.2%	4,805K ↓9.5%	126.0
SWE-Pruner	320/500 ↓4.2%	4,881K ↓8.0%	127.1
SWE-Pruner Pro	**335/500** ↓1.2%	**4,590K** ↓13.5%	139.8

Input tokens are per-trajectory averages. Arrows report changes from the same-backbone *No Pruning* baseline. Bold values are best among pruning methods; shaded rows denote SWE-Pruner Pro.

V2-Flash, all pruners improve the resolve rate; SWE-Pruner Pro reaches +3.8% at roughly half the token overhead of the next-best pruner (SWE-Pruner: +4.2% resolve at +14.9% tokens). On Qwen3-Coder-Next, every pruner instead loses resolves, but SWE-Pruner Pro gives the most favorable degradation profile, losing only 6 solves (-1.2 points) while achieving the largest input-token reduction (-13.5%), outperforming the other pruners on both resolve rate and input-token use.

API calls provide a separate efficiency signal from input tokens, because pruning changes the history the agent conditions on in later turns rather than only shortening a fixed trajectory. On MiMo-V2-Flash, all pruners increase calls relative to *No Pruning*, with SWE-Pruner Pro using the most. On Qwen3-Coder-Next, LongCodeZip, RAG, and SWE-Pruner reduce calls, whereas SWE-Pruner Pro increases calls from 131.9 to 139.8 while still achieving the largest input-token reduction. Because input-token cost and API-call count trade off in opposite directions across backbones, we report them separately rather than collapsing them into a single efficiency number.

5.2 Ablations

To quantify how much each design choice contributes, we ablate the loss function and the length-aware embedding while holding everything else fixed (same 22k feature set from Qwen3-Coder-Next). We score the resulting checkpoints on a held-out judge set (n=100, GPT-5.4-mini on a 1–10 rubric) alongside per-line F1.

Table 3 Ablation results on the held-out judge set.

Design axis	Variant	F_1	Judge
Loss function	BCE	0.475	5.95
	Focal	0.593	6.37
	Dice	0.591	5.30
	Tversky	0.591	3.03
	Per-sample balanced focal ⋆	**0.635**	**7.08**
Length-aware embedding	Without length embedding	**0.636**	6.86
	With length embedding ⋆	0.635	**7.08**

Each design axis is varied while the other is held at its default (⋆). Judge: GPT-5.4-mini; F_1 is measured per line.

Loss function. We compare per-sample balanced focal against four alternatives: BCE, corpus-level Focal, Dice, and Tversky (Table 3, loss-function block; full hyperparameters in Appendix B). Per-sample balanced focal wins on both metrics, beating BCE by +1.13 judge and +0.16 F1. Dice and Tversky match its F1 but their judge collapses, because per-line label match cannot tell a useful skeleton from a precision-correct but unusable one (Appendix G). The takeaway is that the keep-or-prune imbalance is per-sample rather than global, and rebalancing inside each sample is what unlocks the gain.

Length-aware embedding. Adding the length-aware embedding lifts judge from 6.86 to 7.08 at essentially identical F1 (Table 3, length-aware-embedding block). The embedding does not change overall line-decision accuracy; it redistributes mistakes toward longer responses where mis-pruning a single line is far less harmful, exactly the length-conditioned asymmetry Eq. 1 is designed to capture.

5.3 Latency analysis

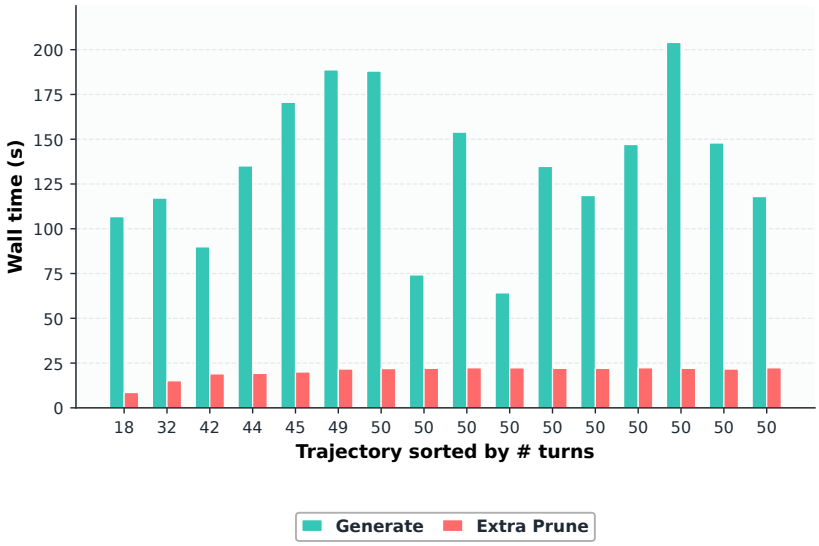

Figure 5 Per-call pruning overhead relative to the immediately following generation step, broken down by trajectory.

On a 16-trajectory MiMo-V2-Flash replay (Figure 5; setup details in Appendix E), with the in-engine pruning head of Appendix E.3 enabled, the pruning calls add 15.0% aggregate wall time relative to total generation time, with per-trajectory ratios at $p_{50} = 14.7\%$ and $p_{95} = 34.8\%$. The bound holds because the head reuses the prefill the backbone already runs on the new tool response, and colocating the head inside the inference engine avoids transferring hidden states across the engine boundary, so the only added compute is a single head forward. This per-call overhead is paid roughly once per assistant turn, while the token savings reported in Table 1 reduce backbone work on every subsequent turn, so the net wall-clock impact across a trajectory is dominated by the token reduction rather than the pruning overhead. The overhead profile is expected to shrink further in heavier-decode regimes and with tighter hidden-state quantisation; we discuss the off-engine variant and remaining headroom in Appendix E.3.

6 Related Work

6.1 Prompt Compression for Code

Reducing the prompt length seen by a code-oriented LLM (Rando et al., 2025), now widely applied from code generation (Shi et al., 2024; Chen et al., 2026) to repository-level issue resolution (Jimenez et al., 2024), has been approached along three broad axes. Token-level pruners score tokens by self-information or perplexity and drop the lowest-ranking ones (Jiang et al., 2023, 2024; Li et al., 2023b; Fang et al., 2025a); retrieval-based shorteners replace verbatim content with retrieved or aggregated snippets (Lewis et al., 2020; Cheng et al., 2024, 2026; Shi et al., 2026a; Zhang et al., 2023; Lai et al., 2026). Both families optimise content-level signals

only and disregard the syntactic and structural constraints that code understanding requires (Shi et al., 2025a,b). A third, code-aware family preserves program structure during compression (Zhang et al., 2022; Wang et al., 2024b; Shi et al., 2025a, 2026b; Hu et al., 2026; Zeng et al., 2025), but is largely evaluated on single-round proxy tasks rather than multi-turn agent loops, and applies fixed policies (Yang et al., 2024a) regardless of the agent's intent. The closest work targeting coding agents end-to-end is SWE-Pruner (Wang et al., 2026b), which performs line-level pruning conditioned on an explicit goal-hint query at each turn; SWE-Pruner Pro inherits its line-level granularity and benchmark setting but discards both the separate scoring model and the explicit query, reading the signal directly from the agent backbone's hidden states.

6.2 Agent Context Management

Even with the 128k-plus context windows of current coding LLMs (Achiam et al., 2023; Team et al., 2024), agents on real codebases routinely overflow them and the underlying models suffer quality drops as context grows (Liu et al., 2023; Laban et al., 2025; Li et al., 2023a), making context length a first-class engineering problem for software-engineering agents (Yao et al., 2023; Wei et al., 2022; Yang et al., 2024b; Wang et al., 2024a; Xia et al., 2024; Bouzenia et al., 2024; Qin et al., 2024; Liu et al., 2024; Fang et al., 2025b; Zhao et al., 2026; Chen et al., 2025; Li et al., 2025; Zhang et al., 2026b). Practical deployments cope by summarising the trajectory on overflow, hard-truncating the prefix, or masking older observations (Cursor, 2025; Anthropic, 2025; Gao et al., 2025; Lindenbauer et al., 2025); recent learned approaches train policies that manage history through reinforcement learning, hierarchical oversight, or proactive folding (Lu et al., 2025; Sun et al., 2025; Wan et al., 2025; Kang et al., 2025; Xiao et al., 2026; Ye et al., 2025; Gao et al., 2026; Zhang et al., 2026a). All of these operate on the agent's prior interaction history; SWE-Pruner Pro instead operates one level upstream, at the agent-environment boundary, by compressing incoming environment observations before they enter the history. The same boundary is targeted by SWE-Pruner (Wang et al., 2026b); where it reconstructs the pruning signal with a dedicated external model, SWE-Pruner Pro instead reads it from representations the backbone has already formed.

