FUNCTION-AWARE FILL-IN-THE-MIDDLE AS MID-TRAINING FOR CODING AGENT FOUNDATION MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents must integrate external tool returns into ongoing reasoning. We draw a structural analogy between the *action* $\rightarrow$ *observation* $\rightarrow$ *continuation* loop of a coding agent and a function call site, where a caller binds arguments, a callee returns a value computed elsewhere, and downstream code consumes that value. This analogy motivates learning from function-level dependencies in ordinary code through *function-aware fill-in-the-middle (FIM) mid-training*: an infilling objective that masks functions selected via program dependency graph analysis and a complexity–inferability double criterion, with teacher-generated rationale-plus-implementation targets. We mid-train Qwen2.5-Coder-Instruct (7B/14B) and Qwen3-8B on a 2.6B-token decontaminated corpus drawn from 968 GitHub repositories, then apply existing agentic post-training pipelines. Mid-training improves SWE-Bench-Verified by $+2.8/+3.0$ at 7B/14B and by $+3.2$ on Qwen3-8B; SWE-Bench-Lite gains are $+3.7/+4.0/+5.3$ on the same models. The improvement holds across two post-training pipelines (R2E-Gym, SWE-Smith) and extends to Qwen3-8B with SWE-Lego. Beyond in-domain gains, mid-training improves final-agent performance on non-agent coding (e.g., LiveCodeBench) and non-coding tool-use benchmarks (τ-bench, BFCL). A single Python-derived corpus thus improves the resulting agents across coding and tool-use tasks.

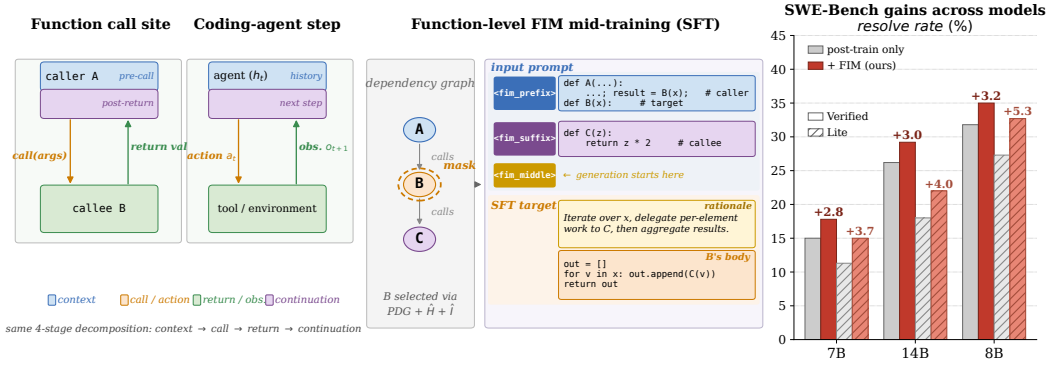

Figure 1: **Left:** A function call and a coding-agent step follow analogous stages: context, call/action, return/observation, and continuation. **Middle:** We exploit this analogy via function-aware FIM mid-training. A function B is selected from the program dependency graph using complexity ($\hat{H}$) and inferability ($\hat{I}$) scores; the model is then mid-trained to fill in B's body together with a CoT rationale, conditioned on the masked file in a chat prompt; FIM token labels are schematic. **Right:** Mid-training yields consistent gains across both Qwen2.5-Coder-Instruct (7B, 14B) and Qwen3 (8B) on SWE-Bench-Verified (solid bars) and SWE-Bench-Lite (hatched bars).

1 INTRODUCTION

Coding agents that resolve real software engineering issues have moved from research demos to deployed systems (Jimenez et al., 2023; Yang et al., 2024; Wang et al., 2024). Their progress, however, has been driven almost entirely by scaling synthetic agent-trajectory data during post-training: pipelines such as SWE-Gym (Pan et al., 2024), R2E-Gym (Jain et al., 2025), and SWE-

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.

Smith (Yang et al., 2025b) curate or synthesize trajectories that imitate human or LLM behavior on issue-resolution tasks. These pipelines typically start from code LLMs trained with next-token prediction and fill-in-the-middle (FIM) objectives (Hui et al., 2024; Guo et al., 2024; Li et al., 2023). The Qwen2.5-Coder-Instruct checkpoints used in this work already receive structure-aware infilling supervision: their instruction tuning includes FIM examples constructed by extracting AST-based logical blocks (Hui et al., 2024). We study whether a dedicated *mid-training* stage that concentrates on function dependencies further improves these checkpoints before agent-specific training.

Our motivation is a structural analogy between function calls and agent interaction (Figure 1, left). An agent maintains a history h_t, samples an action $a_t \sim \pi(a_t \mid h_t)$, receives an externally produced observation o_{t+1}, and continues from the updated trace. A function call site similarly connects pre-call context, a call, a returned value, and downstream usage. This analogy suggests using ordinary code to train models to infer behavior from surrounding dependencies. Function-level FIM conditions implementation generation on the visible signatures, callers, helpers, and downstream code. We view this auxiliary task through the lens of representation learning. Under conditional-independence and non-degeneracy assumptions, reconstruction and language-modeling objectives can retain latent information useful for downstream prediction (Lee et al., 2021; Wei et al., 2021). In code, candidate latent factors include function contracts, data transformations, and caller–callee relationships. Function-aware infilling provides supervision for inferring these factors from context. Our hypothesis is that this supervision provides a useful initialization for subsequent agentic learning; its transfer to repository repair and tool use is evaluated empirically in Section 3.

Our recipe builds on FIM (Bavarian et al., 2022) with three design choices. First, we select complete functions and connected function groups using program dependency graphs and a base-model-agnostic complexity–inferability criterion (Section 2.3). Second, teacher-generated rationales precede their corresponding implementations in the training response (Section 2.4). Third, we concentrate this supervision in a dedicated stage immediately before agentic post-training. Together, these choices target contextual reasoning about function behavior within existing code.

We evaluate this recipe along three axes of robustness. *On the Qwen2.5-Coder-Instruct series*, mid-training improves SWE-Bench-Verified by $+2.8$ and $+3.0$ points at 7B and 14B respectively. *Across two post-training pipelines*, mid-training improves both R2E-Gym and SWE-Smith on the same 7B base, with the SWE-Smith pairing yielding $+5.3$ points on SWE-Bench-Verified. *On Qwen3-8B*, mid-training paired with SWE-Lego yields a $+3.2$-point gain on SWE-Bench-Verified. A second set of results examines broader capabilities of the final coding agent. At 14B, R2E-Gym post-training reduces LiveCodeBench, BFCL, and τ-bench scores by 13.1, 7.4, and 2.3 points relative to the Instruct base. Adding FIM mid-training before the same post-training improves these scores by $+11.1$, $+2.4$, and $+3.9$, respectively, and raises performance on all six additional benchmarks (Table 2). The mid-training tasks are derived from Python repositories without tool-use trajectories. These results demonstrate benefits across task families after agentic post-training.

Contributions. **(1)** Framing function call sites as the internet-scale analogue of the agent action–observation–continuation loop, motivating function-granularity FIM as a code-derived prior for agent capability. **(2)** A function-aware FIM mid-training pipeline combining program dependency graph analysis, a base-model-agnostic complexity–inferability double criterion, and rationale–implementation training targets. **(3)** Robustness validation along three axes—two model sizes (Qwen2.5-Coder-Instruct 7B/14B), two post-training pipelines (R2E-Gym, SWE-Smith), and one alternative base (Qwen3-8B with SWE-Lego)—with consistent in-domain gains in every configuration. **(4)** Cross-task evaluation showing that the same Python-derived mid-training corpus improves final-agent scores on τ-bench, BFCL, and LiveCodeBench after agentic post-training. **(5)** Release, upon publication, of the 968-repository decontaminated corpus (400K FIM samples, 2.6B tokens), the selection pipeline, and mid-training checkpoints.

2 METHOD

2.1 MOTIVATION: FUNCTION CALLS AS AGENT-LIKE STRUCTURES

Function calls mirror the coding-agent loop (Figure 1, left): pre-call context, call, return, and downstream usage align with history, action, observation, and continuation. A masked function participates in both sides of this analogy: it invokes visible helpers and returns values consumed by

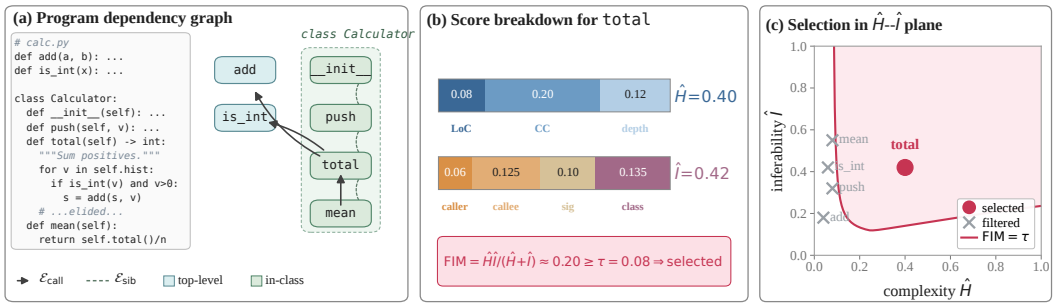

Figure 2: Function-aware FIM target selection on a small calculator example. **(a)** Program dependency graph parsed from the AST: solid arrows are call edges $\mathcal{E}_{\text{call}}$, dashed lines are sibling edges $\mathcal{E}_{\text{sib}}$ between same-class methods. **(b)** Stacked bars decompose the complexity score $\hat{H} = 0.40$ (code length, branching, and nesting) and the inferability score $\hat{I} = 0.42$ (four source-code cues) for `Calculator.total`. Their combined score is approximately 0.20, above the selection threshold of 0.08. **(c)** Candidate functions and the FIM score threshold.

visible callers. Unlike an agent, infilling never conditions on a realized return value; whether the shared component transfers is tested in Section 3. Function-aware FIM uses these dependencies as context for reconstruction, selecting targets by program structure and contextual predictability.

2.2 Data Collection and Decontamination

We curate a corpus of 968 Python repositories from GitHub. Starting from $\sim$ 2,000 candidates retrieved by combining a star-count threshold with ten topic queries, we manually prioritize self-contained implementations and algorithmic code with rich logical dependencies, while deprioritizing overly engineered codebases. We exclude the source repositories of SWE-Bench (Jimenez et al., 2023) and their known forks from the mid-training corpus. The resulting $\approx$ 400K FIM samples comprise 320K single-function, 60K pair, and 20K triple targets, for a total of $\approx$ 2.6B tokens as measured with the Qwen2.5-Coder tokenizer. Permissive licenses (MIT, Apache-2.0, BSD) cover over 80% of the repositories; the license inventory and topic distribution are given in Appendix A.

2.3 Function-Aware FIM Target Selection

Each mask target consists of a single function or a connected group of 2–3 functions within one file. We mask their bodies, preserving their signatures and the rest of the file as context. Selection has four stages: dependency-graph construction, complexity scoring, inferability scoring, and threshold-based selection. Figure 2 illustrates the procedure on a `Calculator` class with two top-level helpers.

