DISSECTING MODEL BEHAVIOR THROUGH AGENT TRAJECTORIES

Anonymous authors
Paper under double-blind review

ABSTRACT

AI agent performance is not just a modeling problem, it is fundamentally a systems problem. The advanced capabilities of models are realized through agent harnesses. Therefore, a gap between model assumptions and harness behavior can easily prevent the model's full capabilities from translating into agent performance. We formalize this as the 'intent-execution' gap: the mismatch between what the model intends and what the harness executes, and vice versa. We argue that minimizing this intent-execution gap is as important as other aspects of harness design such as tools and execution loops. To illustrate the impact of this harness-model alignment, we develop a simple and customizable harness called 'Simple Strands Agent' (SSA). SSA aims to find the bulk of common patterns which generalize across different model families (such as Claude, Gemini, GPT, Grok, Qwen), as well as a small number of model-specific preferences. We make two contributions: (i) we **reproduce or improve on the pass@1** performance reported by diverse model-provider families on popular agentic benchmarks (SWE-Pro, SWE-Verified, Terminal-Bench-2, and Deep-SWE), and (ii) building on an **analysis of 154k trajectories generated by SSA**, we look beyond the `pass@1` numbers which tend to be relatively even across frontier models. By representing agent trajectories in code state-spaces, we compute solution distance and backtracking to observe model-level differences in problem-solving behavior. Finer-grained metrics such as edit frequency, testing activity, and phase-transitions reveal how individual models allocate effort across different stages of problem solving.

1 INTRODUCTION

A modern coding agent couples an LLM with a *harness*, the software layer that mediates every tool call, feeds execution outcomes back into the context window, and manages the thought-action-observation loop (Wang et al., 2024a; Xi et al., 2023; Schluntz & Zhang, 2024; Yang et al., 2024a).

As base models improve, the alignment of the model with the harness becomes critical to realizing the model's full capabilities. Informally, given two models $\mathcal{M}_A > \mathcal{M}_B$ in raw capability, a harness H without such alignment can yield $\mathrm{acc}(H, \mathcal{M}_A) \approx \mathrm{acc}(H, \mathcal{M}_B)$ where acc is accuracy.

We formalize this misalignment between harness and model as the **intent-execution gap**: a bidirectional mismatch between what the model intends and what the harness executes, and between what happened in the environment and what is revealed to the model . The gap is bidirectional because failures compound in both directions as a misinterpreted edit silently corrupts state, and an under-specified execution result leads the model to reason from false premises. For instance, a model may intend to replace a single occurrence of a function call, while the harness silently patches every match; or a long shell output may be truncated in a way that hides the test failure the model needs to see.

Problem-specific tunings such as bespoke prompts (Brown et al., 2020; Kojima et al., 2023), specialized tool schemas (Schick et al., 2023; Qin et al., 2023), or hand-crafted execution graphs (Yao et al., 2023; Shinn et al., 2023; Wu et al., 2023) can reduce this gap for a given model on a given benchmark. But such gains are brittle. As models improve (Wei et al., 2022; Guo et al., 2025), behaviors shift and optimizations that overfit one model's quirks can degrade on the next (Schluntz & Zhang, 2024; Wang et al., 2024a). This motivates a different strategy: rather than only chasing model-specific tricks, identify *invariant* interface principles that remain effective across model families, model

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.

generations, and benchmarks, and adapt only where families genuinely differ. This perspective raises two fundamental questions at the model-harness boundary:

1. Does the harness faithfully execute what the model intended?
2. Is the model given an accurate account of what the harness did?

We show that minimizing this bidirectional gap without task-specific tuning is sufficient to match or exceed official benchmark numbers for models we evaluate across seven families (Claude, GPT, Gemini, Grok, Kimi, Muse Spark, Qwen) on SWE-Bench-Verified, SWE-Bench-Pro, Terminal-Bench-2 and Deep-SWE.

We also analyze SSA trajectories to answer a question that pass@1 cannot: *how* do different model families solve the same problems? By projecting agent state into a text-level code space and measuring distance to verified solutions at each cycle, we surface stable behavioral signatures (Section 3) in the form of tool-call intent, phase (exploration vs implementation), backtracking rates that differentiate families even when their pass@1 scores are indistinguishable.

Main Contributions.

1. We introduce Simple Strands Agent (SSA), an open-source harness designed around the intent-execution gap, and show that it reproduces or exceeds the published benchmark numbers for 61 of 66 model–benchmark pairs across seven model families on SWE-Bench-Verified, SWE-Bench-Pro, Terminal-Bench-2 and Deep-SWE.
2. We define a solution-distance metric over a text-level code space that tracks, at each agent cycle, how far the repository state is from a verified-correct fix, and decompose it exactly into best-so-far progress and a backtracking term, giving backtracking a precise and time-dependent definition.
3. We present a trajectory-level analysis across all 38 evaluated models revealing family-specific behavioral signatures such as edit/test ratios, phase schedules, and backtracking patterns, that are stable across benchmarks and invisible to aggregate metrics.
4. We identify and quantify a git-history leakage channel in the public SWE-Bench-Pro containers that inflates measured pass@1 by up to 6.9% for the affected models (Appendix E).

Section 2 explains harness improvements that make SSA effective, Section 3 introduces a framework for tracking agent code-states as they evolve over the course of a trajectory, and Section 4 reports SSA's performance on agentic benchmarks and the analysis of model trajectories. Appendix A outlines the SSA architecture.