7 Conclusion

We have shown that the internal representations a coding agent forms while reading a tool output already encode line-level importance, removing the need for a separate scoring model or an explicit goal-hint query. SWE-Pruner Pro reads this signal out through a lightweight head that shares the backbone's prefill pass. Two design choices carry most of the empirical lift, namely a learned length-aware embedding and a per-sample balanced focal loss. The result is consistent across two open-weight backbones and four multi-turn benchmarks, where SWE-Pruner Pro reduces token consumption while keeping quality within a narrow band of the unpruned baseline, at negligible added inference cost. This points to a broader principle: the representations a backbone forms while passively reading an observation already encode the relevance judgment that prior approaches spent an extra model call to recover. Rather than building pruning around the agent, it suffices to read the signal the agent has already formed.

Limitations

We close with two scope notes. SWE-Pruner Pro reads the agent backbone's internal hidden states, so the present evaluation covers only open-weight models; the recipe itself extends to any backbone that exposes its hidden states, and the consistent quality preservation across the SWE-QA family and the natural-language Oolong setting suggests the paradigm transfers across both code and natural-language tool outputs, with only per-backbone head retraining required for a new model. As for language coverage, although our agent-task benchmarks are Python-centric, the pipeline is language-agnostic, and Oolong (a long-context natural-language aggregation benchmark) gives an out-of-domain check on which SWE-Pruner Pro preserves both token savings and quality on both backbones (Table 1); broader coverage across programming languages reuses the same pipeline and is left to future work.

Ethical Considerations

All training trajectories are drawn from publicly released datasets; no private repositories or proprietary codebases were used. The pruning head is trained solely to compress redundant tool output and does not alter the agent's generation or reasoning; deployment should be validated per-backbone before use in safety-critical settings, as aggressive pruning may degrade task quality in ways not captured by our benchmarks.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Anthropic. Claude code: Built for developers, 2025. URL https://claude.com/product/claude-code.

Amanda Bertsch, Adithya Pratapa, Teruko Mitamura, Graham Neubig, and Matthew R. Gormley. Oolong: Evaluating long context reasoning and aggregation capabilities, 2025. URL https://arxiv.org/abs/2511.02817.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. 2024. URL https://doi.org/10.48550/arXiv.2403.17134.

Songcheng Cai, Zhiheng Lyu, Yuansheng Ni, Xiangchao Chen, Baichuan Zhou, Shenzhe Zhu, Yi Lu, Haozhe Wang, Chi Ruan, Benjamin Schneider, Weixu Zhang, Xiang Li, Andy Zheng, Yuyu Zhang, Ping Nie, and Wenhu Chen. SWE-QA-Pro: A representative benchmark and scalable training recipe for repository-level code understanding. In *Findings of the Association for Computational Linguistics (ACL Findings)*, 2026. URL https://arxiv.org/abs/2603.16124.

Silin Chen, Shaoxin Lin, Yuling Shi, Heng Lian, Xiaodong Gu, Longfei Yun, Dong Chen, Lin Cao, Jiyang Liu, Nu Xia, et al. Swe-exp: Experience-driven software issue resolution. *arXiv preprint arXiv:2507.23361*, 2025.

Yeheng Chen, Chaoxiang Xie, Yuling Shi, Wenhao Zeng, Yongpan Wang, Hongyu Zhang, and Xiaodong Gu. Classeval-pro: A cross-domain benchmark for class-level code generation. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software*, pages 340–348, 2026.

Xin Cheng, Xun Wang, Xingxing Zhang, Tao Ge, Si-Qing Chen, Furu Wei, Huishuai Zhang, and Dongyan Zhao. xRAG: Extreme Context Compression for Retrieval-augmented Generation with One Token, May 2024.

Zhiyuan Cheng, Longying Lai, and Yue Liu. Resolving the robustness-precision trade-off in financial rag through hybrid document-routed retrieval, 2026.

Cursor. Cursor – the ai code editor, 2025. URL https://cursor.com/.

Yixiong Fang, Tianran Sun, Yuling Shi, and Xiaodong Gu. Attentionrag: Attention-guided context pruning in retrieval-augmented generation. *arXiv preprint arXiv:2503.10720*, 2025a.

Yixiong Fang, Tianran Sun, Yuling Shi, Min Wang, and Xiaodong Gu. Lastingbench: Defend benchmarks against knowledge leakage. *arXiv preprint arXiv:2506.21614*, 2025b.

Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*, 2025.

Shuzheng Gao, Wenhao Zeng, Zhaojian Yu, Jianqiao Wangni, Chaozheng Wang, Kai Cai, Shilin He, and Michael R Lyu. Swe-mem: Learning adaptive memory management for long-horizon coding agents. *arXiv preprint arXiv:2606.28434*, 2026.

Chao Hu, Wenhao Zeng, Yuling Shi, Beijun Shen, and Xiaodong Gu. In line with context: Repository-level code generation via context inlining. *arXiv preprint arXiv:2601.00376*, 2026.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *ICLR*, 2024.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. LLMs Get Lost In Multi-Turn Conversation. 2025. URL https://doi.org/10.48550/arXiv.2505.06120.

Longying Lai, Zhiyuan Cheng, Kai Cheng, and Xiaoxi Qi. Do transformers always win? an empirical study of semantic embeddings for short-text e-commerce reviews, 2026.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

Han Li, Yuling Shi, Shaoxin Lin, Xiaodong Gu, Heng Lian, Xin Wang, Yantao Jia, Tao Huang, and Qianxiang Wang. Swe-debate: Competitive multi-agent debate for software issue resolution. *arXiv preprint arXiv:2507.23348*, 2025.

Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. Loogle: Can long-context language models understand long contexts? *arXiv preprint arXiv:2311.04939*, 2023a.

Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023b.

Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. The Complexity Trap: Simple Observation Masking Is as Efficient as LLM Summarization for Agent Context Management. 2025. URL https://doi.org/10.48550/arXiv.2508.21433.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large Language Model-Based Agents for Software Engineering: A Survey. 2024.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*, 2023.

Miao Lu, Weiwei Sun, Weihua Du, Zhan Ling, Xuesong Yao, Kang Liu, and Jiecao Chen. Scaling llm multi-turn rl with end-to-end summarization-based context management. *arXiv preprint arXiv:2510.06727*, 2025.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, et al. Llmlingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 963–981, 2024.

Weihan Peng, Yuling Shi, Yuhang Wang, Xinyun Zhang, Beijun Shen, and Xiaodong Gu. SWE-QA: Can language models answer repository-level code questions? In *Findings of the Association for Computational Linguistics (ACL Findings)*, 2026. URL https://arxiv.org/abs/2509.14635.

Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. Agentfl: Scaling llm-based fault localization to project-level context. *arXiv preprint arXiv:2403.16362*, 2024.

Qwen Team. Qwen3-Coder-Next technical report, 2026. URL https://arxiv.org/abs/2603.00729.

Stefano Rando, Luca Romani, Alessio Sampieri, Luca Franco, John Yang, Yuta Kyuragi, Fabio Galasso, and Tatsunori Hashimoto. Longcodebench: Evaluating coding llms at 1m context windows. *arXiv preprint arXiv:2505.07897*, 2025.

Yuling Shi, Songsong Wang, Chengcheng Wan, Min Wang, and Xiaodong Gu. From code to correctness: Closing the last mile of code generation with hierarchical debugging. *arXiv preprint arXiv:2410.01215*, 2024.

Yuling Shi, Yichun Qian, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. Longcodezip: Compress long context for code language models. *arXiv preprint arXiv:2510.00446*, 2025a.

Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. Between lines of code: Unraveling the distinct patterns of machine and human programmers. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1628–1639. IEEE, 2025b.

Yuling Shi, Maolin Sun, Zijun Liu, Mo Yang, Yixiong Fang, Tianran Sun, and Xiaodong Gu. Reasoning in trees: Improving retrieval-augmented generation for multi-hop question answering. *arXiv preprint arXiv:2601.11255*, 2026a.

Yuling Shi, Chaoxiang Xie, Zhensu Sun, Yeheng Chen, Chenxu Zhang, Longfei Yun, Chengcheng Wan, Hongyu Zhang, David Lo, and Xiaodong Gu. Codeocr: On the effectiveness of vision language models in code understanding. *arXiv preprint arXiv:2602.01785*, 2026b.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

Guangya Wan, Mingyang Ling, Xiaoqi Ren, Rujun Han, Sheng Li, and Zizhao Zhang. Compass: Enhancing agent long-horizon reasoning with evolving context. *arXiv preprint arXiv:2510.08790*, 2025.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. 2024a.