2.3.1 Program Dependency Graph

We parse each file's AST into function nodes $\mathcal{V}$, including top-level functions and class methods. *Call edges* $\mathcal{E}_{\text{call}}$ link callers to their callees; *sibling edges* $\mathcal{E}_{\text{sib}}$ link methods in the same class to capture potential coupling through shared instance state.

2.3.2 Complexity Score $\hat{H}$

The complexity score $\hat{H}$ combines lines of code, McCabe cyclomatic complexity (branching), and maximum control-flow nesting depth using a weighted sum of normalized, capped values. It favors substantive implementation logic while limiting the influence of unusually large functions.

2.3.3 Inferability Score $\hat{I}$

A target should be recoverable from its surrounding context. The inferability score $\hat{I}$ combines four cues: argument specificity at call sites, the number of intra-file callees, type annotations and descriptive names in the signature, and sibling methods in the same class. Target docstrings are masked with the bodies and receive no inferability bonus. Rule-based scoring makes target selection

independent of a reference model and the selected corpus reusable across base models; a teacher judge enters only in the later quality filter of Section 2.4.

2.3.4 SINGLE-FUNCTION SELECTION

We combine complexity and inferability with a harmonic-mean-like score that is high only when both are high, damped by a difficulty penalty when complexity substantially exceeds inferability (Eqs. 4–6 in Appendix B.2.3); this favors challenging targets supported by sufficient context. We retain functions with 10–200 source lines and exclude trivial implementations and Python special methods such as `__init__`, and skip files outside 50–1,800 lines or functions with $\hat{H} < 0.15$. The remaining candidates are ranked by their combined score, retaining those at or above 0.08 (all thresholds are listed in Appendix B.2.3).

2.3.5 MULTI-FUNCTION GROUP SELECTION

To expose dependencies across related functions (Jimenez et al., 2023), our main recipe also masks pairs and triples linked by calls or class relationships. We recompute each member's inferability under joint masking, excluding signals from the other hidden bodies, and average member complexities and recomputed inferabilities. The group score combines these quantities with a difficulty penalty and a coupling factor based on call edges, sibling relations, and shared instance state. We retain high-scoring groups subject to size and coupling thresholds, selecting them greedily without overlapping functions. Section 3.4 compares masking granularities.

For a caller–callee pair, joint masking removes the call site inside the caller that would otherwise help reconstruct the callee. Both signatures and external callers remain visible. The hidden call edge contributes to coupling, while the surviving context determines the recomputed inferability. Joint completion therefore requires the two generated bodies to agree on argument use and returned values.

2.4 CHAIN-OF-THOUGHT AUGMENTATION

For our default CoT-augmented recipe, each selected target passes through a three-stage pipeline. **Generate.** Gemini-3-Flash sees only the masked file and produces a step-by-step rationale together with a candidate function implementation, with no access to the original body. **Filter.** A separate Gemini-3-Flash judge scores the candidate implementation against the original code for feasibility, correctness, executability, API usage, readability, and completeness; the ~ 400K samples that pass the judge's discard rule (Appendix C.2) are retained. **Format.** We use chat-formatted SFT, with the masked file in the user message and the rationale followed by the predicted body in the assistant response, applying autoregressive loss to that response. This supervision mirrors an agent's think-then-act structure. In the *self-CoT* variant (Section 3.4), the model under training generates both parts of the target, so no external teacher is involved in constructing its targets.

For masked prompt x, rationale r, and code b, the per-example objective is

$$\mathcal{L}_{\mathrm{FIM}}(x, r, b) = -\log p_\theta(r \mid x) - \log p_\theta(b \mid x, r). \tag{1}$$

The implementation term trains the model to use the preceding rationale while satisfying the visible code context. The *no CoT* ablation removes r from the same retained Gemini samples, yielding $-\log p_\theta(b \mid x)$ with identical inputs and code targets.

Appendices A–C detail corpus statistics, selection rules, sample construction, and training.

3 EXPERIMENTS

3.1 SETUP

Benchmarks. We evaluate on three groups. *Coding-agent benchmarks*: SWE-Bench-Verified (500) and SWE-Bench-Lite (300) (Jimenez et al., 2023), our primary in-domain target. *Non-agent coding benchmarks*: LiveCodeBench (Jain et al., 2024), OJBench (Wang et al., 2025), and FullStackBench-EN (Cheng et al., 2024b); these probe pure code generation and serve as a regression check. *Tool-use and OOD agent benchmarks*: Terminal-Bench (Merrill et al., 2026), τ-bench (Yao et al., 2024), and

BFCL (Patil et al., 2025); the latter two contain no Python code-editing trajectories and test whether the function-call inductive bias transfers across task families.

Mid-training and post-training pipeline. We mid-train Qwen2.5-Coder-Instruct (7B/14B) (Hui et al., 2024) and Qwen3-8B (Yang et al., 2025a) on the selected FIM corpus using chat-formatted SFT with autoregressive loss on the assistant response (rationale plus body) and each model's native chat template. We then apply the agentic post-training recipes from R2E-Gym (Jain et al., 2025) or SWE-Smith (Yang et al., 2025b) for the Qwen2.5-Coder runs, and SWE-Lego (Tao et al., 2026) for Qwen3-8B. For Qwen3-8B we use 2 SWE-Lego epochs (officially 4) to prevent overfitting.

Evaluation protocol. We evaluate the final checkpoint of each pipeline. Results measured in our setup (Tables 1 and 3) are means over three evaluation runs, with population standard deviations in parentheses. Officially reported results retain their original reporting protocol. Percent signs are omitted in table cells. *Baseline* is base + post-training; *ours* is base + FIM mid-training + identical post-training. All agent results are measured after agentic post-training, evaluating the benefit of mid-training in the completed pipeline. The agent harness is fixed by the post-training pipeline: R2E-Gym uses an OpenHands fork (Wang et al., 2024; Jain et al., 2025), SWE-Smith uses SWE-agent (Yang et al., 2024), and the Qwen3-8B runs use OpenHands directly (SWE-Lego trajectories exceed the 32K context of the Qwen2.5-Coder models).

Models and baselines. For Qwen2.5-Coder-Instruct at 7B/14B (Hui et al., 2024) we report (i) the instruction-tuned base, (ii) the R2E-Gym baseline under our protocol, (iii) the originally reported R2E-Gym numbers, (iv) FIM mid-training followed by R2E-Gym post-training, and at 7B the analogous SWE-Smith (Yang et al., 2025b) variants. The 7B R2E-Gym baseline is the official checkpoint run with the official scripts in our environment; it scores 15.00 on Verified (std 1.50) and does not recover the reported 19.00. Baseline cells deviate from published scores by up to four points in both directions (Table 1). Qwen3-8B uses the SWE-Lego (Tao et al., 2026) pipeline.

3.2 Main Results on SWE Agent Benchmarks

Table 1 summarizes in-domain agent results.

Consistent gains on the Qwen2.5-Coder-Instruct series. Holding the post-training pipeline fixed at R2E-Gym, FIM mid-training improves SWE-Bench-Verified by +2.80 on 7B-Instruct and +3.00 on 14B-Instruct, with matching directional gains on SWE-Bench-Lite (+3.67/ + 4.00). The same mid-training corpus and recipe improve both model sizes on both benchmarks.

Transfer across post-training pipelines. Replacing R2E-Gym with SWE-Smith on the same 7B base yields +5.27 points on Verified, alongside +2.80 under R2E-Gym. On Lite, the respective gains are +0.44 and +3.67. Mid-training improves both benchmarks under both post-training pipelines.

Transfer to Qwen3-8B with SWE-Lego. Switching to Qwen3-8B paired with SWE-Lego, mid-training improves Verified by +3.20 and Lite by +5.33, comparable to the Qwen2.5-Coder-Instruct gains. The same corpus thus benefits both Qwen generations and a third post-training pipeline.

3.3 Broader Capabilities and Cross-Domain Transfer

Table 2 evaluates six additional benchmarks at 14B with R2E-Gym, comparing the Instruct base, post-training alone, and FIM mid-training followed by post-training.

Broader capabilities after agentic post-training. At 14B, R2E-Gym alone reduces scores on all five non-agent coding and tool-use benchmarks relative to the Instruct reference: LiveCodeBench by 13.10, BFCL by 7.40, FullStackBench-EN by 6.08, τ-bench by 2.30, and OJBench by 2.40. Averaged across the six benchmarks the post-trained model loses 4.81 points relative to Instruct.

FIM mid-training improves all six additional benchmarks. Adding FIM mid-training before the same post-training raises LiveCodeBench by +11.10, OJBench by +1.94 (within 0.46 of the Instruct reference), and FullStackBench-EN by +0.53. The six-benchmark average rises from 16.04 to 19.56 (+3.52 over post-training only), alongside a +3.00-point gain on SWE-Bench-Verified (Table 1). The agent thus improves across repository repair, code generation, and tool use.

Cross-domain transfer to non-coding tool use. τ-bench and BFCL contain no Python code-editing data, and our mid-training corpus contains no tool-use trajectories. Mid-training improves τ-bench

Table 1: Main results on coding agent benchmarks. All numbers are percentages; results measured in our setup are averaged over three evaluation runs, with population standard deviations in parentheses. *Average* is the unweighted mean of Verified and Lite. Our recipe is labelled in bold. Within each post-training block, bold cells mark the best and underlined cells the second-best score per column; rows tagged *officially reported* are quoted from the original publications (R2E-Gym also reports standard deviations) and included in this ranking. *Re-evaluated*: released checkpoint scored in our environment; *reproduced*: retrained by us with the official scripts.

Setting	SWE-Bench-Verified	SWE-Bench-Lite	Average
Qwen2.5-Coder-7B-Instruct			
— (no agentic training)	1.80 (±1.30)	1.00 (±0.98)	1.40
+ R2E-Gym (Jain et al., 2025) (re-evaluated)	15.00 (±1.50)	<u>11.33</u> (±1.19)	13.17
+ R2E-Gym (officially reported)	**19.00** (±1.00)	11.00 (±0.80)	<u>15.00</u>
+ FIM-Midtrain + R2E-Gym	<u>17.80</u> (±1.40)	**15.00** (±1.09)	**16.40**
Δ (ours vs. re-evaluated)	+2.80	+3.67	+3.23
+ SWE-Smith (Yang et al., 2025b) (reproduced)	12.33 (±1.20)	<u>14.22</u> (±1.40)	13.28
+ SWE-Smith (officially reported)	<u>15.20</u>	11.70	<u>13.45</u>
+ FIM-Midtrain + SWE-Smith	**17.60** (±1.30)	**14.67** (±0.98)	**16.13**
Δ (ours vs. reproduced)	+5.27	+0.44	+2.86
Qwen2.5-Coder-14B-Instruct			
— (no agentic training)	4.00 (±1.61)	2.67 (±0.94)	3.33
+ R2E-Gym (reproduced)	26.20 (±1.40)	18.00 (±1.19)	22.10
+ R2E-Gym (officially reported)	<u>26.80</u> (±1.40)	<u>20.67</u> (±0.70)	<u>23.73</u>
+ FIM-Midtrain + R2E-Gym	**29.20** (±1.50)	**22.00** (±1.25)	**25.60**
Δ (ours vs. reproduced)	+3.00	+4.00	+3.50
Qwen3-8B			
— (no agentic training)	7.60 (±1.18)	5.78 (±0.87)	6.69
+ SWE-Lego (Tao et al., 2026) (reproduced)	<u>31.80</u> (±1.02)	<u>27.33</u> (±1.09)	<u>29.57</u>
+ FIM-Midtrain + SWE-Lego	**35.00** (±1.50)	**32.67** (±1.25)	**33.83**
Δ (ours vs. reproduced)	+3.20	+5.33	+4.27

Table 2: Broader capabilities and cross-domain transfer at 14B with R2E-Gym. Values are percentages; bold marks the better trained variant in each column. Instruct provides reference scores.