2 SSA Refinements

We now describe methods that make SSA effective, namely, feedback-rich tool boundaries in Section 2.1, our approach to prompting agents for a better balance between reasoning and tool-use in Section 2.2, as well as leaning on the model's preferences for tool interfaces in Section 2.3.

2.1 Feedback-rich tool boundaries

The key design principle of the SSA harness is to make the information exchanged at the model-tools boundary transparent and explicit. We do so through the following mechanisms:

Structured execution feedback. All shell calls return the following information: command, status, exit code, output, and error text. When interrupted by a timeout, partial output is returned when available. Long outputs are clipped in the middle but preserve the start and end of the outputs so that failure summaries and final test results are not lost.

Safer editing semantics. All edits are anchored to neighboring lines. Partial-line matches and duplicate matches are rejected with feedback that helps the model disambiguate and issue a more precise edit in the next turn. Successful edits return a unified diff reducing the need for the model to verify if the intended change was actually applied.

2.2 BALANCING REASONING WITH TOOL-CALLING

Chain-of-thought reasoning (Wei et al., 2023; Kojima et al., 2023) is clearly valuable. It allows the model to decompose a problem, plan next-steps, and decide which tool to invoke. Without sufficient reasoning, tool-usage becomes reactive, leading to shallow exploration, redundant calls, or poor sequencing of actions. However, excessive thinking introduces its own failure mode. When the model spends too long reasoning internally, it begins to form assumptions about the environment rather than verifying them. As a result, the agent may issue poorly grounded tool calls or skip necessary validation steps altogether, creating a fundamental tension between the model's internal state and the external environment.

Effective agents must continuously reconcile these two demands, and we refer to this balance as *tool calling with a reasoning nudge*. The idea is to encourage the model to perform just enough reasoning to decide the next action and then prioritize evidence-gathering interactions with the environment over further reasoning. Rather than extending internal chains-of-thought, the agent is nudged toward validating its hypotheses through tool outputs (Yao et al., 2023; Shinn et al., 2023; Schick et al., 2023).

In practice, we did not find a single "golden prompt" that reliably balances reasoning and tool interaction across all model families. For the Claude variants, we found that a quantitative target, e.g., "make 50+ tool calls" or "ideal tool call count is 100"[1] helps shift the balance from reasoning toward interaction with the environment. While the exact number of target tool calls is not important, it serves as a useful north star that biases the model toward action. Under the same prompt, Gemini and Grok runs frequently emitted empty tool calls up to the stated mark. Here, we found that using a qualitative direction such as "You should use tools as much as possible" achieves the same balance.

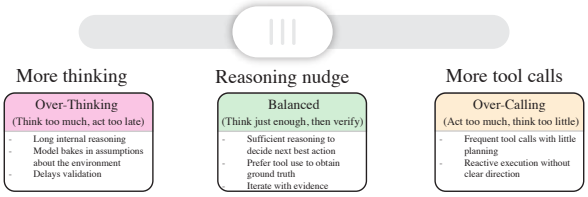


Figure 1: **Reasoning nudges in SSA**: Each family is nudged toward tool use in the form, quantitative or qualitative, it responds to.

2.3 ALIGNING HARNESS WITH THE MODEL'S TOOL USE PREFERENCES

Across agents, while tools function in exactly the same way, models tend to exhibit distinct preferences in how they invoke them. For example, GPT models prefer to update code by using an `apply_patch` command to splice in text from a separate file, formatted in a particular way. Performance depends on honoring these formatting preferences (Yang et al., 2024a; Kim et al., 2026; Wang et al., 2024b; Xia et al., 2024).

Similarly, for Grok-4.20, a single monolithic tool for editing and viewing performs poorly, whereas splitting functionality into atomic operations yields better results even though the functionality remains unchanged. Additionally, viewing line numbers in a file helps most models, but Grok has trouble separating prefixes from line numbers. Here, disabling line-numbers helps the model use the view-tool effectively. These preferences are a by-product of training, and an effective harness should leverage them rather than fight them (Schluntz & Zhang, 2024; Wang et al., 2024a).

This reinforces a broader design principle that agent performance is a function of not only what tools are available but also how naturally those tools align with the model's learned behaviors. A well-designed harness should meet the model where it is, adapting interfaces and feedback to its strengths while still enforcing the invariants needed for reliable execution.

[1]This style of quantitative tool-call prompt first appeared in the Claude Sonnet 4.5 release: `https://www.anthropic.com/news/claude-sonnet-4-5`.

3 SOLUTION DISTANCE: MEASURING TRAJECTORY QUALITY

Motivation. Standard agentic-coding metrics such as pass@1, tool-calls distribution, number of output-tokens summarize *outcome* and *cost* but not the *trajectory quality*. Two runs that achieve similar pass@1 on the same instance can still differ markedly. One may proceed through a clean explore→localize→implement→verify arc with a single decisive edit while another may oscillate between candidate fixes, undo and reapply edits, or commit a correct patch and subsequently retract it. To study and isolate this behavior, we introduce metrics in this subsubsection that answer a single, mechanistic question at every point in a trajectory: *how close is the agent's current repository state to a verified-correct solution* for this instance, and *does it move monotonically towards one or does it backtrack?*

Text-level code space. Let $\mathcal{P}$ be the universe of absolute file paths and Σ^* the set of exact file-content strings. Let x be a map from a file-path to its contents such that $x : \mathcal{P} \to \Sigma^* \cup \{\perp\}$. The ambient text-level code space is

$$\mathcal{X}^{\text{text}} = \{\, x \mid |\{f : x(f) \neq \perp\}| < \infty \,\}, \tag{1}$$

where $x(f)$ is file f's exact text and $\perp$ denotes absence of content. Therefore, creation and deletion are represented by transitions from or to $\perp$. For brevity, we drop the superscript in $\mathcal{X}^{\text{text}}$ and write this text-level space as $\mathcal{X}$.