Yan Wang, Xiaoning Li, Tien N Nguyen, Shaohua Wang, Chao Ni, and Ling Ding. Natural is the best: Model-agnostic code simplification for pre-trained large language models. *Proceedings of the ACM on Software Engineering*, 1(FSE): 586–608, 2024b.

Yifei Wang, Ziteng Wang, Yuling Shi, Silin Chen, Xinrui Wang, Yueqi Wang, Beijun Shen, Linjing Li, Xiaodong Gu, Julian McAuley, and Daniel Dajun Zeng. Context compression for llm agents: A survey of methods, failure modes, and evaluation. *Preprints*, May 2026a. doi: 10.20944/preprints202605.2065.v1. URL https://doi.org/10.20944/preprints202605.2065.v1.

Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. SWE-Pruner: Self-adaptive context pruning for coding agents, 2026b. URL https://arxiv.org/abs/2601.16746.

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.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based Software Engineering Agents. 2024.

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. Reducing cost of LLM agents with trajectory reduction. In *Proceedings of the ACM on Software Engineering (FSE)*, 2026. doi: 10.1145/3797084. URL https://arxiv.org/abs/2509.23586.

Xiaomi LLM-Core Team. MiMo-V2-Flash technical report, 2026. URL https://arxiv.org/abs/2601.02780.

Guang Yang, Yu Zhou, Wei Cheng, Xiangyu Zhang, Xiang Chen, Terry Yue Zhuo, Ke Liu, Xin Zhou, David Lo, and Taolue Chen. Less is More: DocString Compression in Code Generation, 2024a.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024b. URL https://arxiv.org/abs/2405.15793.

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

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Siheng Chen, Jingren Zhou, and Yong Jiang. Agentfold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*, 2025.

Wenhao Zeng, Yaoning Wang, Chao Hu, Yuling Shi, Chengcheng Wan, Hongyu Zhang, and Xiaodong Gu. Pruning the unsurprising: Efficient code reasoning via first-token surprisal. *arXiv preprint arXiv:2508.05988*, 2025.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, 2023.

Shaoqiu Zhang, Maoquan Wang, Yuling Shi, Yuhang Wang, Xiaodong Gu, Yongqiang Yao, Rao Fu, and Shengyu Fu. Fastcontext: Training efficient repository explorer for coding agents. *arXiv preprint arXiv:2606.14066*, 2026a.

Shaoqiu Zhang, Yuhang Wang, Jialiang Liang, Yuling Shi, Wenhao Zeng, Maoquan Wang, Shilin He, Ningyuan Xu, Siyu Ye, Kai Cai, et al. Swe-explore: Benchmarking how coding agents explore repositories. *arXiv preprint arXiv:2606.07297*, 2026b.

Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. Diet code is healthy: Simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1073–1084, 2022.

Jiahao Zhao, Shaoxuan Xu, Zhongxiang Sun, Fengqi Zhu, Jingyang Ou, Yuling Shi, Chongxuan Li, Xiao Zhang, and Jun Xu. Dllm-searcher: Adapting diffusion large language model for search agents. *arXiv preprint arXiv:2602.07035*, 2026.

A Training Data

Overview. The training corpus aggregates agent trajectories from five publicly released HuggingFace datasets: four are SWE-style code-modification rollouts and one (`terminal-wrench-trajectories`) covers a mix of CLI / shell tasks spanning chess, machine learning, cryptography, databases and shell scheduling. We deliberately include both code-patch and non-code agent traces so that the SWE-Pruner Pro head learns to handle the heterogeneous tool outputs encountered in practice. After all stages of the pipeline (Figure 6) the labelled corpus contains 22,609 (*history*, *tool_call*, *tool_response*) samples drawn from 6,252 unique trajectories (≈3.6 steps per trajectory on average); the per-source breakdown is given in Table 4.

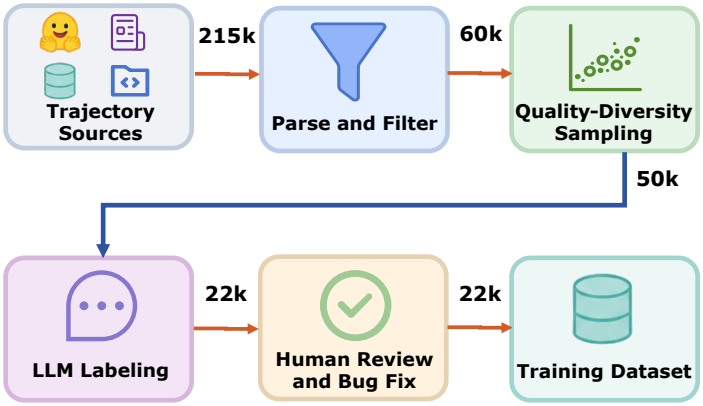

Figure 6 Data construction pipeline. Raw trajectories from five HuggingFace sources are parsed into a uniform per-step schema, light-filtered, then sub-sampled to a 50k diverse pool by quality-diversity selection, labelled per line by Claude Sonnet 4.6, and finally passed through a small human review pass that drops a few hundred of malformed labels.

Table 4 Source-level composition of the labelled training corpus.

HuggingFace dataset	Traj.	Samples	Share
`few-sh/terminal-wrench-trajectories`	3 349	6 848	30.3%
`TIGER-Lab/SWE-Next-SFT-Trajectories`	899	6 632	29.3%
`ByteDance-Seed/Multi-SWE-bench_trajs`	872	3 760	16.6%
`zai-org/CC-Bench-trajectories`	280	3 074	13.6%
`AweAI-Team/Scale-SWE-Distilled`	852	2 295	10.2%
Total	**6 252**	**22 609**	**100%**

A sample is one (*history*, *tool call*, *tool response*) step; a trajectory is one globally deduplicated `instance_id`.

Pre-filtering and diversity sampling. A per-source adaptor parses each raw trajectory into a flat list of chat messages following the OpenAI tool-calling schema (`system / user / assistant / tool` roles), and emits one (*history*, *tool_call*, *tool_response*, *next_turn*) quadruple for every `tool` message. The `history` is kept as a sliding window that preserves complete `(assistant, tool)` turn pairs; partial pairs that do not fit are dropped rather than truncated, so `tool_calls` are never split. A light pre-filter drops responses with fewer than two non-blank lines, implausibly low code-line ratios, and pure stack-trace error output, so that the labelling budget is spent on tool responses with substantive content. The surviving pool is larger than what we want to label, so we sub-sample it to a 50k diverse subset. The selection maximises a facility-location objective over instance metadata (language, repo, response-size bucket and a log-step length kernel) under a per-instance step cap. This is run greedily with lazy evaluation, which gives the standard $(1-1/e)$ approximation guarantee and keeps the selection cost negligible relative to labelling.

Table 5 Language and category composition of the training corpus.

Language / category	Samples	Share
Python	8 927	39.5%
C++	976	4.3%
JavaScript	828	3.7%
C	709	3.1%
Java	561	2.5%
TypeScript	500	2.2%
Go	186	0.8%
Bash (dominant)	3 074	13.6%
Domain-mixed (CLI)	6 848	30.3%

Per-line labelling. Each retained quadruple is labelled by CLAUDE SONNET 4.6 via the Anthropic API. The labeller sees the full `history`, the triggering `tool_call`, the line-numbered `tool_response` (with internal `cat -n` numbers stripped to avoid confusing inner with outer indices), and the `next_turn` snippet. It emits a list of 1-based line indices that the agent must retain, together with a short `reasoning` field and a `confidence` flag. About half of the input pool yields a non-empty label set; the rest is naturally rejected by the labeller as containing nothing worth retaining (boilerplate, pure logs, etc.) and dropped. A separate category covers responses where the labeller cannot identify specific lines to prune. These are marked `skeleton` when the labeller declares the entire output is a scaffold the agent will reuse downstream (≈17% of the corpus, mostly in TIGER-Lab, CC-Bench and Multi-SWE-bench); for these samples every token in the tool response receives label 1, giving the head explicit examples of the "do not prune" regime. At training time, the kept-line set is expanded to per-token binary labels (a token receives label 1 if its character span overlaps at least one kept line) and tokens outside the `<tool_response>...</tool_response>` span are masked out of the loss. A final manual-review pass removes a few hundred labels with obvious inner/outer line-number confusion, empty kept-sets on non-trivial responses, and malformed `instance_id`s, leaving the 22,609-sample corpus used throughout the paper.