	Non-agent coding			Agent OOD	Tool use		
Setting	LiveCode	OJBench	FSB-EN	Terminal	τ-bench	BFCL	Avg
Instruct Model	37.20	5.20	53.80	0.00	5.70	23.20	20.85
+ R2E-Gym	24.10	2.80	47.72	2.41	3.40	15.80	16.04
+ FIM Mid-Train + R2E-Gym	**35.20**	**4.74**	**48.25**	**3.66**	**7.30**	**18.20**	**19.56**
Δ (vs. R2E-Gym only)	+11.10	+1.94	+0.53	+1.25	+3.90	+2.40	+3.52

by +3.90 and BFCL by +2.40 over post-training alone; Terminal-Bench also improves by +1.25. The same Python-derived mid-training corpus thus improves both coding-agent performance and non-coding tool use, demonstrating transfer across task families.

3.4 Ablation Studies

We study CoT augmentation, function selection, mask granularity, and mask direction on Qwen2.5-Coder-7B-Instruct with R2E-Gym post-training. The first three ablations share a mid-training-free baseline and a controlled 200K-target budget. The mask-direction experiment keeps callee bodies visible and masks call sites and downstream continuations. We also compare against AST-FIM (Gong et al., 2025) using its published infilling setup, followed by R2E-Gym post-training; it samples syntax-subtree spans at random granularity, without dependency-based scoring or rationale targets.

Function-aware FIM benefits from CoT augmentation. Block (A) compares CoT variants with function selection fixed. The *no CoT* variant uses the same Gemini-generated implementations as the Gemini-CoT variant, with their rationales removed. It reaches an average of 14.31, improving

Table 3: Ablations and AST-FIM comparison on the 7B model with R2E-Gym post-training (means over three evaluation runs; population standard deviations in parentheses). **(A)** CoT augmentation: the same Gemini-generated implementations without or with their accompanying rationales, plus rationale–implementation pairs generated by the model itself (*self-CoT*). **(B)** function-selection algorithm. **(C)** mask granularity (single vs. multi-function groups). **(D)** AST-FIM following its published setup (Gong et al., 2025). **(E)** inverted masking of call sites and downstream continuations with callee bodies visible. Blocks (A)–(C) and (E) share the baseline (*w/o mid-train*) and a controlled 200K-target budget; CoT is fixed to Gemini-3-Flash in (B), (C), and (E); (D) uses the same corpus. Bold marks the best configuration in each of (A)–(C); the $80\%/15\%/5\%$ mixture in (C) is the recipe used in our main results (Table 1), which is trained on the full corpus rather than the 200K budget here.

Setting	SWE-Bench-Verified	SWE-Bench-Lite	Average
(A) Chain-of-thought augmentation (selection fixed to "full")			
w/o mid-train (baseline)	15.00 ($\pm$1.50)	11.33 ($\pm$1.19)	13.17
+ FIM, Gemini-3-Flash, no CoT	16.07 ($\pm$1.09)	12.56 ($\pm$1.03)	14.31
+ FIM, self-CoT	16.40 ($\pm$1.07)	13.33 ($\pm$1.25)	14.87
+ FIM, Gemini-3-Flash CoT	**17.00** ($\pm$1.30)	**14.22** ($\pm$1.10)	**15.61**
(B) Function-selection algorithm (CoT fixed to Gemini-3-Flash)			
w/o mid-train (baseline)	15.00 ($\pm$1.50)	11.33 ($\pm$1.19)	13.17
Random	15.27 ($\pm$1.16)	12.56 ($\pm$1.26)	13.91
Gemini-selected	16.40 ($\pm$1.47)	13.67 ($\pm$0.94)	15.03
PDG only	16.07 ($\pm$0.84)	13.56 ($\pm$1.59)	14.81
PDG + complexity ($\hat{H}$)	16.47 ($\pm$1.27)	13.56 ($\pm$1.85)	15.01
PDG + inferability ($\hat{I}$)	16.67 ($\pm$1.33)	14.00 ($\pm$1.66)	15.33
Full (PDG + $\hat{H}$ + $\hat{I}$)	**17.00** ($\pm$1.30)	**14.22** ($\pm$1.10)	**15.61**
(C) Mask granularity (selection fixed to "full", CoT fixed to Gemini-3-Flash)			
w/o mid-train (baseline)	15.00 ($\pm$1.50)	11.33 ($\pm$1.19)	13.17
Single-function only	17.00 ($\pm$1.30)	14.22 ($\pm$1.10)	15.61
85% single + 15% pair ($k = 2$)	17.20 ($\pm$1.28)	14.56 ($\pm$0.87)	15.88
95% single + 5% triple ($k = 3$)	17.00 ($\pm$0.98)	14.44 ($\pm$1.55)	15.72
80% single + 15% pair + 5% triple	**17.40** ($\pm$0.91)	**14.78** ($\pm$1.57)	**16.09**
(D) AST-FIM comparison (published setup)			
AST-FIM (Gong et al., 2025)	15.20 ($\pm$1.45)	12.00 ($\pm$1.63)	13.60
(E) Mask direction (callee bodies visible)			
Inverted mask (call sites + continuation)	15.80 ($\pm$1.31)	13.00 ($\pm$1.70)	14.40

over the baseline by $+1.14$. Self-generated rationale–implementation pairs (*self-CoT*) reach 14.87, a 1.70-point gain over the baseline. Gemini-3-Flash pairs yield the highest average of 15.61, a 2.44-point gain over the baseline and 0.74 points above self-CoT. With inputs and implementation targets fixed, retaining the rationales adds 1.30 points to the two-benchmark average.

Dependency-based selection achieves the highest scores. Block (B) varies the selection algorithm with CoT and budget held fixed. *Random* masking reaches 13.91, and *Gemini-selected* reaches 15.03. Restricting candidates to functions with at least one PDG neighbor (*PDG only*) reaches 14.81; adding $\hat{H}$ or $\hat{I}$ on top yields 15.01 and 15.33, respectively. Each scoring component improves on the structural filter, and combining them (*Full*) achieves the highest average of 15.61.

Combining masking granularities gives the best result. Block (C) mixes single-function targets with multi-function groups (Section 2.3.5). Adding 15% pair targets raises the average from 15.61 to 15.88; adding 5% triple targets to single-function FIM gives 15.72. The full $80\%/15\%/5\%$ mix achieves the highest average of 16.09, combining single-function infilling with reconstruction of pairs and triples. These gains complement the Verified multi-function task results in Section 4.2.

Comparison with AST-FIM. AST-FIM in block (D) scores 15.20 on Verified and 12.00 on Lite (average 13.60). Under the matched no-CoT condition, function-aware selection (block (A), 14.31)

is 0.71 higher; with rationales and multi-function groups, the 80%/15%/5% recipe (16.09) is 2.49 higher, with the larger margin on Lite (+2.78 vs. +2.20 on Verified).

Function-body infilling outperforms inverted masking. Block (E) keeps the callee body visible and masks the call site and downstream continuation. This variant scores 15.80 on Verified and 13.00 on Lite (average 14.40), compared with 17.00 and 14.22 for single-function FIM in block (C). Function-body infilling yields +1.20 on Verified and +1.22 on Lite. A masked body exposes both roles of the target (Section 2.1), whereas the inverted mask exposes only the caller side over shorter spans; as the variants differ in span construction, we do not attribute the gap to direction alone.

4 Analysis

We analyze the 14B R2E-Gym agents to connect performance gains with repair behavior and gold-patch structure, comparing post-training alone with FIM mid-training followed by the same post-training. Appendix D provides metric definitions, full statistics on Verified and Lite, and a repair example with successive edits and tool feedback.

4.1 Recovery from Negative Observations

A trajectory contains a *negative observation* if any tool output matches a fixed set of error patterns, such as stack traces, "`No replacement was performed`", or shell errors. The fraction of such trajectories is 88.8% for the baseline and 91.8% for the mid-trained agent. The *recovery rate*—the fraction of error-containing trajectories that end with a passing patch—rises from 24.8% to 28.8% (+4.0 pp; +16% relative; Table 4). Thus, the mid-trained agent more often completes tasks successfully on trajectories containing detected error signals. Mid-training also shifts the agent toward an iterate-and-verify policy, increasing edits per solved task from 3.3 to 7.4 and trajectory length from 15.1 to 23.6 steps. The action distribution (Appendix D.2.2) makes this shift concrete: the share of `search` actions falls from 15.3% to 11.0%, while `execute_bash` rises from 19.5% to 24.6%. The agent allocates more of its interaction budget to scripts, tests, and edits.

On `scikit-learn-26323` (Appendix D.2.4), the baseline finishes in one step with an empty patch. The mid-trained agent makes about 20 edits over 41 steps, continuing through multiple negative tool returns to reach a passing patch.

Table 4: Headline trajectory metrics on SWE-Bench-Verified (14B, R2E-Gym). Each entry is averaged over three independent evaluation runs of the corresponding checkpoint.

Setting	Recovery rate (%)	Edits / solved	Steps / solved	Pass (%)
+ R2E-Gym	24.8	3.3	15.1	26.2
+ FIM-Midtrain + R2E-Gym	**28.8**	**7.4**	**23.6**	**29.2**

4.2 Gains by Gold-Patch Structure

To examine how the gains relate to the scope of the required edits, we stratify the 500 Verified tasks by the shape of the gold patch. On the 88 tasks whose gold patch modifies ≥ 2 functions within a single file, the baseline solves 13.6% on average and ours solves 22.7%, an absolute gain of +9.1 pp—more than 4× the gain on the 341 single-function tasks (+2.1 pp). On multi-file tasks (n=71), both checkpoints solve ∼11.3%; the current FIM targets operate within individual files. The largest per-task gain on Verified occurs in the multi-function single-file bucket, which matches the granularity of our function-group targets. On Lite, the improvement comes from the 246 single-function tasks (+4.9 pp), with unchanged performance on the 54 multi-function tasks (Appendix D.3). Across the two benchmarks, the recipe improves repair success over different distributions of edit scope.