For a problem instance i (such as a repository-level coding task), let $\mathcal{R}_i$ be the given workspace root, $\mathcal{P}_{\mathcal{R}_i} \subset \mathcal{P}$ the paths under that root, x_i^0 the given code, and $\mathcal{F}_i \subset \mathcal{P}_{\mathcal{R}_i}$ the retained source-path set. The instance-level text subspace is defined as follows:

$$\mathcal{X}_i = \{\, x \in \mathcal{X} : x(f) = x_i^0(f) \;\forall\, f \in \mathcal{P} \setminus \mathcal{P}_{\mathcal{R}_i} \,\}. \tag{2}$$

Thus states in $\mathcal{X}_i$ may vary only on the retained paths under the workspace root. A trajectory state $x_i(t)$ is the accepted textual mutations applied to x_i^0 and projected to this subspace. We write the retained-path projection as $\pi_i(x) = x|_{\mathcal{P}_{\mathcal{R}_i}}$.

Prior Work Generate-and-validate repair systems define *patch* or *search* spaces over syntactic program modifications, AST edits, edit locations, mutation operators, and repair ingredients (Le Goues et al., 2019; 2012; Long & Rinard, 2016; Barr et al., 2014); program-representation work maps code into token, AST-path, data-flow, or graph-structured representations (Alon et al., 2018; Guo et al., 2021); semantic genetic programming defines distances in a semantic output space rather than directly over program syntax (Moraglio et al., 2012); and recent work contrasts discrete symbolic source-token spaces with continuous numerical program spaces (Silva et al., 2026).

Since most coding agents (including SSA) operate on the observable source text and apply edits as text file mutations and evaluation also uses text patches, using $\mathcal{X}$ as a text-level code state space is the most useful. Note that $\mathcal{X}_i$ is an instance-level subspace and not a semantic program space.

Let $O_i : \mathcal{X}_i \to \{0, 1\}$ be the evaluation oracle, with $O_i(x) = 1$ iff materializing state x in the testing environment passes all of the instance's tests. The true solution set in this space is

$$\mathcal{S}_i = \{\, x \in \mathcal{X}_i : O_i(x) = 1 \,\}. \tag{3}$$

Next, to be able to express similarity in the observable space, we define the text-diff projection as

$$\phi_i(x) = \tau\big(\text{diff}(\pi_i(x_i^0), \pi_i(x))\big)\,, \tag{4}$$

where `diff` is the unified diff against the projected base state and τ extracts the patch features defined in Appendix B.1.3.

Definition 3.1 (Recall divergence)**.** The recall divergence from the current state $x_i(t)$ to the nearest oracle-correct state is

$$D_i(t) \;=\; \min_{x^\star \in \mathcal{S}_i} \left(1 - \frac{|\,\phi_i(x_i(t)) \cap \phi_i(x^\star)\,|}{|\,\phi_i(x^\star)\,|}\right), \tag{5}$$

where the minimization is over oracle-correct states with non-empty projected source diffs. The recall denominator is deliberate as it measures recovery of a known-correct fix, while the solution oracle remains responsible for judging whether additional edits are harmless.

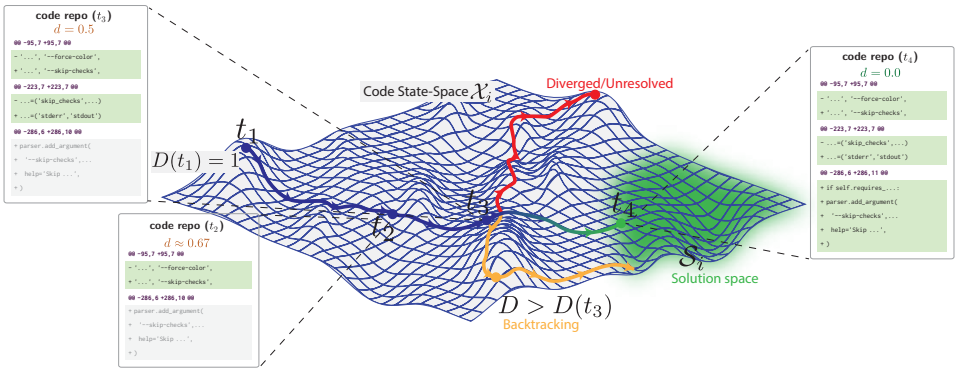


Figure 2: **Agent trajectory in code state-space**. Solution divergence $D(t)$ evolution as agent traverses code state-space $\mathcal{X}_i$ from the given initial state at t_1 till it reaches solution space $\mathcal{S}_i$ at step t_4 along the blue path. Each inset compares the reconstructed live diff with the closest element in $\mathcal{S}_i$. Green lines are reproduced patch features and gray lines are still missing. For the successful work, the trace follows a staircase $D = 1.0 \to D \approx 0.67 \to D = 0.5 \to D = 0$ as the agent accumulates the repair. The red path is an unresolved run, which terminates outside $\mathcal{S}_i$. The orange path backtracks: it transiently moves away from the solution space, $D > D(t_3)$, before returning and resolving. Appendix B.1.3 describes the empirical reference construction and replay procedure.