Distribution. The `tool_response` length distribution is heavy-tailed but bounded by an upstream length cutoff: mean 76 lines, median 56 lines, with the 10/90/99 percentiles at 25/143/294 and a hard maximum of 465 lines. The CLAUDE-assigned keep-ratio (kept lines ÷ total lines) has mean 0.32 and median 0.23, matching our compression target of ≈30%. In terms of language coverage, languages are recovered from `instance_id` for `Multi-SWE-bench_trajs` (which encodes the language as `mswe_<lang>_...`) and inferred from the source dataset for the others: `Scale-SWE-Distilled` and `TIGER-Lab/SWE-Next` are Python-only, `CC-Bench` is bash-dominant with some Python, and `terminal-wrench` is a domain-mixed CLI corpus. The resulting breakdown is shown in Table 5, grouped into single-language rows (top) and mixed/CLI rows (bottom). Python and CLI/shell-style content together account for ≈83% of the corpus; six explicit non-Python languages from `Multi-SWE-bench_trajs` make up the remaining ≈17%.

Probe subset for §2. The motivation analysis in §2 uses a random 10% subset of the labelled corpus drawn at the trajectory level (≈ 625 trajectories, ≈ 2,260 tool responses, ≈ 155k lines). Within this subset we further split trajectories 90/10 into train/eval for the linear probe, so no trajectory contributes lines to both splits; the AUC 0.83 and best-F_1 0.63 reported in Figure 2 are computed on the held-out trajectories. The subset is intentional: the analysis is an existence proof for the keep-or-prune signal in the backbone's hidden states, and a logistic regression on a fraction of the corpus is sufficient to test separability. The head architecture and training in §3 use the full corpus.

B Training Details

Head architecture. The pruning head is a per-token feed-forward MLP applied to the frozen backbone's last-layer hidden states h_i. The continuous length-aware embedding $\mathbf{e}(N)$ of Eq. 1 is realised over 8 log-spaced `n_lines` buckets (0–2, 3–5, 6–10, 11–20, 21–50, 51–100, 101–200, >200), additively combined with each hidden

state in the tool-response span and zero-initialised so f_θ reduces to its length-agnostic limit at the start of training. The classifier f_θ itself is a LayerNorm followed by two `Linear-GELU-Dropout` blocks with hidden dimension matching the backbone hidden size d (so the head is re-sized per backbone) and dropout 0.4, then a final Linear projection to a single keep logit.

Optimisation. The head is optimised with AdamW (β_1=0.9, β_2=0.999, ϵ=10^{-8}, weight decay 0) at a peak learning rate of 3×10^{-5}. The schedule is a linear warmup over the first 5% of total updates, followed by cosine decay to a floor of 1.5×10^{-5}, so the effective learning rate stays in $[1.5 \times 10^{-5}, 3 \times 10^{-5}]$ throughout the run. Gradient clipping with max-norm 1.0 is applied to the head parameters. Random seeds are fixed (`seed=42`); no backbone parameters are updated at any point.

Loss. The training objective is the per-sample class-balanced focal loss of Eq. 6, with focal exponent $\gamma = 2$ (Eq. 4) and the equal-weight 0.5/0.5 keep/prune mix of Eq. 5. Samples that happen to contain only one class contribute the surviving branch with weight 1 and drop the absent branch from the average.

Frozen-backbone, feature-cache pipeline. Hidden states are pre-extracted from the frozen backbone once per dataset × backbone pair and cached on disk as memory-mapped feature files; all subsequent head training reads directly from these features rather than re-running the backbone. A full 10-epoch pass over the 22,609 samples completes in $\approx$ 15 min on a single 8 × H200 node, which makes the loss / length-bias ablation grid in §5.2 cheap to run end-to-end.

C Oolong Benchmark Conversion

Oolong as a multi-turn agent benchmark. Oolong was originally released as a single-turn benchmark: each of the 280 examples contains a long synthetic context (10K–65K tokens, drawn from AGNEWS, IMDB, MULTINLI, SPAM, etc.) and a counting-style question (e.g., most-common label, label frequency), and a model is expected to produce the answer in a single pass from the full context. Long-context aggregation is a natural fit for agentic exploration: the model can locate and aggregate the relevant rows itself rather than scanning the full context end-to-end. We therefore re-cast Oolong into a multi-turn tool-use setting that matches the harness of our other benchmarks, and reuse the official scorer.

Multi-turn re-cast. For each example, the long context is materialised as a read-only `data.txt` inside a Docker sandbox; the model is no longer fed the context directly, but is instead given a single `bash` tool and asked to explore the file with standard CLI utilities (`grep`, `awk`, `sort`, `uniq -c`, `wc`, ...) until it produces an `Answer:` line. Tool outputs are returned to the model as `tool_response` messages and pruned on-the-fly by each method under test, exactly as in our other multi-turn benchmarks. Scoring reuses Oolong's official `synth_process_response` routine, so numbers remain directly comparable to the original single-turn leaderboard. In practice this turns each question into a short trajectory (typically 3–14 tool calls, depending on backbone), where the agent must locate and aggregate the relevant rows itself rather than relying on the model to scan the full context end-to-end.

D Per-turn execution

Algorithm 1 expands the per-turn pipeline of §3.1 as pseudocode, including the turn-boundary substitution that replaces r_t with the pruned $\tilde{r}_t$ in the history H_t used by turn $t{+}1$.

Algorithm 1 Per-turn execution of SWE-Pruner Pro

Require: History H_{t-1}, tool call c_t, raw response r_t with N lines, head f_θ, threshold τ
Ensure: Pruned response $\tilde{r}_t$

$\{h_i\}_{i=1}^{L} \leftarrow \text{Prefill}(r_t \mid H_{t-1}, c_t)$ ▷ only r_t forwarded; prefix cached
$\mathbf{b} \leftarrow \mathbf{e}(N)$ ▷ length-aware embedding, Eq. 1
for $i = 1, \ldots, L$ **do**
  $\tilde{h}_i \leftarrow h_i + \mathbf{b}$
  $p_i \leftarrow \sigma\Big(f_\theta(\tilde{h}_i)\Big)$ ▷ Eq. 2
end for
for each line ℓ **do**
  $\hat{y}_\ell \leftarrow \mathbb{1}\Big[\frac{1}{|\ell|}\sum_{i\in\ell}\mathbb{1}[p_i > \tau] > \frac{1}{2}\Big]$ ▷ Eq. 3
end for
$\tilde{r}_t \leftarrow r_t$ with all lines ℓ satisfying $\hat{y}_\ell = 0$ removed
Substitute $\tilde{r}_t$ for r_t when forming H_t for turn $t{+}1$
return $\tilde{r}_t$

E In-server hidden-state extraction

The SWE-Pruner Pro head consumes per-token last-layer hidden states from the prefill the backbone already performs on each tool response (§3.1). We serve the backbone on SGLang[3] 0.5.10.post1 with `--enable-return-hidden-states`, which ships those hidden states back inside the generation response. In practice we found three correctness gaps on the hidden-state path where guards already enforced on the logprob path were missing: alignment, chunked-prefill, and prefix-cache. We additionally found that routing hidden states through the existing JSON response field is impractical for tensors three orders of magnitude larger than a typical text payload. We close these gaps below in three stages — three correctness fixes (§E.1), a payload-size fix (§E.2), and an in-engine head colocation that removes the cross-process boundary entirely (§E.3) — then validate the corrected hidden states against a pure-transformers reference and quantify the payload reduction.

E.1 Correctness: closing the hidden-state vs. logprob asymmetry

We identified three failure modes on the hidden-state path. Each is the same structural problem in different clothing: a guard that the logprob path already enforces is simply absent on the hidden-state side. We restore symmetry rather than introduce new mechanisms, which keeps the patches small, local, and consistent with SGLang's existing design idioms.

Batch alignment. `tokenizer_manager.py` indexes `recv_obj.output_hidden_states[i]` assuming the list is aligned with `rids`. Upstream, the scheduler appended to `output_hidden_states` only for requests with `return_hidden_states=True`, so in a mixed batch (agent generation calls interleaved with pruner hidden-state requests) the list ran shorter than the batch and the index raised `IndexError`. The logprob path already preallocates a per-request slot in this situation. We mirror that on the hidden-state side: preallocate `[] if any(r.return_hidden_states for r in reqs) else None` in `stream_output_generation` and always append either the per-request hidden states or `None`, so `len(output_hidden_states) == len(rids)` holds in every batch.