The Verified outcome breakdown records a net increase of 15 solved tasks per run. No-patch failures fall from about 11 to 1, while localization errors fall from about 131 to 126 and patch errors remain near 227. These categories distinguish an empty diff, a nonempty patch with no file overlap with the gold patch, and an overlapping patch that fails tests (Appendix D.2.4).

5 Related Work

Mid-training and continued pretraining. Mid-training specializes models on curated data (Tu et al., 2025; Gururangan et al., 2020). MiniCPM (Hu et al., 2024), OLMo (Groeneveld et al., 2024), and DeepSeek-V3 (Liu et al., 2024a) use curated stages late in pretraining; Code-Llama (Roziere et al., 2023) targets context-length generalization. Studies of data timing (Huang et al., 2026; Akter et al., 2025) motivate our dedicated stage for learning an agent-oriented prior from function dependencies.

Fill-in-the-middle and structure-aware code objectives. FIM (Bavarian et al., 2022; Fried et al., 2022) is widely used in code pretraining (Roziere et al., 2023; Lozhkov et al., 2024; Nijkamp et al., 2022; Hui et al., 2024; Guo et al., 2024). AST-T5 (Gong et al., 2024) and AST-FIM (Gong et al., 2025) mask AST subtrees; Horizon-Length Prediction (Ding et al., 2024) adds planning signals; Instruction-aware FIM (Sun et al., 2025) incorporates developer comments. GraphCoder (Liu et al., 2024b) and DRACO (Cheng et al., 2024a) use program dependencies for repository-level retrieval at inference time. Our recipe selects complete functions using dependency graphs and a complexity–inferability criterion, adds explicit reasoning before the implementation, and concentrates this supervision in a dedicated mid-training stage before agentic post-training.

Coding agent foundation models. SWE-Bench (Jimenez et al., 2023) and its extensions (Zan et al., 2025; Deng et al., 2025) evaluate repository-level repair, while SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2024), and Agentless (Xia et al., 2024) provide different action interfaces. SWE-Gym (Pan et al., 2024), R2E-Gym (Jain et al., 2025), SWE-Smith (Yang et al., 2025b), SWE-Lego (Tao et al., 2026), and Skywork-SWE (Zeng et al., 2025) build trajectory-based post-training pipelines, with further work incorporating RL (Golubev et al., 2025; Wei et al., 2025). Our function-aware FIM mid-training provides an initialization for agentic post-training that improves the resulting agent's issue-resolution performance and broader capabilities.

Distillation from frontier models. Our Gemini-3-Flash rationale–implementation targets build on teacher distillation (Mukherjee et al., 2023; Gandhi et al., 2023; Wei et al., 2023) and CoT distillation (Ho et al., 2023; Hsieh et al., 2023; Chen et al., 2025). Our ablation (Section 3.4) shows gains with both self-generated and Gemini-generated rationale–implementation pairs.

6 Limitations and Discussion

The current recipe derives its training tasks from Python repositories and selects functions within individual files. It relies on explicit function boundaries and in-file dependencies; cross-file groups and code organized as scripts or notebooks require corresponding target definitions. Extending these targets to other languages would broaden the available mid-training corpus.

Our experiments cover three Qwen checkpoints and three agentic post-training pipelines. Broader-capability and trajectory analyses use the 14B R2E-Gym configuration. Additional model families would extend the coverage of these analyses. The default data construction uses Gemini-3-Flash; self-generated rationale–implementation pairs achieve a 14.87 average on Verified and Lite, compared with 15.61 for Gemini-generated pairs (Section 3.4). This comparison motivates studying how teacher choice affects the quality of both rationales and generated implementations.

7 Conclusion

Motivated by the structural analogy between function calls and agent interaction, we introduce function-aware FIM mid-training to learn from dependencies in ordinary source code. This stage combines dependency-based target selection, complexity–inferability scoring, and rationale–implementation supervision. Across three base checkpoints and three post-training pipelines, the recipe delivers consistent gains on coding-agent benchmarks, and the same Python-derived corpus transfers to non-coding tool-use benchmarks (τ-bench, BFCL). The corpus, pipeline, and checkpoints will be released upon publication.

Future work will extend function-aware selection across languages and combine this initialization with reinforcement learning driven by execution results and tool feedback, starting from the released corpus, selection pipeline, and checkpoints.

AI USE STATEMENT

Generative AI tools assisted with manuscript drafting and editing, data processing and filtering, visualization, and implementation. As described in Section 2.4, Gemini-3-Flash generated candidate implementations with rationales and judged completion quality during dataset filtering.

REFERENCES

Syeda Nahida Akter, Shrimai Prabhumoye, Eric Nyberg, Mostofa Patwary, Mohammad Shoeybi, Yejin Choi, and Bryan Catanzaro. Front-loading reasoning: The synergy between pretraining and post-training data. *arXiv preprint arXiv:2510.03264*, 2025.

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

Xinghao Chen, Zhijing Sun, Guo Wenjin, Miaoran Zhang, Yanjun Chen, Yirong Sun, Hui Su, Yijie Pan, Dietrich Klakow, Wenjie Li, et al. Unveiling the key factors for distilling chain-of-thought reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 15094–15119, 2025.

Wei Cheng, Yuhan Wu, and Wei Hu. Dataflow-guided retrieval augmentation for repository-level code completion. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7957–7977, 2024a.

Yao Cheng, Jianfeng Chen, Jie Chen, Li Chen, Liyu Chen, Wentao Chen, Zhengyu Chen, Shijie Geng, Aoyan Li, Bo Li, et al. Fullstack bench: Evaluating llms as full stack coders. arXiv preprint arXiv:2412.00535, 2024b.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Yifeng Ding, Hantian Ding, Shiqi Wang, Qing Sun, Varun Kumar, and Zijian Wang. Horizon-length prediction: Advancing fill-in-the-middle capabilities for code generation with lookahead planning, 2024.

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

Sanchit Gandhi, Patrick Von Platen, and Alexander M Rush. Distil-whisper: Robust knowledge distillation via large-scale pseudo labelling. arXiv preprint arXiv:2311.00430, 2023.

Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Sergey Abramov, Andrei Andriushchenko, Filipp Fisin, et al. Training long-context, multi-turn software engineering agents with reinforcement learning. *arXiv preprint arXiv:2508.03501*, 2025.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. Ast-t5: Structure-aware pretraining for code generation and understanding. *arXiv preprint arXiv:2401.03003*, 2024.

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

Dirk Groeneveld, Iz Beltagy, Evan Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, et al. Olmo: Accelerating the science of language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15789–15809, 2024.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, et al. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence. arXiv preprint arXiv:2401.14196, 2024.

Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A Smith. Don't stop pretraining: Adapt language models to domains and tasks. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pp. 8342–8360, 2020.

Namgyu Ho, Laura Schmid, and Se-Young Yun. Large language models are reasoning teachers. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, pp. 14852–14882, 2023.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alex Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 8003–8017, 2023.

Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. Minicpm: Unveiling the potential of small language models with scalable training strategies. arXiv preprint arXiv:2404.06395, 2024.

Junjie Huang, Jiarui Qin, Di Yin, Weiwen Liu, Yong Yu, Xing Sun, and Weinan Zhang. Remit: Rl-guided mid-training for iterative llm evolution. *arXiv preprint arXiv:2602.03075*, 2026.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2.5-Coder technical report. arXiv preprint arXiv:2409.12186, 2024.

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. arXiv preprint arXiv:2403.07974, 2024.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. arXiv preprint arXiv:2504.07164, 2025.

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 *The twelfth international conference on learning representations*, 2023.

Jason D. Lee, Qi Lei, Nikunj Saunshi, and Jiacheng Zhuo. Predicting what you already know helps: Provable self-supervised learning. In *Advances in Neural Information Processing Systems*, volume 34, pp. 309–323, 2021. URL `https://proceedings.neurips.cc/paper/2021/hash/02e656adee09f8394b402d9958389b7d-Abstract.html`.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. StarCoder: May the source be with you! arXiv preprint arXiv:2305.06161, 2023.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024a.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *arXiv preprint arXiv:2406.07003*, 2024b.

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

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. arXiv preprint arXiv:2601.11868, 2026.

Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Awadallah. Orca: Progressive learning from complex explanation traces of gpt-4. arXiv preprint arXiv:2306.02707, 2023.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. arXiv preprint arXiv:2412.21139, 2024.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code Llama: Open foundation models for code. arXiv preprint arXiv:2308.12950, 2023.

Zhensu Sun, Chengran Yang, Chao Peng, Pengfei Gao, Xiaoning Du, Li Li, and David Lo. Bridging developer instructions and code completion through instruction-aware fill-in-the-middle paradigm. *arXiv preprint arXiv:2509.24637*, 2025.

Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, et al. SWE-Lego: Pushing the limits of supervised fine-tuning for software issue resolving. arXiv preprint arXiv:2601.01426, 2026.

Chengying Tu, Xuemiao Zhang, Rongxiang Weng, Rumei Li, Chen Zhang, Yang Bai, Hongfei Yan, Jingang Wang, and Xunliang Cai. A survey on llm mid-training. *arXiv preprint arXiv:2510.23081*, 2025.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Zhexu Wang, Yiping Liu, Yejie Wang, Wenyang He, Bofei Gao, Muxi Diao, Yanxu Chen, Kelin Fu, Flood Sung, Zhilin Yang, et al. Ojbench: A competition level code benchmark for large language models. arXiv preprint arXiv:2506.16395, 2025.

Colin Wei, Sang Michael Xie, and Tengyu Ma. Why do pretrained language models help in downstream tasks? an analysis of head and prompt tuning. In *Advances in Neural Information Processing Systems*, volume 34, pp. 16158–16170, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/86b3e165b8154656a71ffe8a327ded7d-Abstract.html.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120*, 2023.

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. Toward training superintelligent software agents through self-play swe-rl. *arXiv preprint arXiv:2512.18552*, 2025.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. arXiv preprint arXiv:2505.09388, 2025a.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent–computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652, 2024.

John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. arXiv preprint arXiv:2504.21798, 2025b.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. arXiv preprint arXiv:2406.12045, 2024.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025.

Liang Zeng, Yongcong Li, Yuzhen Xiao, Changshi Li, Chris Yuhao Liu, Rui Yan, Tianwen Wei, Jujie He, Xuchen Song, Yang Liu, et al. Skywork-swe: Unveiling data scaling laws for software engineering in llms. *arXiv preprint arXiv:2506.19290*, 2025.

A CORPUS DETAILS

This appendix expands on the data-collection summary (Section 2.2). We report category coverage, per-property statistics, and the license inventory of the 968 released repositories.

A.1 MANUAL REPOSITORY SELECTION

Manual screening follows two qualitative criteria. **Self-containment.** We favor repositories whose core implementation logic can be understood from local code context. **Algorithmic content and logical complexity.** We prioritize repositories centered on algorithmic implementations with rich logical dependencies, and deprioritize codebases dominated by engineering scaffolding and excessive layers of abstraction. These criteria focus the corpus on self-contained code with substantive logic for function-aware infilling. Figure 3 shows the topic distribution and Table 5 the corpus statistics.