In practice, $\mathcal{S}_i$ is not available or enumerable. Our experiments therefore approximate Equation 5 with a finite empirical reference set of verified patches and faithful replay of the live repository state (see Appendix B.1.3). The resulting time-series is used for mean-distance curves, per-edit ΔD backtracking, area under the distance curve, and time-to-threshold summaries such as t at $D \leq 0.1$.

Note: D starts at 1 (no edits, no overlap) and falls to 0 once the state $x(t)$ fully reproduces a correct repair under the projection. Since D could rise or fall, the per-edit change $\Delta D > 0$ becomes a clean signal of backtracking as we formalize in Section 3.1.

3.1 Level sets and backtracking

Consider N distance (Eq.(5) trajectories $D_r : [0,1] \to [0,1]$, each a right-continuous step function on normalized time t with $t = 0/1$ at beginning/end of the agent run. The mean, $\mathbb{E}\, D(t)$ (mean over all instances) can descend gradually even when individual runs make a single sharp drop at different times (see Figure 6 in Appendix B.1.4).

Definition 3.2 (Level-set processes)**.** For each run r and level $\theta \in [0,1]$, define the first entry into the distance sublevel set and the best-so-far distance by

$$T_r(\theta) = \inf\{t \in [0,1] : D_r(t) \leq \theta\}, \qquad M_r(t) = \min_{0 \leq s \leq t} D_r(s), \tag{6}$$

Finite step trajectories ensure that every finite crossing is attained. Let $R \sim \mathrm{Unif}\{1, \ldots, N\}$ denote a uniformly selected run. The corresponding *level-survival curve* is

$$S_\theta(t) := \Pr\{T_R(\theta) > t\} = \Pr\{M_R(t) > \theta\} = 1/N \sum_{r=1}^{N} \mathbf{1}\{T_r(\theta) > t\}. \tag{7}$$

It is the fraction of runs that have not yet reached level θ, and is non-increasing in both t and θ.

Next, we write the cohort averages as

$$\bar{D}(t) = 1/N \textstyle\sum_{r=1}^{N} D_r(t), \quad \bar{M}(t) = 1/N \textstyle\sum_{r=1}^{N} M_r(t).$$

Theorem 3.3 (Level-set decomposition)**.** *For every $t \in [0,1]$, these averages satisfy*

$$\bar{D}(t) = \bar{M}(t) + \bar{B}(t), \tag{8}$$

where $\bar{M}(t) = \int_0^1 S_\theta(t)\, d\theta$ and $\bar{B}(t) := 1/N \sum_{r=1}^{N} [D_r(t) - M_r(t)] \geq 0$.

Proposition 3.4 (Zero backtracking). *For any fixed $t \in [0, 1]$, $\bar{B}(t) = 0$ iff $D_r(t) = M_r(t)$ for every run r. Consequently, $\bar{B} \equiv 0$ iff every run is non-increasing.*

Proofs of Theorem 3.3 and Proposition 3.4 are given in Appendix B.1.3.

Thus, backtracking $\bar{B}(t)$ is the mean distance above each run's best previous state: coverage previously attained but currently lost. Interestingly, this signal can remain positive while a run recovers from an earlier reversal.

4 Experimental Results

We evaluate SSA on four agentic benchmarks, namely, SWE-Bench-Verified (Jimenez et al., 2024), SWE-Bench-Pro (Deng et al., 2025), Terminal-Bench-2 (Merrill et al., 2026) and Deep-SWE (DataCurve, 2026) and seven model families: Claude, GPT, Gemini, Grok, Kimi, Muse Spark and Qwen. Under one harness, SSA reproduces or exceeds the published pass@1 for 61 of the 66 model–benchmark pairs with a published figure, and is ahead of widely-used open-source harnesses. Next, in this Section, we analyze trajectories (154k in total) from all these runs to better understand how each model-family approaches problem solving, information that is not expressed just looking at pass@1.

4.1 Setup and protocol

Experimental-setup details (compute cluster, model serving, per-model inference settings, and the evaluation-hygiene protocol) are given in Appendix G, and the prompts in Appendix H. All four benchmark ship with pre-existing tests. The SWE benchmarks serialize the agent's workspace changes into a patch and replay it in a separate evaluator container. Terminal-Bench-2 evaluates in-place but imposes explicit per-task compute and wall-time limits that the SWE benchmarks do not. Throughout, we report pass@1 following the convention of Chen et al. (2021). We follow a strict routine of counting every failure (agent or infra) into pass@1 and not alter denominator which otherwise can inflate the scores. Each model is run five times on the full benchmark. In total we evaluate 38 models (see Appendix C). The main sweep of 21 models over three benchmarks yields $(500 + 731 + 89) \times 5 \times 21 = 138{,}600$ trajectories; the second Terminal-Bench-2 setting for the 13 closed models adds $89 \times 5 \times 13 = 5{,}785$, and the 17 models on Deep-SWE add $113 \times 5 \times 17 = 9{,}605$ yielding $153{,}990$ trajectories in total, ≈ 154k, which we analyze further for model behavior.