Chunked-prefill accumulation. Under chunked prefill each forward emits hidden states only for the current chunk, but the original code only captured them on `is_chunked<=0` (the final chunk), so for any prompt that crossed the chunk boundary the returned tensor covered only the tail span and was tagged

[3] https://github.com/sgl-project/sglang

Table 6 Validation of patched SGLang hidden states against a pure-transformers `flash_attention_2` reference.

Metric	Before patches	After patches
Shape match	1 / 48	**48 / 48**
Cosine, median	n/a	**0.997**
Cosine, mean	n/a	0.983

Shape match counts samples with matching tensor shape; cosine is computed per token against the reference and then aggregated.

with the wrong absolute positions. The logprob path already accumulates per-chunk results into the request slot using the per-chunk span recorded in `extend_input_len_per_req[i]`; we apply the same accumulation pattern, appending each chunk's hidden states into `req.hidden_states[0]` via `np.concatenate`. Because `event_loop_overlap` overwrites `req.extend_input_len` in place before the consumer reads it, we further widen the existing logprob snapshot guard in `scheduler.py` from `batch.return_logprob` to `batch.return_logprob or batch.return_hidden_states` so the per-request span is captured for hidden-state requests too. After the patch, `req.hidden_states[0]` covers positions $[0, L)$ for any chunked prefill of length L.

Prefix-cache exemption. SGLang's radix prefix cache stores the KV produced by a previous forward but not the hidden states. When a later request shares a leading token sub-sequence with a cached entry, those positions are skipped during prefill: the KV is reused directly and the model never produces hidden states for them. The returned hidden-state tensor therefore covers only the suffix `[prefix_len: total_len]`, and any consumer that needs hidden states for a region the cache happens to cover sees a silently truncated tensor. The condition triggers whenever the cached prefix covers tokens whose hidden states the consumer needs, e.g. an earlier identical prompt. Logprob has the exact same problem in principle, and SGLang already solves it: `init_next_round_input` caps `max_prefix_len` at `logprob_start_len` when `return_logprob` is on, forcing positions $\geq$ `logprob_start_len` through the forward pass. We mirror this on the hidden-state side by threading a new `hidden_states_start_len` field through `GenerateReqInput`, `TokenizedGenerateReqInput` and `Req`, and extending the same cap in `init_next_round_input`:

```
if self.return_hidden_states and self.hidden_states_start_len >= 0:
    max_prefix_len = min(max_prefix_len, self.hidden_states_start_len)
```

The caller sets `hidden_states_start_len` to the start of the region it needs hidden states for; tokens before the cap still benefit from cache reuse, while tokens after it are guaranteed to forward and produce hidden states.

Validation against a transformers reference. We validate the value-level patches (batch alignment and chunked-prefill accumulation) with a regression harness on a held-out 48-sample probe set bucketed by prompt length ($<$ 2k / 2–8k / 8–16k / 16–24k, 12 samples each), comparing patched SGLang hidden states against a pure-transformers reference path using `flash_attention_2`. The prefix-cache patch cannot be exercised against this reference (transformers has no shared radix cache); we validate it directly by toggling the `hidden_states_start_len` cap and re-issuing a probe whose prefix is a known superset of an earlier request, then checking that the returned tensor covers the requested region rather than the post-prefix suffix only.

After patching, all 48 samples match shape exactly and per-token cosine reaches 0.997 at the median (Table 6). The mean is pulled down by a tail with cos $<$ 0.98 (min 0.33), which we attribute to bf16 arithmetic noise from different attention backends and GEMM reduction order rather than to a residual correctness bug.

E.2 Payload: replacing text-shaped serialization

Because the pruning head reuses the backbone's existing prefill, the only additional traffic over a non-pruning deployment is a single hidden-state tensor crossing the SGLang $\rightarrow$ pruner boundary per tool response. Our initial implementation routed the tensor through SGLang's existing JSON response path: the numpy hidden-state tensor is written into `meta_info.hidden_states` and serialized by `orjson` as a nested list of fp32 values through `OPT_SERIALIZE_NUMPY`. This default is fine for short text fields but ill-suited to hidden-state tensors:

Table 7 Per-request hidden-state payload by serialization format.

Format	Payload size
Nested JSON list of fp32	1–3 GB text
Binary envelope, fp32	170 MiB
Binary envelope, fp16	**85 MiB**

Measured for a 16 k-token prompt with hidden size H=2048.

for a 16 k-token prompt at hidden size 2048 the 16384 × 2048 fp32 tensor is 128 MB of raw bytes but expands to 1–3 GB after ASCII float formatting, with both the server `orjson` walk and the client `json.loads` paying per-element. Two nested optimisations close this gap: replace the nested list with a raw binary envelope to avoid the ASCII-formatting cost, then cast to fp16 to halve the byte count at a controlled precision cost.

We carry the tensor in the same `meta_info.hidden_states` field as a base64 binary envelope:

```
arr = recv_obj.output_hidden_states[i][0]
meta_info["hidden_states"] = {
    "__binary__": True, "shape": list(arr.shape),
    "dtype": str(arr.dtype),
    "data": pybase64.b64encode(arr.tobytes()).decode("utf-8"),
}
```

The same envelope carries either fp32 or fp16 depending on `arr.dtype`; we cast to fp16 on the SGLang side before encoding.

The binary envelope removes the ASCII-formatting blowup, and stacking fp16 on top halves the byte count again, giving a roughly 20× end-to-end reduction over the default path (Table 7) at a precision cost bounded by the median-token cosine 0.997 reported in Table 6. This payload reduction is what keeps the per-call pruning overhead in Figure 5 bounded; the default nested-JSON path would dominate per-call wall time on long prompts.

E.3 In-engine pruning head

The patches above describe the off-engine path: hidden states leave the inference engine and the head runs as a separate pruner process. When the head is small (here an 18M-parameter FFN on top of a multi-billion-parameter MoE backbone), this boundary can be removed by colocating the head inside the SGLang scheduler itself. The head runs directly on the captured hidden-state tensor on the scheduler's GPU and returns per-token logits in place of the hidden states, so the same payload envelope of §E.2 carries the result with the shape reduced from $[T, H]$ to $[T]$ and the cross-process hidden-state transfer eliminated entirely.

On the same 16-trajectory MiMo-V2-Flash replay used in §5.3, the aggregate per-call overhead drops from 19.3% (off-engine, with all correctness and payload optimisations of §E.1–§E.2 applied) to 15.0% (in-engine), with the residual cost coming from the prefix-cache-exempt prefill the head still needs as input. The trade-off is that in-engine colocation modifies the inference engine's request schema and ties the deployment to a specific head architecture, whereas the off-engine path is engine-agnostic and treats the head as an external service.

Two factors will lower the relative overhead reported in §5.3, acting on opposite sides of the ratio. **Numerator-side**, the pruning cost itself can shrink: the decode path our pruning overhead is measured against has received years of inference-engine optimisation (overlap, paged attention, speculative decoding, prefix caching, custom kernels), whereas our pruning path is currently a hand-tuned baseline of an fp16 base64 envelope and a single-process head; quantising the hidden-state payload to fp8 or INT8 with a custom pack/unpack kernel halves or quarters the bytes on top of the fp16 reduction in Table 7. **Denominator-side**, modern coding agents increasingly interleave extended reasoning between tool calls, with per-turn thinking budgets now standard practice across frontier coding models; under such heavy-decode workloads per-turn decode dominates wall-clock and the same per-call pruning cost occupies a much smaller fraction of it. The bounded 15% overhead reported in §5.3 is therefore closer to the worst-case end of the regime than the typical one.