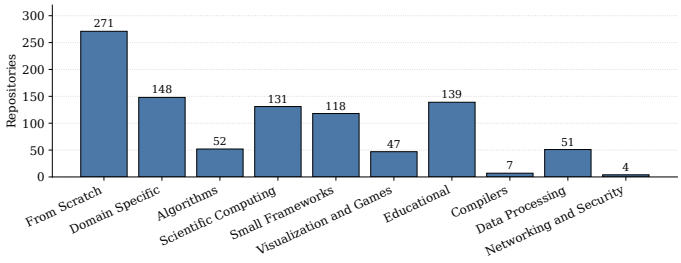

Figure 3: Distribution of the 968 source repositories across ten topic categories. The corpus is dominated by reference implementations, scientific computing, and small frameworks; compiler and networking/security tails are kept by design to maintain coverage diversity.

Table 5: Mid-training corpus statistics after decontamination and quality filtering. Token counts use the Qwen2.5-Coder tokenizer. All numeric quantities reflect the released corpus.

Property	Value
Source GitHub repositories	968
Topic categories	10
Filtered self-contained Python files	$\approx$ 78K
Total FIM samples	$\approx$ 400K
Mid-training token budget	$\approx$ 2.6B
Single-function FIM targets	$\approx$ 320K ($\approx$ 2.0B tokens)
Multi-function targets (k=2)	$\approx$ 60K ($\approx$ 0.4B tokens)
Multi-function targets (k=3)	$\approx$ 20K ($\approx$ 0.2B tokens)
Mean target LoC	$\approx$ 34
Targets with Gemini-3 CoT	100%
SWE-Bench source-repo overlap	0

A.2 LICENSE INVENTORY

Figure 4 shows the license distribution across the 968 released repositories. The corpus is dominated by permissive licenses (MIT, Apache-2.0, BSD-3); the remainder is split among copyleft (GPL, AGPL, LGPL, MPL) and Creative Commons families. Every license in the corpus permits at least non-commercial research use, and all small categories are aggregated in this chart as "Other research-permissive licenses." The per-repository license is released alongside the repository list.

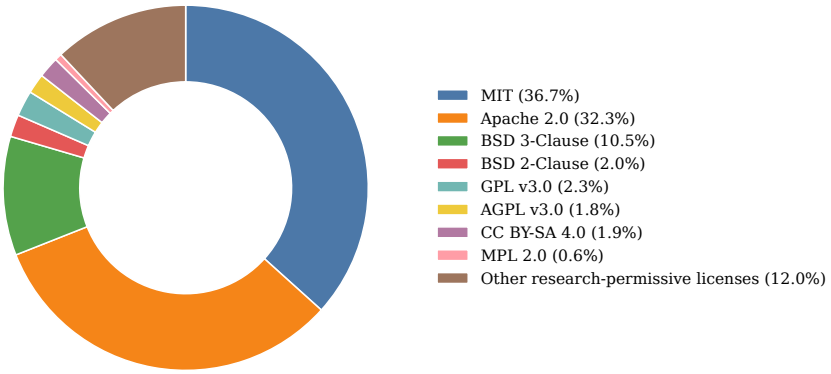

Figure 4: License distribution of the 968-repository corpus. Permissive licenses (MIT, Apache-2.0, BSD) account for over 80% of the corpus. Small categories (LGPL, ISC, Boost, MIT-0, Unlicense, CC0, CC BY, CC BY-NC, etc.) are aggregated as "Other research-permissive licenses," all of which permit at least non-commercial research use.

B TARGET-SELECTION DETAILS

This appendix gives the complete selection procedure summarized in Section 2.3: dependency-graph construction, single-function scoring and filtering, multi-function selection, and worked examples. The supplementary implementation includes both selection pipelines and all default hyperparameters listed here.

B.1 PROGRAM DEPENDENCY GRAPH: CALL RESOLUTION

For every `Call` node inside the body of $u \in \mathcal{V}$, we attempt to resolve its callee to some $v \in \mathcal{V}$, adding a directed edge $u \to v$ in $\mathcal{E}_{\text{call}}$. Resolution handles three Python idioms:

- **Direct module-level invocations**: `foo(...)` is resolved by qualified-name lookup in the same module.
- **Class instantiation**: `ClassName(...)` is resolved to `ClassName.__init__`.
- **Method calls**: `self.m(...)` or `cls.m(...)` inside class bodies is resolved within the enclosing class scope.

Calls unresolved by qualified-name lookup use short-name matching against the function definitions in the same file. Sibling edges $\mathcal{E}_{\text{sib}}$ are constructed by enumerating all pairs of methods belonging to the same class.

B.2 SINGLE-FUNCTION SCORING AND SELECTION

The selector balances implementation complexity with contextual inferability, applies a difficulty penalty, and filters candidates by length and score. We define each component before presenting the complete algorithm.

B.2.1 COMPLEXITY SCORE: CAPS AND WEIGHTS

For each function v, the complexity score is

$$\hat{H}(v) \;=\; w_\ell\,\phi(\mathrm{LoC}(v),\,c_\ell) + w_c\,\phi(\mathrm{CC}(v),\,c_c) + w_d\,\phi(\mathrm{D}(v),\,c_d)\,, \tag{2}$$

Here $\mathrm{LoC}(v)$ is the source-line count, $\mathrm{CC}(v)$ is McCabe cyclomatic complexity, and $\mathrm{D}(v)$ is the maximum nesting depth of control-flow constructs. The normalization $\phi(x, c) = \min(x/c, 2)$ scales

each quantity by a soft cap. We use caps $(c_\ell, c_c, c_d) = (50, 10, 5)$ and weights $(w_\ell, w_c, w_d) = (0.4, 0.4, 0.2)$. The cap $c_\ell = 50$ on lines of code lies above the mean target length of ≈ 34 lines (Table 5), so typical targets score below the cap; the cap $c_c = 10$ on McCabe complexity is roughly twice the median over our filtered functions; the cap $c_d = 5$ on nesting depth covers the tail of practical Python code. The cap value of 2 in $\phi(x, c) = \min(x/c, 2)$ allows genuinely complex functions to score above the median without letting outliers dominate.

B.2.2 Inferability Score: Component Formulas

The inferability score combines four source-code cues:

$$\hat{I}(v) \;=\; \alpha\, C_{\text{caller}}(v) + \beta\, C_{\text{callee}}(v) + \gamma\, C_{\text{sig}}(v) + \varepsilon\, C_{\text{class}}(v). \tag{3}$$

The components measure caller specificity, intra-file callees, signature information, and class context, respectively. Default mixing weights are $(\alpha, \beta, \gamma, \varepsilon) = (0.30, 0.25, 0.20, 0.15)$, applied directly without renormalization. Target docstrings are masked with the bodies and contribute no bonus to either single-function or group inferability. All numerical constants used inside the components are listed in Table 6; the formulas reference them directly.

Caller signal C_{caller}. For each in-file caller u of v we compute a per-caller specificity $\text{sp}(u, v)$ by scanning every call site inside u that targets v (matched by short name). At each matching site we accumulate $b_{\text{site}} = 0.50$ for the call itself, plus $\eta_{\text{lit}} = 0.15$ per literal-constant argument, $\eta_{\text{name}} = 0.05$ per name argument, $\eta_{\text{other}} = 0.08$ per other expression argument, $\eta_{\text{kw}} = 0.12$ per keyword argument, and a $b_{\text{found}} = 0.10$ "found" bonus. The per-caller specificity is clipped at $\text{cap}_{\text{caller}} = 1.5$, with a fallback value of 0.5 when the caller does not actually invoke v by name. Summing over callers and normalizing, $C_{\text{caller}} = \min(\sum_u \text{sp}(u, v) \,/\, n_{\text{caller}},\, 1)$, with normalization constant $n_{\text{caller}} = 3$.

Callee signal C_{callee}. $C_{\text{callee}} = \min(|\text{IntCallees}(v)|/n_{\text{callee}},\, 1)$ where $\text{IntCallees}(v)$ is the set of intra-file functions that v calls, and $n_{\text{callee}} = 4$. A function that orchestrates known helpers is more recoverable than one calling only opaque externals.

Signature signal C_{sig}. A monotone aggregate of four sub-bonuses, capped at 1: $+0.30$ if a return-type annotation is present, $+0.25$ if any parameter carries a type annotation, $+\min(\text{nameparts}/5,\, 0.25)$ for the descriptiveness of the function name (split on _), and $+\min(|\text{params}_{\neg\text{self}}|/6,\, 0.20)$ for the count of non-`self`/`cls` parameters.

Class signal C_{class}. For methods, $C_{\text{class}} = \min(\text{sib}(v)/n_{\text{sib}},\, 1)$ where $n_{\text{sib}} = 5$, plus a $b_{\text{init}} = 0.30$ bonus when a distinct `__init__` exists in the same class; the total is clipped at 1. Module-level functions receive $C_{\text{class}} = 0$.

Table 6: Numerical constants inside the $\hat{I}$ components.

Component	Constant	Value
C_{caller}	call-site base b_{site}	0.50
	literal-arg increment η_{lit}	0.15
	name-arg increment η_{name}	0.05
	other-arg increment η_{other}	0.08
	keyword-arg increment η_{kw}	0.12
	found bonus b_{found}	0.10
	per-caller cap $\text{cap}_{\text{caller}}$	1.5
	expected-callers norm n_{caller}	3
C_{callee}	expected-fan-out norm n_{callee}	4
C_{sig}	return-type bonus	$+0.30$
	param-type bonus	$+0.25$
	name-parts term	$\min(\text{parts}/5,\, 0.25)$
	non-self params term	$\min(\lvert p_{\neg\text{self}}\rvert/6,\, 0.20)$
C_{class}	sibling norm n_{sib}	5
	`__init__` bonus b_{init}	0.30

B.2.3 Combined Score, Difficulty Penalty, and Filters

We combine complexity and inferability as

$$\mathrm{FIM}(v) \;=\; \frac{\hat{H}(v)\,\hat{I}(v)}{\hat{H}(v)+\hat{I}(v)+\epsilon}\cdot\rho(\Delta(v))\,. \tag{4}$$

Here $\epsilon > 0$ provides numerical stability. The harmonic-mean-like term rewards targets with both high complexity and high inferability. The penalty ρ depends on residual difficulty $\Delta(v)$, the fraction of complexity exceeding inferability:

$$\Delta(v) \;=\; \frac{\max\Big(0,\,\hat{H}(v)-\hat{I}(v)\Big)}{\hat{H}(v)+\epsilon} \;\in\; [0,1). \tag{5}$$

The penalty ρ is one-sided: targets with $\Delta(v) \le \tau_d$ are left intact, while targets above the threshold are damped by a Gaussian factor:

$$\rho(\Delta) \;=\; \begin{cases} 1, & \Delta \le \tau_d, \\ \exp\!\left(-\dfrac{(\Delta-\tau_d)^2}{2\sigma^2}\right), & \Delta > \tau_d. \end{cases} \tag{6}$$

For single-function selection we use $\tau_d = 0.50$ and $\sigma = 0.20$. The hard filters below remove trivial targets. The one-sided penalty preserves scores when context supports the target's complexity and progressively down-weights targets with larger residual difficulty.