4.2 Published pass rates are recovered under one harness

SSA reproduces or exceeds official model-card numbers under a single harness. Every model is run under the same execution loop, feedback semantics and evaluation protocol (Section 4.1); only the narrow interface adapters of Appendix G differ. Tables 1 and 2 record, for each model and benchmark, whether the figure the provider publishes falls inside SSA's 95% confidence interval, below it, or above it. Appendix C reports the mean per-run resolution rate with 95% confidence intervals over all per-instance trials ($2{,}500$ for SWE-Bench-Verified, $3{,}655$ for SWE-Bench-Pro, 445 for Terminal-Bench-2, and 565 for Deep-SWE). Of the 66 model–benchmark pairs for which a published figure exists (43 in Table 1, 17 in Table 2, and 6, all reproduced, from models added after the main sweep that appear only in Appendix C), 61 fall within or below SSA's interval (44 reproduced and 17 exceeded) and 5 lie above it, four of them on SWE-Bench-Pro, whose public containers leak future git history (Appendix E; per-model figures in Appendix C). These results characterize model–harness configurations under the stated evaluation conditions and they do not establish a general ordering of provider capability.

For Terminal-Bench-2 we report two settings (motivated in Appendix D): *Constrained*, which honors the benchmark's memory and time budgets, and *Unconstrained*, which removes them and acts as an upper bound on performance. Providers report their figures under the Constrained setting, except for the Qwen models, whose published figures are Unconstrained; Table 1 compares each published figure against the matching setting. Open-source models were run only Unconstrained. Overall, we found that removing the memory and agent-time limits lifts pass@1 by roughly 5-10% across models (see Appendix D).

Table 1: **SSA reproduces or exceeds published pass@1.** Each provider's model-card figure is compared with SSA's five-run pass@1 and its 95% confidence interval. Terminal-Bench-2 uses the setting the provider reports (Constrained for closed models, Unconstrained for Qwen). Rows are grouped by model family, with families in alphabetical order and models by version within a family. The order carries no ranking. Full numbers, and six further pairs from models added after the main sweep, are in Appendix C.

Family	Model	SWE-Bench-Verified	SWE-Bench-Pro	Terminal-Bench-2
CLAUDE	Opus 4.6	✓	✓✓	✓✓
	Opus 4.5	✓	✓✓	✓✓
	Sonnet 4.6	✓	—	✓✓
	Sonnet 4.5	✓	—	✓
	Sonnet 4.0	✓	—	—
	Haiku 4.5	✓	—	✓
GEMINI	Gemini 3.1 Pro	✓	✓	✓
	Gemini 3 Flash	✓	—	✓✓
GPT	GPT-5.4	—	✓✓	✓
	GPT-5.3	—	✓✓	✓
	GPT-5.2	✓	✓	✓
	GPT-5.2-codex	—	▲	✓
	GPT-OSS 120B	✓✓	—	—
	GPT-OSS 20B	✓	—	—
GROK	Grok 4.2	—	—	—
QWEN	Qwen3-Coder Next	✓	✓✓	✓
	Qwen3-Coder 480B	▲	—	—
	Qwen3-Coder 30B	✓	—	—
	Qwen3.6 35B-A3B	✓	▲	✓
	Qwen3.6 27B	✓	▲	✓
	Qwen3.5 27B	✓✓	▲	✓✓
Totals		14 ✓ 2 ✓✓ 1 ▲	2 ✓ 5 ✓✓ 4 ▲	10 ✓ 5 ✓✓ 0 ▲

✓ reproduced ✓✓ exceeded ▲ not reached — nothing published

On Deep-SWE, whose tasks require substantially larger repairs than either SWE benchmark, the reference numbers come from vendor model cards where one exists and otherwise from the DataCurve v1.1 artifact, against SSA's five attempts at a single fixed configuration (Appendix C). The token-cost view of the SWE results, a Pareto plot of total output tokens against pass@1, is in Appendix B.3 (Figure 11).

Taken together, the benchmark results show that a harness built to minimize the intent-execution gap, with only narrow family-level interface adapters, recovers published performance, within confidence intervals, for 61 of the 66 model-benchmark pairs, and out-performs the default open-source harnesses practitioners typically use.

Family	Model	Deep-SWE
CLAUDE	Opus 5.5	✓
	Opus 5.0	✓✓
	Opus 4.8	✓
	Sonnet 5.0	✓
	Fable 5	✓
GEMINI	Gemini 3.8 Flash	✓
GPT	GPT-6 (Astra/Sol/Luna)	✓✓/✓✓/✓
	GPT-5.6 (Sol/Terra/Luna)	✓/✓/✓
	GPT-5.5	✓✓
GROK	Grok 4.7	✓
KIMI	Kimi-K3	✓✓
MUSE SPARK	Muse Spark 1.3	✓
QWEN	Qwen3.8 27B	✓

Table 2: **Deep-SWE.** Marks as in Table 1, against the provider's system or model card where published, otherwise the DataCurve v1.1 artifact.

4.3 TRAJECTORY-LEVEL ANALYSIS

Performance via pass@1 reports *whether* a run was successfully resolved but says nothing about *how*. Using an (i) LLM-judge (see Appendix B.1.1 for details), we label each trajectory's tool-calls across R1-R8 rubrics as detailed in Table 3. We partition tool calls into source-edit, scratch-testing (tests written by the agent itself), and suite-testing labels (pre-existing tests in the repository). We calculate the per-cycle R6 phase composition (`explore/localize/implement/verify`). Next, (ii) with code state-space, the solution-distance

curve $D(t)$ and backtracking signal $B(t)$ (see Appendix B.1.7) are captured. Together, these metrics give us a better understanding of how the models work in the setting of an agentic problem solving.

The full per-model data on these metrics for both SWE benchmarks can be found in Appendix B.1.3 (Figures 14–30), and we summarize three findings below.