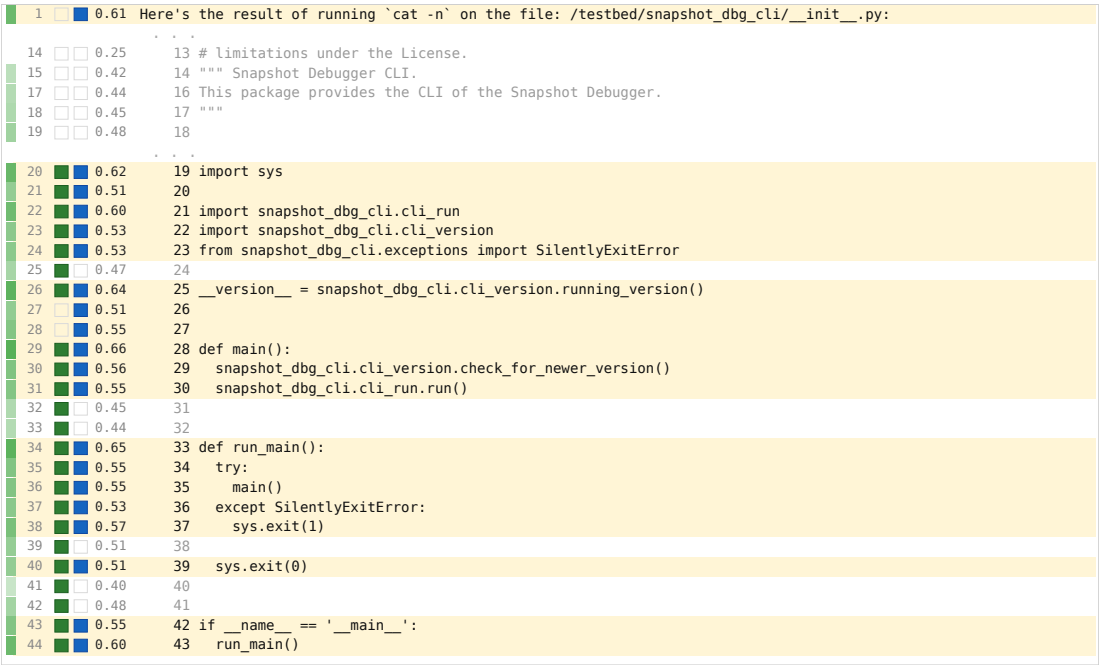

Figure 7 Read case: `view /testbed/snapshot_dbg_cli/__init__.py`. **Legend (shared by all four cases).** The left score gutter encodes the head's predicted keep probability $p_i = \sigma(z_i)$, fading from white (low) to **green** (high). **G** is filled when the gold annotator marked the line as *keep*; **P** is filled when the head predicts *keep* at τ=0.5. Predicted-keep rows additionally get a yellow row wash with crisp black text; pruned rows are dimmed. **Case.** The head assigns the lowest scores in the response (0.25–0.48) to the license/docstring boilerplate—the omitted lines 2–13 and the visible `# limitations under the License` plus the package docstring on lines 14–19—while concentrating mass at 0.51–0.66 on the imports, the `__version__` assignment, and the two function bodies (`main`, `run_main`), exactly the symbols the agent will reference when wiring its patch. The path-header line 1 is also kept, anchoring the file identity for downstream turns.

F Qualitative cases: per-line head scores on tool outputs

We sample four cases from the held-out evaluation set (Qwen3-Coder-Next backbone), one for each of the most common non-edit tool families: `cat`, `grep`, `ls`, and test execution. For each case we show the head's per-line scores alongside the gold annotation.

F.1 Read case: `view /testbed/snapshot_dbg_cli/__init__.py`

Setup. The agent opens a package's `__init__.py` to learn the CLI entry-point structure before patching it. The 44-line response is summarised in Figure 7; gold keeps 23 lines, the head keeps 18 at τ=0.5.

F.2 Search case: `grep -A 25` for a Flask helper

Setup. The agent is investigating Flask's templating layer and pipes `cat -n templating.py` through `grep -A 25 'def render_template_string'` to view the function and its tail context (Figure 8). The 20-line response has gold 12, predicted 6.

F.3 Listing case: `ls -la` on legacy admin config dirs

Setup. The agent is auditing a legacy admin home and lists three sibling configuration directories (`cron/`, `db/`, `secrets/`) in one call (Figure 9). Of the 25-line response, gold keeps 4 lines and the head keeps 12.

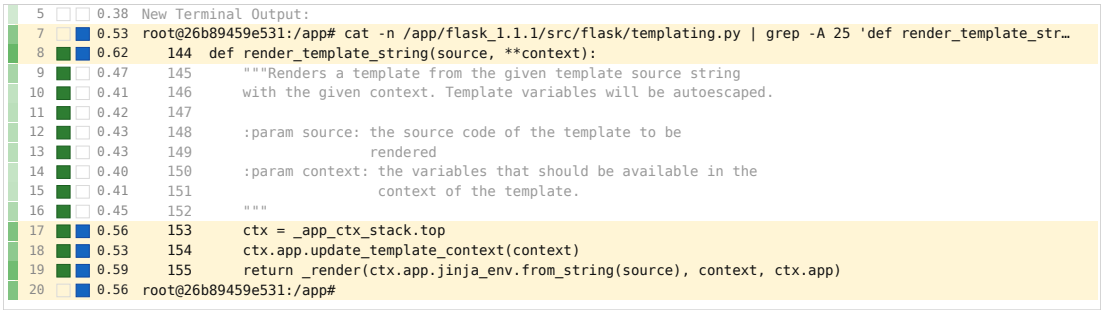

Figure 8 Search case: `cat -n /app/flask_1.1.1/src/flask/templating.py | grep -A 25 'def render_template_string'`. The head allocates the highest scores to the function's executable body—the `_app_ctx_stack` lookup, the `update_template_context` call, and the final `_render(...)` return—which together fully specify how a string template is dispatched. The intervening docstring lines are conservatively pruned even though the annotator keeps them; the head prioritises the call sequence over the prose as the authoritative spec.

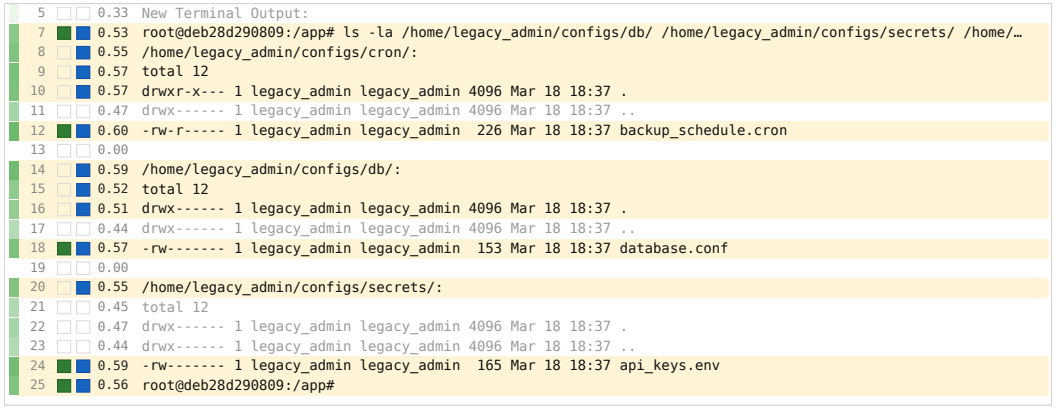

Figure 9 Listing case: `ls -la /home/legacy_admin/configs/{cron,db,secrets}/`. The head keeps the issued command and every actual file (`backup_schedule.cron`, `database.conf`, `api_keys.env`), which are the only artefacts the agent will need to act on. The remaining eight over-keeps are concentrated on group boundaries rather than scattered noise: the three directory-path headers (`/cron/:`, `/db/:`, `/secrets/:`), two of the three `total 12` summaries, two leading `.` entries, and the closing shell prompt. The head still cleanly prunes the redundant `..` parents, blank separators, and the third `total 12`, so its conservative bias on this listing costs structural boilerplate, not signal.

F.4 Test case: `python test_fix.py` traceback

Setup. The agent is debugging a JSON-schema generator (`lollipop-jsonschema`) and re-runs its repro script after an edit (Figure 10). The 21-line tail has gold 13, predicted 15.

Cross-case observation. A consistent pattern across the four cases is that per-token sigmoid scores are tightly compressed into a narrow ∼0.4–0.7 band rather than saturated near 0 or 1. The head is therefore not learning a single global keep threshold; it produces a fine-grained relative ranking and relies on the length-aware embedding of Eq. 1 to shift the operating point so the per-line majority vote of Eq. 3 lands at the right absolute keep rate for each response length. This matches the design intent of §3.2: the head identifies relative importance, and the length-aware embedding absorbs the absolute keep rate that varies with response length.