Hard filters. Independent of the smooth score, we discard *(i)* files outside the line-count window $[L_{\min}, L_{\max}] = [50, 1800]$ entirely; *(ii)* functions with $\mathrm{LoC}(v) \notin [10, 200]$ to control training-instance length; *(iii)* dunder methods (`__init__`, `__repr__`, etc.), which serve as scaffolding rather than reasoning targets; *(iv)* functions with $\hat{H}(v) < \tau_{\hat{H}} = 0.15$ to remove trivial cases; and *(v)* functions with $\mathrm{FIM}(v) < \tau_{\mathrm{FIM}} = 0.08$. A file with no surviving target contributes nothing to the corpus.

B.2.4 Selection Algorithm

Algorithm 1 applies the scores and filters above to each source file and returns the retained single-function targets.

We use $L_{\min} = 50$, $L_{\max} = 1800$, $\ell_{\min} = 10$, $\ell_{\max} = 200$, $\tau_{\hat{H}} = 0.15$, and $\tau_{\mathrm{FIM}} = 0.08$ throughout.

Algorithm 1 Single-function FIM target selection for one file.

Require: Source file s; thresholds $\tau_d, \tau_{\text{FIM}}, \tau_{\hat{H}}$; file-line bounds $[L_{\min}, L_{\max}]$; LoC bounds $[\ell_{\min}, \ell_{\max}]$

if $\text{Lines}(s) \notin [L_{\min}, L_{\max}]$ **then**
  return $\emptyset$ ▷ file-level pre-filter
end if
$T \leftarrow \text{ParseAST}(s)$
$\mathcal{V}, \mathcal{E}_{\text{call}}, \mathcal{E}_{\text{sib}} \leftarrow \text{BuildPDG}(T)$ ▷ Section B.1
$\mathcal{C} \leftarrow \emptyset$
for each $v \in \mathcal{V}$ **do**
  if v is a dunder method **or** $\text{LoC}(v) \notin [\ell_{\min}, \ell_{\max}]$ **then**
    continue
  end if
  Compute $\hat{H}(v)$ and $\hat{I}(v)$ ▷ Eqs. (2), (3)
  if $\hat{H}(v) < \tau_{\hat{H}}$ **then continue**
  end if
  $\Delta(v) \leftarrow \max(0, \hat{H}(v) - \hat{I}(v))/(\hat{H}(v) + \epsilon)$
  $\text{FIM}(v) \leftarrow \dfrac{\hat{H}(v)\,\hat{I}(v)}{\hat{H}(v) + \hat{I}(v) + \epsilon} \cdot \rho(\Delta(v))$
  if $\text{FIM}(v) \geq \tau_{\text{FIM}}$ **then**
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{v\}$
  end if
end for
return $\mathcal{C}$ ▷ ranked by FIM descending

B.3 Multi-Function Group Score

Group score. For a group $G = \{v_1, \ldots, v_k\}$ enumerated from one of eight topology patterns,

$$\text{FIM}(G) \;=\; \text{Coup}(G) \cdot \frac{\hat{H}(G)\,\hat{I}(G)}{\hat{H}(G) + \hat{I}(G) + \epsilon} \cdot \rho^{G}(\Delta(G))\,, \tag{7}$$

where $\hat{H}(G) = \frac{1}{k}\sum_v \hat{H}(v)$ averages individual complexities, $\Delta(G)$ is defined analogously to the single-function case, ρ^G uses group-specific parameters $\tau_d^G = 0.55$ and $\sigma^G = 0.20$ (slightly more permissive than the single-function $\tau_d = 0.50$, accounting for context removed by joint masking), and the coupling term $\text{Coup}(G) \in [0, 1]$ is a normalized count of intra-group edges plus a shared-state ratio:

$$\text{Coup}(G) \;=\; w_{\text{c}}\,\frac{|E_G^{\text{call}}|}{k(k-1)} + w_{\text{s}}\,\frac{|E_G^{\text{sib}}|}{\binom{k}{2}} + w_{\text{st}}\,\overline{\text{Jacc}}_G, \tag{8}$$

with $\overline{\text{Jacc}}_G$ the mean Jaccard similarity over pairs in G between sets of accessed instance attributes (`self.x`). Pairs with two empty attribute sets contribute zero shared-state similarity. We use $(w_{\text{c}}, w_{\text{s}}, w_{\text{st}}) = (0.50, 0.20, 0.30)$. Multiplying by the coupling term favors groups with strong dependencies among their members, making cross-function reasoning part of the reconstruction task.

Group inferability. Let $\hat{I}(v; G)$ denote the score in Eq. (3) recomputed for member v with *all* bodies in G masked. Group inferability averages these member scores:

$$\hat{I}(G) = \frac{1}{|G|}\sum_{v \in G} \hat{I}(v; G). \tag{9}$$

In each $\hat{I}(v; G)$, caller, callee, and sibling contributions from inside G are excluded, and the `__init__` bonus in C_{class} is awarded only when `__init__` is itself outside G. Visible function signatures supply the signature component. The remaining constants $(b_{\text{site}}, \eta_{\text{lit}}, \ldots)$ in Table 6 are unchanged. Thus, surviving context determines inferability, while intra-group dependencies contribute to the coupling score.

Topology taxonomy. We generate candidate groups using eight topology patterns. Retained groups must additionally form a connected subgraph in $\mathcal{E}_{\text{call}} \cup \mathcal{E}_{\text{sib}}$ (ignoring call-edge directions for connectivity) and satisfy the coupling and score thresholds below. The candidate patterns are:

- $k = 2$: *caller-callee* (direct $A \to B$ edge); *co-callee* (two callees of a common in-file caller); *sibling-coupled* (same-class methods sharing ≥ 1 instance attribute); *mutual-call* ($A \rightleftharpoons B$).
- $k = 3$: *call-chain* ($A \to B \to C$); *hub* ($A \to B,\ A \to C$); *fan-in* ($B \to A,\ C \to A$); *class-triad* (three pairwise state-sharing methods of the same class).

Sharing a caller alone does not establish an intra-group edge or a positive coupling score. A co-callee pair is retained only if its members also have a call or sibling relation and pass the same thresholds as other candidates.

Group selection. We filter candidate groups for internal connectivity, then score them and discard those failing any of the following defaults: each member must satisfy $\text{LoC} \in [10, 200]$ and not be a dunder; $\text{Coup}(G) \geq \tau^G_{\text{coup}} = 0.15$; $\hat{H}(G) \geq \tau^G_{\hat{H}} = 0.15$; total LoC ratio of the group relative to the file does not exceed $\theta_2 = 0.30$ for $k = 2$ or $\theta_3 = 0.40$ for $k = 3$; and the score clears a size-specific floor, $\text{FIM}(G) \geq \tau^G_{\text{FIM},2} = 0.04$ for pairs and $\text{FIM}(G) \geq \tau^G_{\text{FIM},3} = 0.03$ for triples. Remaining candidates are sorted by $\text{FIM}(G)$ and selected greedily under non-overlap: a function may belong to at most one selected group per file. Per-file caps $(N_2, N_3) = (5, 3)$ prevent any single file from dominating the corpus. Algorithm 2 summarizes the procedure.

Algorithm 2 Multi-function group FIM target selection for one file.

Require: Source file s; per-function scores $\hat{H}(v), \hat{I}(v)$; thresholds $\tau^G_{\text{coup}} = 0.15$, $\tau^G_{\hat{H}} = 0.15$, $\tau^G_{\text{FIM},2} = 0.04$, $\tau^G_{\text{FIM},3} = 0.03$; LoC ratio caps $(\theta_2, \theta_3) = (0.30, 0.40)$; per-file caps $(N_2, N_3) = (5, 3)$

$\mathcal{V}, \mathcal{E}_{\text{call}}, \mathcal{E}_{\text{sib}} \leftarrow \text{BuildPDG}(\text{ParseAST}(s))$
$\mathcal{G} \leftarrow \emptyset$
for each topology pattern $P \in$ {caller-callee, co-callee, sibling-coupled, mutual-call, call-chain, hub, fan-in, class-triad} **do**
  Enumerate candidate groups $G \subseteq \mathcal{V}$ matching P
  for each candidate G **do**
    if G is not internally connected in $\mathcal{E}_{\text{call}} \cup \mathcal{E}_{\text{sib}}$ **then**
      continue ▷ ignore call-edge directions
    end if
    if any $v \in G$ fails the per-function filter **or** $\text{LoC_ratio}(G, s) > \theta_{|G|}$ **or** $\text{Coup}(G) < \tau^G_{\text{coup}}$ **or** $\hat{H}(G) < \tau^G_{\hat{H}}$ **then**
      continue
    end if
    $\hat{I}(G) \leftarrow |G|^{-1} \sum_{v \in G} \hat{I}(v; G)$ under joint masking ▷ exclude intra-group references
    $\Delta(G) \leftarrow \max(0, \hat{H}(G) - \hat{I}(G))/(\hat{H}(G) + \epsilon)$
    $\text{FIM}(G) \leftarrow \text{Coup}(G) \cdot \dfrac{\hat{H}(G)\,\hat{I}(G)}{\hat{H}(G) + \hat{I}(G) + \epsilon} \cdot \rho^G(\Delta(G))$
    if $\text{FIM}(G) \geq \tau^G_{\text{FIM},|G|}$ **then**
      $\mathcal{G} \leftarrow \mathcal{G} \cup \{(G, P, \text{FIM}(G))\}$
    end if
  end for
end for
Sort $\mathcal{G}$ by $\text{FIM}(G)$ descending
$\mathcal{S} \leftarrow \emptyset$
for each $(G, P, \text{FIM}(G)) \in \mathcal{G}$ **do**
  if any $v \in G$ already covered by $\mathcal{S}$ **or** per-size cap (N_2 for $k = 2$, N_3 for $k = 3$) exceeded **then**
    continue
  end if
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{G\}$
end for
return $\mathcal{S}$

B.4 Worked Examples

B.4.1 Worked Example: Scoring `Calculator.total`

To make the numbers in Figure 2(b) reproducible, this subsection walks through every quantity (panel (a) of the figure shows an abbreviated listing; all quantities below use the listing given here) that goes into $\hat{H}$, $\hat{I}$, and the final FIM score for the running example. The target function is:

```
def total(self) -> int:
    """Sum positive integer entries."""
    s, n = 0, 0
    for v in self.history:
        if is_int(v):
            if v > 0:
                s = add(s, v)
                n += 1
    if n == 0: return 0
    return s
```

Lines of code (LoC = 10). Following Python's `ast` convention, $\mathrm{LoC}(v) = \texttt{end_lineno} - (\texttt{lineno} - 1)$, the number of source lines from the `def` keyword through the last statement, both inclusive. Counting the ten lines above (including the docstring) gives $\mathrm{LoC} = 10$.