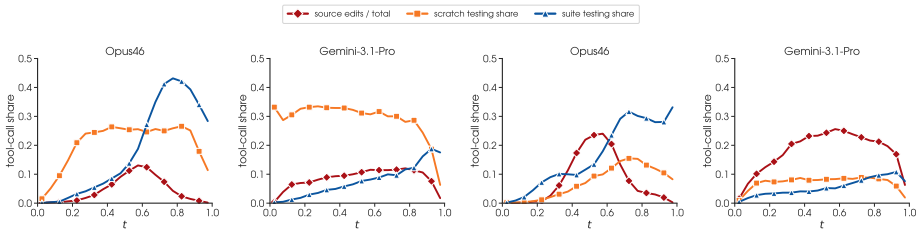


Figure 3: **Composition of activities is a model signature**. Share of three activities (source editing, scratch testing, suite testing) vs normalized cycle position of agent trajectories between 0 and 1, averaged across full benchmark instances (5 runs per instance). For SWE-Bench-Verified (left), Opus 4.6 scratch-tests from early on, edits peak mid-trajectory, and suite testing dominates towards the end; Gemini 3.1 Pro instead scratch-tests at a steady, high rate for almost the whole trajectory, with a low edit share and suite testing only in the final stretch. On SWE-Bench-Pro (right), where tasks need more lines of code, both models shift towards editing and the scratch-testing share falls, Gemini's most sharply. Remaining 19 models in Appendix B.1.3 (Figures 14–15, 27–28).

Trajectories taken by different models can display very diverse behavior even when they have similar resolve rates. On both SWE-Bench-Verified and SWE-Bench-Pro, the trajectory metrics (edits/testing share across agent steps) for Opus 4.6 and Gemini 3.1 Pro look very different (Figure 3) even though their pass@1 on SWE-Bench-Verified is identical and close on SWE-Bench-Pro. The two treat testing differently: Opus 4.6 ramps up suite testing in the second half and devotes most of its late-trajectory calls to it, whereas Gemini 3.1 Pro relies on scratch tests throughout and turns to the suite only at the end. For both models, we see a higher share of edits in SWE-Bench-Pro compared to SWE-Bench-Verified owing to the higher difficulty of tasks in the former.

Backtracking is decoupled from solve rate. Figure 4, on SWE-Bench-Verified (top row), Opus 4.6 and Gemini 3.1 Pro post the same pass@1 but their behavior in code state-space is significantly diverse. Gemini 3.1 Pro has 3× higher backtracking than Opus 4.6; Appendix Figure 10 traces one such Gemini 3.1 Pro backtrack, a deliberate revert that uncovers a second bug before the run resolves. The distance curves $\bar{D}(t)$ follows different shapes to reach the solution space with Opus 4.6 more flat and then steep descent while Gemini 3.1 Pro has more steady decay over the normalized time. The level curves $S_\theta(t)$ are tighter in spread due to dataset requiring atomic edits and mostly single-file. Next, on the more complicated Deep-SWE dataset, GPT-5.6-Sol and Grok 4.7 also post the same pass@1 and both have backtracking terms near zero, yet their level families occupy different parts of the trajectory. GPT-5.6-Sol closes most of the distance in a short burst before mid-run, while Grok 4.7 makes almost no progress until the final third and then closes it in a short, steep drop. It is also coherent from edit/testing share plots in Appendix B.6 that edits land later for Grok 4.7, hence distance drops later in time. Overall, we find that neither backtracking nor timing is recoverable from pass@1; both reflect model-internal behavior on the way to the same success. Per-model backtracking rates for every model and benchmark are given in Appendix B.3.

Different model-families execute same tasks with different phase schedules. Appendix Figures 20–21 split each trajectory into explore/localize/implement/verify phases for every model. The implementation phase peaks at different positions and widths, and the verify phase carries different long-tails. Note that the sharp drop in 'explore' in Opus 4.6, GPT-5.4, and Qwen3 coder 480B is almost always due to doing an initial workspace-tree survey such as with `ls -l` tool-calls.

Terminal-Bench-2 scores are sensitive to evaluation infrastructure. Beyond model quality, Terminal-Bench-2 results also reflect the stability of the serving and execution stack: inference-backend throughput, node packing, command batching, and the benchmark's own memory and time budgets each move pass@1 by 5–10 points. We analyse these effects in Appendix D.