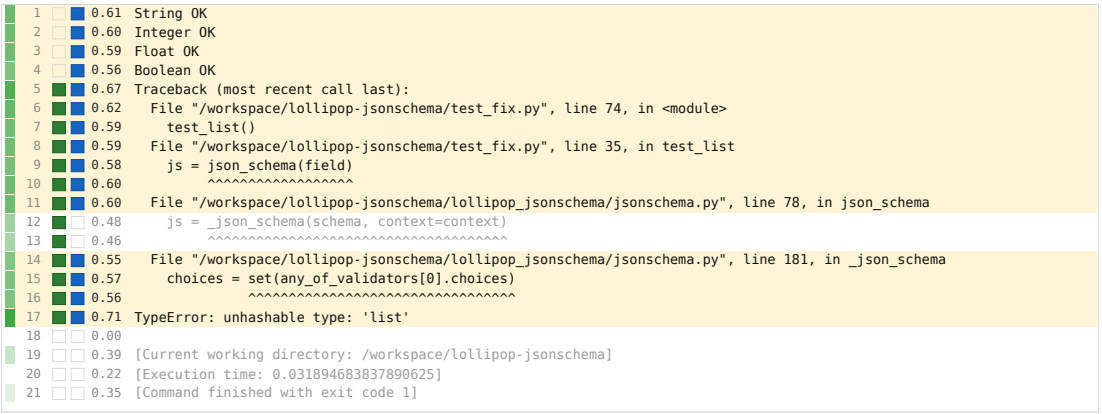

Figure 10 Test case: `python test_fix.py` (`lollipop-jsonschema`). The head locks onto the outer frames of the stack trace: the `Traceback` header, the `<module>` and `test_list` call sites, the public `json_schema` entry, the offending `set(any_of_validators[0].choices)` expression, and the terminal `TypeError: unhashable type: 'list'` (the highest-scoring line in the response at 0.71). The intervening inner-dispatch frame (`js = _json_schema(...)` on line 78 of the same file) is conservatively pruned, since it only forwards into the line that already carries the highest keep score. The four passing lines (`String OK`, `Integer OK`, `Float OK`, `Boolean OK`) are kept too, providing the contrast that localises the bug to the list path. The trailing harness footer (cwd, exit-code banner) prunes cleanly.

G Qualitative cases: where F1 disagrees with the LLM judge

The ablation in §5.2 shows that F1 and the LLM judge can diverge sharply on the loss axis: Dice and Tversky reach F1 within 0.04 of our default but their judge scores collapse to 5.30 and 3.03, and even BCE outscores our default on F1 in some checkpoints despite being clearly worse to the agent. To see why a high-F1 head can still produce an unusable skeleton, we examine two real tool-response pairs from the held-out evaluation set, each scored against the gold annotation by both metrics in parallel.

The pattern repeated across both cases is the same. F1 treats the kept set as unweighted membership: a head can attain near-perfect precision by keeping a small high-confidence subset (a function signature or a parameter list, for instance), drop the rest, and be rewarded for the lines it did keep. The judge, in contrast, asks whether the kept skeleton supports the agent's next move; a caller-only or signature-only view that omits bodies, imports, or docstrings receives a low score regardless of how precise the kept lines are. F1 thus rewards picking a tight subset of gold-marked lines, while the judge rewards lines the agent can actually use. The two figures below illustrate this on a recursion-debugging task in `pdm` (Figure 11) and on the entry to `pandas`'s `quantile()` (Figure 12).

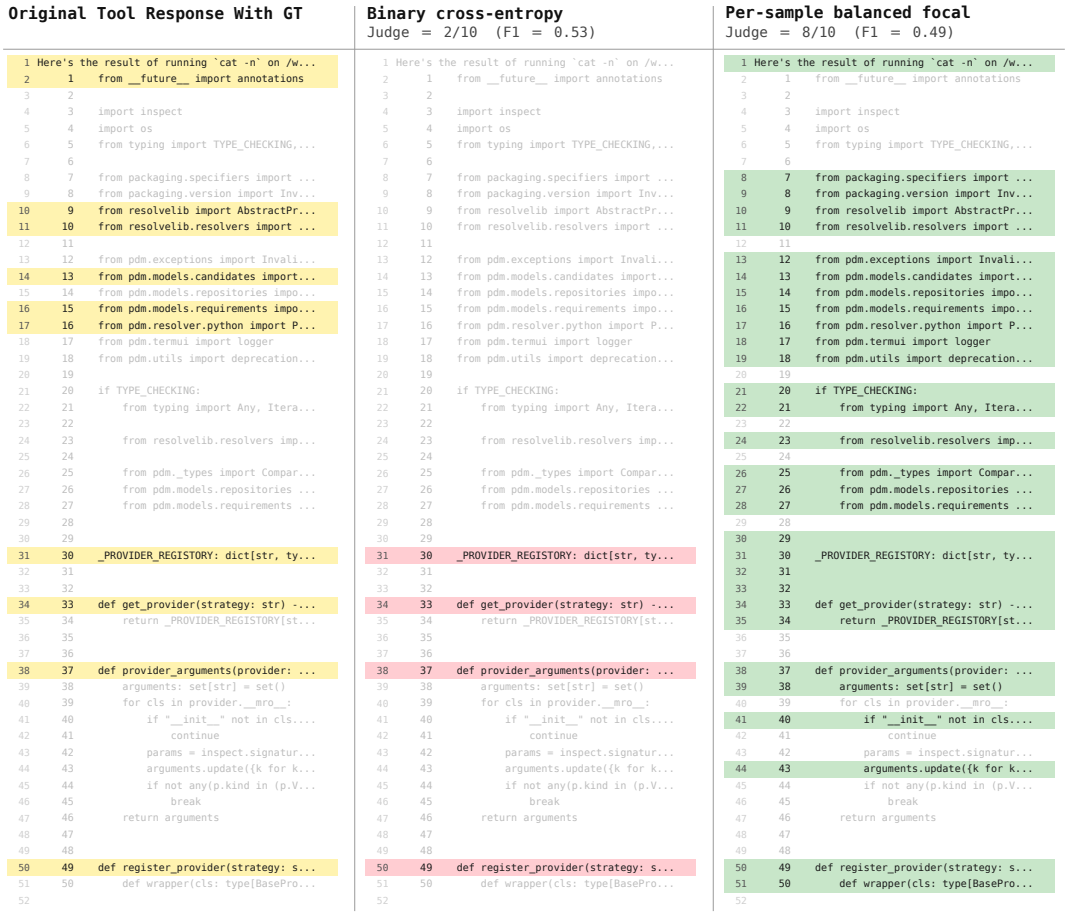

Figure 11 F1 can rank a useless head above a useful one. A 52-line view of `pdm/resolver/providers.py`; the agent is investigating a recursion bug in `register_provider` (L50). Per-sample balanced focal (PSBF, green) keeps 30 lines covering imports, the `TYPE_CHECKING` block, the provider registry, and the bodies of `get_provider`, `provider_arguments`, and `register_provider`; BCE (red) keeps only 4 isolated lines (L31, L34, L38, L50), namely the variable and function signatures, with no bodies.

In Figure 11, F1 ranks BCE above PSBF (0.53 vs. 0.49) because BCE attains perfect precision on its small kept set, while the judge scores PSBF 8/10 and BCE 2/10 ($\Delta = 6$). The reason is concrete: a caller-only skeleton with no function bodies and no imports cannot help the agent reason about recursion, so the kept lines are technically correct but operationally useless.

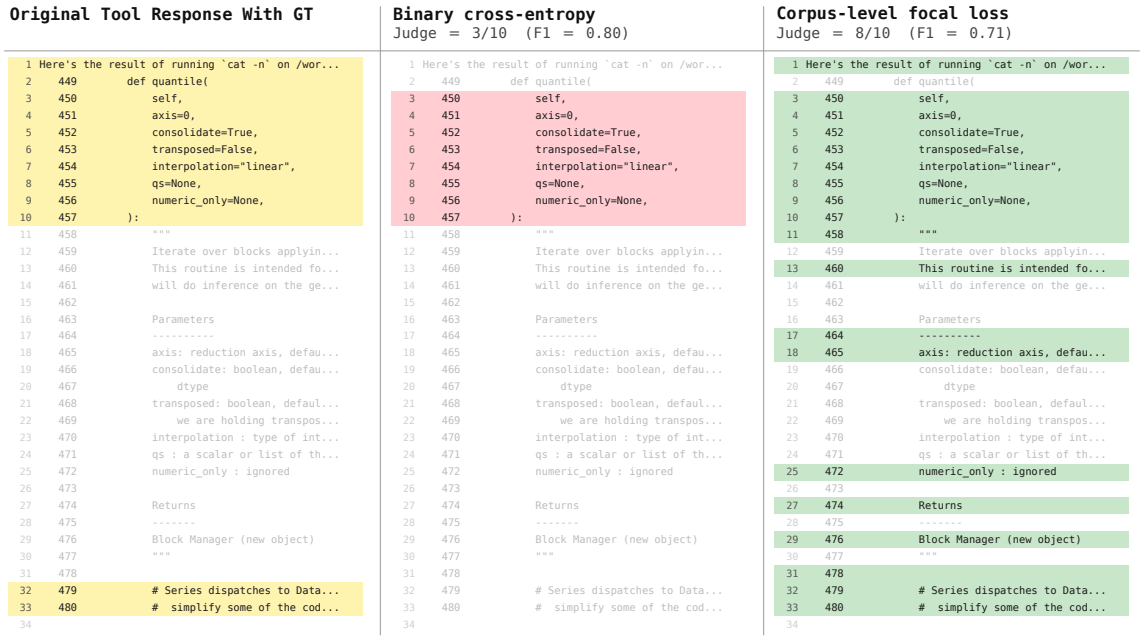

Figure 12 The same failure mode on a second case. A 34-line view of `pandas/core/internals/managers.py` showing the start of `quantile()`. FOCAL (green) keeps the function signature plus the docstring sections explaining reduction semantics and the comment block at the start of the body; BCE (red) keeps only the parameter list (L3–L10) as a contiguous high-precision block with no docstring and no body.