Cyclomatic complexity (CC = 5). The McCabe counter starts at 1 and adds 1 per branching node: `If`/`IfExp`, `For`/`While`, `ExceptHandler`, (values-1) per `BoolOp`, and one per generator inside a comprehension. For `Calculator.total`: base ($+1$), the `for` loop ($+1$), `if is_int` ($+1$), `if v>0` ($+1$), `if n==0` ($+1$), totalling $\mathrm{CC} = 5$. Tuple unpacking and `n+=1` are not branches.

Maximum nesting depth (D = 3). Depth increments only on structural containers (`If`, `For`, `While`, `AsyncFor`, `With`, `AsyncWith`, `Try`). The deepest path in `total` is `for` $\rightarrow$ `if is_int` $\rightarrow$ `if v>0`, giving $\mathrm{D} = 3$.

Complexity score $\hat{H} = 0.40$. Plugging into Eq. 2 with the default $(w_\ell, w_c, w_d) = (0.4, 0.4, 0.2)$ and $(c_\ell, c_c, c_d) = (50, 10, 5)$:

$$w_\ell\,\phi(10,50) = 0.4{\cdot}0.20 = 0.08, \qquad w_c\,\phi(5,10) = 0.4{\cdot}0.50 = 0.20,$$
$$w_d\,\phi(3,5) = 0.2{\cdot}0.60 = 0.12, \quad \hat{H}(\texttt{total}) = 0.08 + 0.20 + 0.12 = 0.40.$$

Inferability score $\hat{I} = 0.42$. *(i) Caller specificity.* The only in-file caller is `Calculator.mean`, which invokes `self.total()` with no arguments and no keywords. Per Table 6, $\mathrm{sp} = b_{\text{site}} + b_{\text{found}} = 0.50 + 0.10 = 0.60$. Aggregating and normalizing by $n_{\text{caller}} = 3$, $C_{\text{caller}} = \min(0.60/3, 1.0) = 0.20$ and $\alpha\, C_{\text{caller}} = 0.06$. *(ii) Callee count.* `total` calls `is_int` and `add` in-file, so $C_{\text{callee}} = \min(2/n_{\text{callee}}, 1.0) = \min(2/4, 1.0) = 0.50$ and $\beta\, C_{\text{callee}} = 0.125$. *(iii) Signature.* Sub-bonuses $+0.30$ for `-> int`; $+\min(1/5, 0.25) = 0.20$ for the descriptive name `total`; 0 for parameter type annotations; 0 for non-`self` parameters: $C_{\text{sig}} = 0.50$ and $\gamma\, C_{\text{sig}} = 0.10$. *(iv) Class context.* `total` sits among three siblings; the base score is $\min(3/n_{\text{sib}}, 1.0) = \min(3/5, 1.0) = 0.60$, plus the $b_{\text{init}} = 0.30$ bonus for the distinct `__init__` of `Calculator`, giving $C_{\text{class}} = 0.90$ and $\varepsilon\, C_{\text{class}} = 0.135$. Summing the unrounded contributions: $\hat{I}(\texttt{total}) = 0.06 + 0.125 + 0.10 + 0.135 = 0.42$.

Final FIM score and selection. $\Delta(\texttt{total}) = \max(0, -0.02)/0.40 = 0$, so $\rho(\Delta) = 1$ and $\mathrm{FIM}(\texttt{total}) = (0.40{\cdot}0.42)/(0.40{+}0.42) \approx 0.2049 \geq \tau_{\text{FIM}} = 0.08$, selecting it as a mask target. The four short helpers are removed before scoring by the LoC hard filter; `Calculator.__init__` is removed by the dunder filter.

B.4.2 Joint Masking of a Caller–Callee Pair

The following code excerpt illustrates how joint masking changes the evidence available for a caller–callee pair:

```
def normalize(text: str) -> str:
    """Lowercase and strip punctuation."""
    out = []
    for ch in text:
        if ch.isalnum() or ch.isspace():
            out.append(ch.lower())
    return "".join(out)

def word_freq(doc: str) -> dict[str, int]:
    """Count tokens of `doc` after normalization."""
    cleaned = normalize(doc)
    counts = {}
    for tok in cleaned.split():
        counts[tok] = counts.get(tok, 0) + 1
    return counts
```

The pair consists of `normalize` and `word_freq`, with one call edge from the latter to the former. As top-level functions, they have no sibling or shared-instance-state contribution, so Eq. (8) gives $\mathrm{Coup}(G) = 0.50 \cdot (1/2) = 0.25$. When only `normalize` is masked, its caller provides evidence about how its output is used. When both bodies are masked, that call site disappears from the visible context and is excluded from the group's caller and callee signals. Their signatures remain visible, while their docstrings are masked with their bodies.

The example illustrates why group inferability is recomputed after joint masking: dependencies within the group contribute to its coupling score, while inferability reflects the surviving context. Candidate groups are then evaluated in their full source files using the member-length, file-ratio, and score thresholds above.

C Sample Construction and Training

This appendix follows the training-data pipeline from rationale and implementation generation through quality filtering and chat serialization, then lists the training settings and compute resources. The prompt templates illustrate the single-function case.

C.1 Chain-of-Thought Generation Prompt Template

For the default recipe, each FIM rationale–implementation pair is produced by Gemini-3-Flash in a single forward pass: given a Python source file with one function body redacted, the model is asked to first reason about the function's behavior and then write a candidate implementation. The masked file supplies the generation context; the original body is used in the subsequent quality-filtering step. Retained rationale–implementation pairs become the assistant targets.

> *[System]* You are an expert Python programmer.
>
> *[User]* Below is a Python file where one function's body has been replaced with `# <MASKED_FUNCTION_BODY>`. Complete the masked function from the surrounding context.
>
> *Procedure.* Analyze the visible context—imports, sibling functions, classes, call sites, and helpers invoked by the target—then reason step by step about what the function must do given its signature, type hints, and usage pattern. Only after this analysis, write an implementation consistent with the file's coding style.
>
> *Output format.* Return two sections in order: `### Reasoning` (a step-by-step natural-language analysis) and `### Implementation` (a fenced `python` block containing the function body only, without the signature).
>
> `<file with target body redacted>`
>
> The function to complete: `<function name>`.

The production prompt organizes generation into context analysis, step-by-step reasoning, and implementation. Its reasoning checklist covers signature semantics, intra-file usage, callee dependencies, and the function's overall purpose. Formatting instructions specify body-only code, four-space indentation, docstring handling, and the Markdown headers and code fences used to separate rationale from implementation. We use the Gemini API's near-greedy configuration (`temperature` $= 0$, `topK` $= 1$). The full production prompt and post-processing logic will be released with the corpus.

C.2 COMPLETION-QUALITY FILTERING PROMPT TEMPLATE

Each candidate FIM sample, paired with a trial completion, is screened by an LLM judge that assesses both whether the masked function is recoverable from its surrounding context and the quality of the completion against the ground truth. Filtering is performed by Gemini-3-Flash using the following prompt skeleton:

> *[System]* You are an expert code reviewer.
>
> *[User]* A model was asked to complete a masked function from its surrounding code context. Evaluate the result along two axes: (i) whether the function is feasible to complete from context alone, and (ii) the quality of the completion compared with the ground truth.
>
> The source file with the target body redacted (shown as `<MASKED>`):
>
> `<file with target body redacted>`
>
> The function name, the ground-truth body, and the model's completion:
>
> `<function name>`, `<ground-truth body>`, `<model completion>`
>
> *Part 1 – Feasibility.* Mark the function *infeasible* if it depends on uncommon external APIs, external conventions, or magic constants that cannot be inferred from the visible context; otherwise mark it *feasible*.
>
> *Part 2 – Quality.* Score the completion on a 1–5 scale along five dimensions—*correctness*, *executability*, *API usage*, *readability*, and *completeness*—each with a one-sentence justification, then assign an overall 1–5 score.
>
> *Output.* Return a JSON object containing the feasibility verdict, the five component scores with reasons, the overall score, and a `should_discard` flag set to `true` when the function is infeasible, when `executability` $= 1$, or when both `overall_score` ≤ 2 and `executability` ≤ 2.

The production rubric defines each score level on all five dimensions. For example, correctness ranges from 1 (completely incorrect), through 3 (partially correct with important errors), to 5 (functionally identical). The feasibility checklist covers external APIs and conventions, available context, and system-specific constants. The JSON schema records component reasons, key similarities and differences, and a free-text `discard_reason`. We use the Gemini API's near-greedy configuration (`temperature` $= 0$, `topK` $= 1$). The full production prompt and post-processing logic will be released with the corpus.

C.3 MID-TRAINING SAMPLE FORMATTING TEMPLATE

After generation and quality filtering, each retained sample in the default CoT-augmented recipe is serialized into a chat-style turn pair: the user turn poses the FIM completion task on the file with one function body redacted, and the assistant turn carries Gemini's rationale followed by Gemini's predicted body, both produced in the upstream pass. The ground-truth body from the source repository provides the reference for quality filtering. We use the model's native chat template and apply autoregressive loss to the assistant response. The FIM notation below schematically expresses the conditioning structure for a single masked body:

`<fim_prefix>` ⟨prefix⟩ `<fim_suffix>` ⟨suffix⟩ `<fim_middle>` ⟨rationale⟩ ⟨body⟩

The corresponding chat-formatted sample is:

> *[System]* You are an expert Python programmer.
>
> *[User]* Below is a Python file where one function's body has been replaced with `# <MASKED_FUNCTION_BODY>`. Complete the masked function from the surrounding context. *[Procedure and output-format instructions identical to Section C.1.]*
>
> `<file with target body redacted>`
>
> The function to complete: `<function name>`.
>
> *[Assistant]*
>
> ```
> ### Reasoning
> <Gemini-produced rationale>
> ### Implementation
> ```python
> <Gemini-produced implementation>
> ```
> ```

In the *no CoT* ablation (Section 3.4), we remove the rationale from each retained Gemini rationale–implementation pair. The input and Gemini-generated implementation are identical to those in the Gemini-CoT variant. In the *self-CoT* variant, the assistant target contains a rationale followed by a function implementation, both generated by the model under training.

C.4 Training Hyperparameters

We use LlamaFactory for FIM mid-training, R2E-Gym post-training, and SWE-Lego post-training, and torchtune for SWE-Smith post-training. All runs use AdamW with bf16 mixed precision and a cosine learning-rate schedule. For R2E-Gym and SWE-Smith we follow the official release scripts; for SWE-Lego we use 2 epochs instead of the official 4 to prevent overfitting (all other hyperparameters unchanged). Tables 7 and 8 list the exact settings; effective (global) batch sizes assume 8 GPUs.

Table 7: FIM mid-training hyperparameters, applied uniformly to all three base models (Qwen2.5-Coder-7B-Instruct, Qwen2.5-Coder-14B-Instruct, and Qwen3-8B).