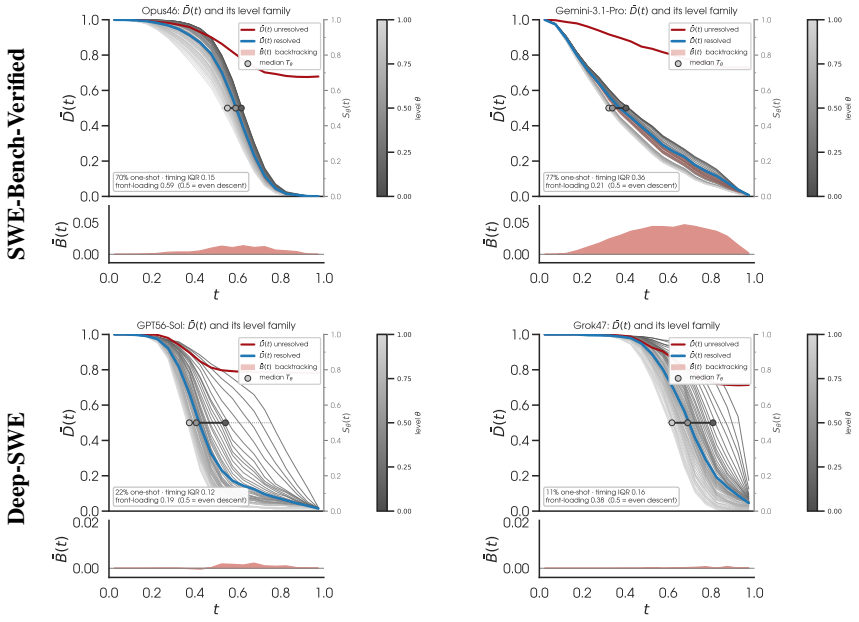


Figure 4: **Level families separate shape, timing and backtracking.** The level families of Section 3 for two models on SWE-Bench-Verified (top) and two on Deep-SWE (bottom); blue is the mean $\bar{D}(t)$ of the resolved runs, dark red that of the unresolved runs. On SWE-Bench-Verified both Opus 4.6 and Gemini 3.1 Pro write the patch in a single edit on most runs (70% and 77% one-shot, respectively), so their mean curves differ not in shape but in timing and backtracking: Gemini 3.1 Pro's solve times are more than twice as dispersed (timing IQR, the interquartile range of the runs' half-way times $T_r(0.5)$: 0.36 vs. 0.15) and its $\bar{B}(t)$ peaks three times higher (0.047 vs. 0.014), which is what makes its mean descent smooth and near-linear. On Deep-SWE, GPT-5.6-Sol and Grok 4.7 are with backtracking near-zero, and what separates them is *when* progress happens. GPT-5.6-Sol closes half the distance by $T_{0.5} = 0.41$; Grok 4.7 stays near $D = 1$ until $t \approx 0.6$ and then falls steeply, reaching $T_{0.5} = 0.69$. The front-loading index in each subplot is $(T_{0.5} - T_{0.9})/(T_{0.1} - T_{0.9})$, with T_θ the median over runs of the first-entry time $T_r(\theta)$ of Section 3 (where S_θ crosses $1/2$): 0.5 is an even descent, lower means most of the distance falls early. Level families for all models are in Appendix B.1.3.

5 Conclusion

As frontier models grow more capable, the harness is critical in determining whether that capability is translated into agent performance or lost in the interface between model and environment. A good harness (i) applies the principles that help every model family, such as line-anchored edits, explicit post-edit diffs, and faithful tool-output serialization, and (ii) adapts to the model's preferences where model-families genuinely differ. Our experiments show that minimizing the bidirectional intent–execution gap is enough to bring the models we test to or above their reported model card numbers, within confidence intervals, in 61 of 66 cases on SWE-Bench-Pro, SWE-Bench-Verified, Terminal-Bench-2 and Deep-SWE with no task-specific tuning; 4 of the remaining 5 are on SWE-Bench-Pro, whose public containers carry a leakage artifact (Section 4.2). The gap between documented and reproduced agent performance is therefore in large part harness friction rather than any fundamental limitation of a simple ReACT (Yao et al., 2023) based single-agent design. And because no tool surface is uniformly best, the goal is to have a harness that holds execution invariants fixed while meeting each model-family where its training left it.

We see that as models converge near the top of these benchmarks, pass@1 stops separating them but their trajectories do not. Our code state-space analysis (solution-distance, backtracking) and phase decomposition show that models with the same solve rate can differ in the path taken by their corresponding agents in code-state space. Going beyond accuracy numbers and understanding trajectory behavior is what makes model-harness co-design possible.

AI USE STATEMENT

In this work, we used generative AI tools as writing and editing assistants. Specifically, (a) to polish the wording of the introduction, (b) developing figures and graphics in tables which the authors then reviewed and edited. AI was also used to help collect, format and check bibliographic references. We have not used generative AI tools to generate research ideas, to design experiments, or to produce or alter experimental results. All four benchmark runs, trajectory measurements and solution distance analyses reported here were carried out with the authors' harness and evaluation pipeline. We have reviewed all AI-assisted work and was checked against the underlying data and edited by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We have aimed to make every number in this paper traceable to a released artifact or a documented procedure. SSA source code, together with the exact configuration files, prompts and tool specifications used for every run, will be released publicly. Appendix G lists, per benchmark, the compute-setup, inference and reasoning settings, prompts and agent tools for every model. Appendix H specifies all system and user prompts. Appendix I gives the complete tool-schemas. Appendix J lists the exact vLLM serve command for every open-weights model. Finally, Appendix K describes the one harness-side parsing fix we applied.

Appendix C reports every model's pass@1 with its 95% confidence interval, pass@2, and the published figure.

Tool-call labels come from a Claude Opus 4.6 judge whose windowed, forced-tool-output classification pipeline is specified in Appendix B.1.1 and whose fixed rubric, with every label value defined, is given in Table 3 (Appendix B.1.2).

REFERENCES

Amazon AGI. The Amazon Nova family of foundation models. https://aws.amazon.com/nova/, 2024.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties, 2018. URL https://arxiv.org/abs/1803.09544.

Anthropic. The Claude 4 model family: System cards and capability notes. https://www.anthropic.com/claude, 2025.

AWS. Strands Agents: A model-driven SDK for building AI agents. https://strandsagents.com, 2025. Open-source SDK.

Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pp. 306–317, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635898. URL https://doi.org/10.1145/2635868.2635898.

BerriAI. LiteLLM: A unified gateway and proxy for LLM APIs. https://github.com/BerriAI/litellm, 2023. Open-source library.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, et al. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, et al. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

DataCurve. Deep-swe: A benchmark of large, multi-file software engineering tasks. https://deepswe.datacurve.ai/, 2026. Leaderboard artifact v1.1, accessed 2026-09-15.