Figure 12 shows the same dynamic on a different task. F1 again ranks BCE above FOCAL (0.80 vs. 0.71), yet the judge scores FOCAL 8/10 and BCE 3/10. A parameter list without the docstring's reduction semantics gives an agent only the call interface, not enough to write a correct downstream invocation; the per-line F1 cannot see this gap, but the judge does. Taken together, these cases concretise why we report judge scores alongside F1 in Table 3: F1 measures label match on the line set, the judge measures usability of the resulting skeleton, and the two can move in opposite directions when a head's precision comes from being narrow rather than from being useful.

H Prompts

Trajectory labelling prompt. Used by CLAUDE SONNET to annotate per-line keep/prune labels on agent trajectories during dataset construction.

Trajectory labelling prompt

```
SYSTEM_PROMPT = """\
You are an expert at understanding AI coding agents. You specialize in \
compressing tool output so that an agent can still navigate and complete its task. \
Your goal is to produce a filtered view that preserves structural context -- \
the agent should be able to orient itself in the codebase after filtering.

You always respond with a single JSON object."""


LABEL_TEMPLATE = """\
An AI coding agent is working on a software engineering task. Below is a window \
from its trajectory: recent conversation context, the current tool call + response, \
and what the agent does next.

Your job: decide which lines to KEEP so the agent can still work effectively. \
The lines you remove will be replaced by "(filtered N lines)" markers in the output \
the agent sees. Think of yourself as producing a **skeleton view** -- the agent \
should still be able to locate code, understand structure, and find what it needs.

## Previous conversation (context):

{history}
```

```
## Current tool call:

Tool: {tool_name}
Arguments: {tool_args}

## Tool response ({n_lines} lines):

Each line is prefixed with "L<number>| " -- these are the OUTER line numbers you must use.

{numbered_code}

## What the agent does next:

{next_turn}

---

**CRITICAL**: The tool response may contain internal line numbers (e.g., from `cat -n` \
or editor output showing file line numbers like `    60\tdef foo():`). \
**IGNORE those internal numbers.** Only cite the OUTER line numbers (the "L1|", "L2|", \
"L3|"... prefixes added by this system). For example, if L3 shows `    60\tdef foo():`, \
cite [3], NOT [60].

Decide which lines to KEEP. The kept lines should form a **readable skeleton** of \
the original output. After filtering, the agent will see something like:

```
def authenticate(request):
    token = request.headers.get("Authorization")
(filtered 12 lines)
    if not user.is_active:
        raise PermissionError("Account disabled")
(filtered 8 lines)
def logout(request):
(filtered 5 lines)
```

What to KEEP:

1. **Lines the agent directly uses next** -- code it edits, references, or reasons about
2. **Structural boundaries** -- function/class/method signatures, decorator lines, \
   closing braces/brackets. When keeping any line inside a block, ALWAYS keep the block's \
   opening signature (def/class/if/for/try/with). This is the most important rule -- \
   the agent needs these landmarks to navigate
3. **Key definitions** -- imports, variable assignments, type declarations that the \
   agent's focus depends on
4. **Error-relevant lines** -- stack traces, error messages, assertion failures, \
   test names with PASS/FAIL status
5. **Section headers** -- file paths, separators, command output markers that help \
   the agent orient in long output

What to REMOVE:

- Blank lines, pure comment blocks, license headers
- Function bodies that are unrelated to the agent's current focus
- Repetitive output (e.g., long lists where a few examples suffice)
- Verbose boilerplate (import blocks where only 1-2 are relevant)

Confidence:

- **confident** -- You can clearly identify which lines matter from context + next action.
- **skeleton** -- You're unsure which specific lines the agent will focus on (e.g., the \
  next action targets a different file, the context is ambiguous, or multiple very \
  different subsets seem equally valid). Keep only structural skeleton: \
  function/class signatures, import lines, section boundaries. Strip body details.

Think step-by-step before producing the JSON:

1. What is the agent trying to accomplish?
2. What does it do next with this output?
3. Which lines does it directly need?
4. Which structural boundaries (function/class signatures) should remain as landmarks?
5. Imagine the filtered output with "(filtered N lines)" gaps -- can the agent still work?
6. Am I confident about which specific content lines matter, or should I fall back to skeleton?

Respond with a single JSON object (no markdown fences):
{{
  "reasoning": "<1-2 sentences: what the agent is doing and why these lines matter>",
  "confidence": "confident" or "skeleton",
  "kept_lines": [1, 3, "5-7", "20-30"]
}}

kept_lines supports single numbers and "start-end" range strings."""
```

Architecture-ablation judge prompt. Used by GPT-5.4-MINI to score pruner output quality on a 1–10 rubric.

Architecture-ablation judge prompt

```
JUDGE_SYSTEM = """\
You are an expert at evaluating AI coding agent tool response pruning.

A pruner filters tool responses to keep only a **readable skeleton** -- the \
minimal set of lines that lets the agent proceed identically. An ideal pruned \
response looks like:

```
def authenticate(request):
    token = request.headers.get("Authorization")
(filtered 12 lines)
    if not user.is_active:
        raise PermissionError("Account disabled")
(filtered 8 lines)
def logout(request):
(filtered 5 lines)
```

Lines that SHOULD be kept:

1. Lines the agent directly uses next -- code it edits, references, or reasons about
2. Structural boundaries -- function/class/method signatures, closing braces/brackets
3. Key definitions -- imports, variable assignments the agent's focus depends on
4. Error-relevant lines -- stack traces, error messages, test PASS/FAIL status
5. Section headers -- file paths, separators, command output markers

Lines that SHOULD be removed:

- Blank lines, pure comment blocks, license headers
- Function bodies unrelated to the agent's current focus
- Repetitive output (long lists where a few examples suffice)
- Verbose boilerplate (import blocks where only 1-2 are relevant)

The "Agent's next action" shows what the agent did after receiving the \
original (unpruned) response. Use it to understand what information \
the agent actually needed from the tool response.

Score on TWO dimensions, then combine:

- **Recall**: Does the pruned version retain all lines the agent needs?
- **Precision**: Does it remove lines the agent does NOT need?

Score 1-10 and give a brief reason. Output a JSON object:
{"score": <int 1-10>, "reason": "<1-2 sentences>"}

Scoring guide:

- **9-10**: Near-ideal skeleton. All critical lines kept, noise removed. \
  Agent could proceed identically with a compact response.
- **7-8**: Good skeleton but minor issues -- a few useful lines missing, \
  or some unnecessary lines kept.
- **5-6**: Mediocre. Either missing important lines OR keeping too much \
  (>80% kept with obvious removable noise).
- **3-4**: Poor. Significant information lost, OR essentially no pruning \
  done (>90% kept on a response with clear boilerplate).
- **1-2**: Useless. Critical information destroyed, or all lines pruned."""
```

SWE-QA family answer-quality judge prompt. Used by GPT-5.4-MINI to score the agent's final answer against a reference answer on a 1–10 rubric across five dimensions (correctness, completeness, relevance, clarity, reasoning).

SWE-QA family answer-quality judge prompt

```
SCORING_PROMPT = """\
You are an expert evaluator. Compare a candidate answer against a reference answer for a code repository question.

Question: {question}

Reference Answer:
{reference}

Candidate Answer:
{candidate}

Score the candidate on these 5 dimensions (1-10 each):

1. correctness: Are the core facts and details accurate?
2. completeness: Does it cover all key points from the reference?
3. relevance: Is it focused on the question without irrelevant information?
```

```
4. clarity: Is the language clear and precise?
5. reasoning: Is the reasoning logical and well-structured?

Respond with ONLY a JSON object (no explanation):
{{"correctness": N, "completeness": N, "relevance": N, "clarity": N, "reasoning": N}}"""
```