Hyperparameter	**Value**
Optimizer	AdamW
Learning rate	1.0×10^{-5}
LR schedule	Cosine
Warmup ratio	0.1
Weight decay	0.05
Epochs	1
Per-device batch size	1
Gradient accumulation	16
Effective batch size	128
Sequence length	32,768
Precision	bf16

Table 8: Agentic post-training hyperparameters for the three pipelines, applied identically to the reproduced baselines and the FIM-midtrained checkpoints. R2E-Gym and SWE-Smith follow their official released scripts; SWE-Lego follows the official recipe except for the epoch count.

Hyperparameter	**R2E-Gym**	**SWE-Smith**	**SWE-Lego**
Base model	Qwen2.5-Coder-7B/14B-Inst.	Qwen2.5-Coder-7B-Inst.	Qwen3-8B
Optimizer	AdamW	AdamW (fused)	AdamW
Learning rate	1.0×10^{-5}	1.0×10^{-4}	1.0×10^{-4}
LR schedule	Cosine	Cosine	Cosine
Warmup	ratio 0.05	5 steps	ratio 0.1
Weight decay	0.0	0.01	0.01
Epochs	2	3	2[†]
Per-device batch size	1	1	1
Gradient accumulation	1	4	8
Effective batch size	8	32	64
Sequence length	32,768	32,768	40,960
Precision	bf16	bf16	bf16

[†]SWE-Lego uses 2 epochs in our setup (the official recipe uses 4) to prevent overfitting in our setup; the setting is shared by the reproduced baseline and the mid-trained run.

C.5 Compute Resources

All experiments are run on a single node of 8 NVIDIA H100 80 GB GPUs. Reproducing the full set of results in this paper—FIM mid-training of three base models (Qwen2.5-Coder-7B/14B-Instruct and Qwen3-8B), the three agentic post-training pipelines (R2E-Gym, SWE-Smith, SWE-Lego) and their corresponding mid-trained variants, the ablations and comparisons of Section 3.4, and the three-run SWE-Bench evaluations behind Tables 1 and 3—takes roughly 30 days of wall-clock time on the

8-GPU node, i.e. $\approx 5{,}760$ GPU-hours in total. Preliminary and discarded runs not reported in this paper account for an additional $\sim 30\%$ of compute.

D BEHAVIORAL ANALYSIS

This appendix supplements Section 4 with metric definitions, full trajectory statistics, outcome breakdowns, and an example repair. We compare the 14B R2E-Gym baseline with FIM-Midtrain + R2E-Gym on Verified and Lite. Statistics are means over three independent evaluation runs of each checkpoint on each split.

D.1 NEGATIVE OBSERVATIONS AND RECOVERY

The recovery-rate analysis (Section 4.1) flags a tool output as a *negative observation* when it matches any of the following patterns: Python stack-trace prefixes (`Traceback (most recent call last):`); error class names (`SyntaxError`, `IndentationError`, `NameError`, `ImportError`, `AttributeError`, `TypeError`, `ValueError`); shell error markers (`command not found`, `permission denied`, `No such file or directory`); and harness-specific failure strings (`No replacement was performed`, `Patch did not apply`, `tests failed`). Patterns are matched case-insensitively.

A trajectory contains a negative observation if at least one tool output matches these patterns. The recovery rate is the fraction of such trajectories that end with a passing patch.

D.2 SWE-BENCH-VERIFIED

The Verified analysis uses all 500 instances per evaluation run.

D.2.1 FULL TRAJECTORY METRICS

Table 9 reports the full trajectory metrics summarized in Table 4, including unsolved-task breakdowns and output/context summaries.

Table 9: Trajectory-level behavioral metrics on SWE-Bench-Verified (14B, R2E-Gym), run-means over three evaluation runs of each checkpoint. Steps, edits, and errors-encountered are means per task. "Empty patch" is the fraction of *failed* trajectories whose final `output_patch` is empty, including trajectories with unsuccessful edit attempts. "Ctx @ success" is the mean prompt context length on solved trajectories.

	Steps / task		Edits / task		Errors enc.		Recovery
Setting	solved	unsolved	solved	unsolved	solved	unsolved	rate (%)
+ R2E-Gym	15.1	22.4	3.3	5.4	3.5	5.8	24.8
+ FIM-Midtrain + R2E-Gym	**23.6**	**32.7**	**7.4**	**10.9**	**6.2**	**8.1**	**28.8**
Output and context summaries							
	Empty patch (% failed)		Ctx @ success (tok)				Pass (%)
+ R2E-Gym	3.0		11,826				26.2
+ FIM-Midtrain + R2E-Gym	**0.3**		**16,397**				**29.2**

D.2.2 ITERATE-AND-VERIFY: ACTION-TYPE DISTRIBUTION

Mid-training shifts the action-type distribution toward an iterate-and-verify policy. The share of `search` actions falls from 15.3% to 11.0%, while the share of `execute_bash` actions rises from 19.5% to 24.6%: the agent spends a smaller fraction of its budget on passive repository lookup and a larger fraction on actively running scripts and tests, then refining its edit. The fraction of trajectories reaching the step cap is $\sim 45\%$ for the mid-trained agent and 7% for the baseline. On solved tasks, edits increase from 3.3 to 7.4 per trajectory, alongside an overall pass-rate increase from 26.2% to 29.2%.

D.2.3 Pass Rate by Gold-Patch Shape

The pass-rate stratification used in Section 4.2 is as follows: the largest gain (+9.1 pp) is on multi-function single-file tasks (n=88), more than 4× the +2.1 pp gain on single-function tasks (n=341); both checkpoints solve ∼11.3% of the multi-file tasks (n=71).

On the 71 Verified tasks whose gold patch spans ≥ 2 files, both checkpoints solve ∼11.3% on average and per-instance head-to-head on this bucket is balanced. Our function-aware FIM mid-training selects functions and function groups within individual files. Extending the selector to cross-file function pairs would expose dependencies that span multiple files.

D.2.4 Repair Outcomes and Trajectory Examples

We label every failed trajectory using a deterministic classification of its final output: *no-patch* (output diff is empty); *localization error* (non-empty patch but zero file overlap with the gold patch); *patch error* (file overlap with gold patch but tests fail). Figure 5 reports the outcome distribution: across the three runs, mid-training cuts no-patch failures by an order of magnitude (∼11 → ∼1 trajectories per run), trims localization errors slightly (∼131 → ∼126), and leaves the patch-error count essentially flat (∼227 → ∼227). The number of solved tasks increases from ∼131 to ∼146 per run.

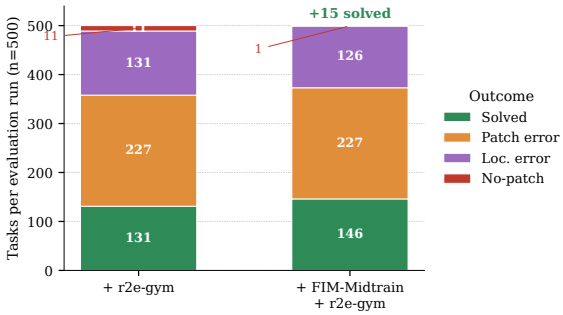

Figure 5: Outcome distribution per evaluation run on SWE-Bench-Verified (14B, R2E-Gym), averaged over three runs.

Empty-patch outcomes. *No-patch* denotes an unsuccessful repair trajectory ending without effective code changes. This category includes failed attempts to apply or revise a patch: the agent encounters an editing error, fails to complete the repair, and ends with an empty diff.

After mid-training, no-patch outcomes fall from ∼ 11 to ∼ 1 per run. Recovery from negative observations improves from 24.8% to 28.8%, while overall task success rises from 26.2% to 29.2%. These results characterize an improvement in carrying repair attempts through to passing patches.

Example trajectory. On `scikit-learn-26323`, the baseline trajectory runs for a single step in every run, immediately emits `<finish>` with an empty edit, and is scored as a failure; on the same task our agent runs ∼41 steps, applies on the order of 20 `str_replace` operations, observes multiple negative tool returns, and recovers to a passing patch. This trajectory illustrates repair completion through continued editing and response to tool feedback on a task that the baseline leaves unresolved.

D.3 SWE-Bench-Lite

We replicate the Verified analysis on the SWE-Bench-Lite test split (300 instances). The same trajectory-parsing pipeline and failure-mode taxonomy are used; statistics are run-means over three independent evaluation runs of each checkpoint. Table 10 and Figure 6 report the metrics and the patch-shape breakdown.

Recovery and edit-iteration trends also hold on Lite: mid-training raises the recovery rate from 18.8% to 22.1%, reduces no-patch outcomes (baseline averages ∼8 no-patch failures per run, ours averages

0), and increases edit iteration on solved trajectories from 4.4 to 7.8 `str_replace` operations per task. On Lite, the gain concentrates on single-function single-file tasks, with a +4.9 pp improvement (19.5% → 24.4%). The split contains 246 such tasks and 54 multi-function single-file tasks, with balanced performance in the latter bucket.

Table 10: Trajectory-level behavioral metrics on SWE-Bench-Lite (14B, R2E-Gym), run-means over three evaluation runs of each checkpoint. Definitions match Table 9.

	Steps / task		Edits / task		Errors enc.		Recovery
Setting	solved	unsolved	solved	unsolved	solved	unsolved	rate (%)
+ R2E-Gym	17.5	21.3	4.4	5.2	4.8	5.4	18.8
+ FIM-Midtrain + R2E-Gym	**24.6**	**31.7**	**7.8**	**11.1**	**5.7**	**7.5**	**22.1**
Output and context summaries							
	Empty patch (% failed)		Ctx @ success (tok)				Pass (%)
+ R2E-Gym	3.3		13,958				18.0
+ FIM-Midtrain + R2E-Gym	**0.0**		**18,104**				**22.0**

Final-outcome counts on Lite move in the same direction as on Verified: no-patch outcomes fall from ∼8 to 0 trajectories per run, localization errors fall by about 11, and patch errors rise by about 7. Across the same runs, the solve rate increases from 18.0% to 22.0%.

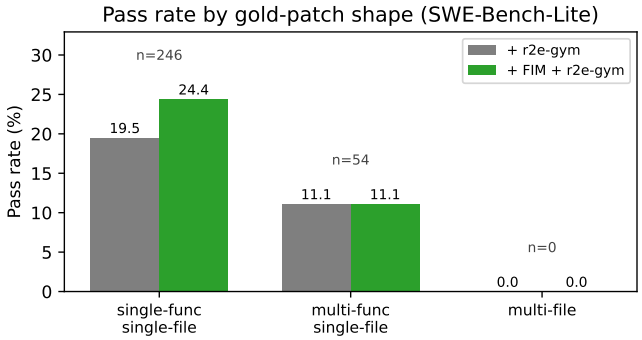

Figure 6: Pass rate on SWE-Bench-Lite stratified by gold-patch shape, averaged over three evaluation runs per checkpoint. The +4.0 pp end-task improvement comes from the single-function single-file bucket (n=246, +4.9 pp); performance is balanced in the multi-function single-file bucket (n=54). Lite contains no multi-file tasks.