Google DeepMind. Gemini 3: Pro and Flash. https://deepmind.google/technologies/gemini/, 2025.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, et al. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, et al. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Quentin Gallouédec and Kashif Rasul. Agentic RL: Token-in, token-out done right, 2026. Accessed 2026-06-07.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, et al. GraphCodeBERT: Pre-training code representations with data flow, 2021. URL https://arxiv.org/abs/2009.08366.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, et al. Qwen2.5-coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

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

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Myeongsoo Kim, Joe Hsu, Dingmin Wang, Shweta Garg, Varun Kumar, and Murali Krishna Ramanathan. CODESTRUCT: Code agents over structured action spaces, 2026. URL https://arxiv.org/abs/2604.05407.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2023. URL https://arxiv.org/abs/2205.11916.

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

Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 3–13. IEEE Press, 2012. ISBN 9781467310673.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Commun. ACM*, 62(12):56–65, November 2019. ISSN 0001-0782. doi: 10.1145/3318162. URL https://doi.org/10.1145/3318162.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems, 2023. URL https://arxiv.org/abs/2306.03091.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, et al. Agentbench: Evaluating llms as agents, 2025. URL https://arxiv.org/abs/2308.03688.

Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems, 2016. URL https://arxiv.org/abs/1602.05643.

Inbal Magar and Roy Schwartz. Data contamination: From memorization to exploitation, 2022. URL https://arxiv.org/abs/2203.08242.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

MiniMax AI. MiniMax-M2: A foundation model with extended context and tool use. https://huggingface.co/MiniMaxAI, 2025.

Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature - PPSN XII*, pp. 21–31, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32937-1.

OpenAI. The GPT-5 model family and the gpt-oss open-weights release. https://openai.com/, 2025a.

OpenAI. openai-harmony: Response format for the gpt-oss models. https://github.com/openai/harmony, 2025b. Accessed 2026-06-07.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, et al. ToolLLM: Facilitating large language models to master 16000+ real-world apis, 2023. URL https://arxiv.org/abs/2307.16789.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, et al. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents, 2025. URL https://arxiv.org/abs/2504.08703.

Oscar Sainz, Jon Ander Campos, Iker García-Ferrero, Julen Etxaniz, Oier Lopez de Lacalle, and Eneko Agirre. Nlp evaluation in trouble: On the need to measure llm data contamination for each benchmark, 2023. URL https://arxiv.org/abs/2310.18018.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

Erik Schluntz and Barry Zhang. Building effective agents. https://www.anthropic.com/research/building-effective-agents, 2024. Anthropic engineering blog.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

André Silva, Gustav Thorén, and Martin Monperrus. Gradient-based program repair: Fixing bugs in continuous program spaces, 2026. URL https://arxiv.org/abs/2505.17703.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), 2024a. URL https://arxiv.org/abs/2308.11432.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. *arXiv preprint arXiv:2402.01030*, 2024b. URL https://arxiv.org/abs/2402.01030.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, et al. Openhands: An open platform for ai software developers as generalist agents, 2025. URL https://arxiv.org/abs/2407.16741.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, et al. Emergent abilities of large language models, 2022. URL https://arxiv.org/abs/2206.07682.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, et al. AutoGen: Enabling next-gen llm applications via multi-agent conversation, 2023. URL https://arxiv.org/abs/2308.08155.

xAI. Grok 4.20 reasoning model. https://x.ai/, 2025.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, et al. The rise and potential of large language model based agents: A survey, 2023. URL https://arxiv.org/abs/2309.07864.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024. URL https://arxiv.org/abs/2407.01489.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, et al. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024. URL https://arxiv.org/abs/2404.07972.

Omry Yadan. Hydra – a framework for elegantly configuring complex applications. https://github.com/facebookresearch/hydra, 2019. GitHub repository.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, et al. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

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, 2024a. URL https://arxiv.org/abs/2405.15793.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. Swe-bench multimodal: Do ai systems generalize to visual software domains?, 2024b. URL https://arxiv.org/abs/2410.03859.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024. URL https://arxiv.org/abs/2404.05427.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023. URL https://arxiv.org/abs/2306.05685.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024. URL https://arxiv.org/abs/2307.13854.

APPENDIX CONTENTS

The appendix is organized into the nine parts below. All entries are hyperlinked; the number on the right is the page on which the (sub)section begins.

A SSA Architecture **15**
B Methodology and Metrics **16**
B.1 Methodology 16
B.1.1 LLM judge for tool-call classification 16
B.1.2 Classification rubric 16
B.1.3 Solution-distance methodology details 18
B.1.4 Illustrative examples 20
B.1.5 Faithful state replay 26
B.1.6 Resolved endpoints: self-anchor 27
B.1.7 Backtracking quantification 27
B.1.8 Limitations of the solution-distance metric 27
B.2 Metrics 28
B.3 Metrics: SWE-Bench-Verified 29
B.4 Metrics: SWE-Bench-Pro 43
B.5 Metrics: Terminal-Bench-2 56
B.6 Metrics: Deep-SWE 66
C Per-Model Pass Rates by Family **76**
D Terminal-Bench-2: Infrastructure Sensitivity **82**
E Evaluation Integrity **83**
F SWE-Bench-Pro Git Leakage **83**
F.1 Sanitized evaluation 83
F.2 Direct leakage metric 83
F.3 Empirical impact 84
F.4 Open-source models and explicit git instructions 85
G Configurations **86**
G.1 SWE-Bench-Verified 88
G.2 SWE-Bench-Pro 89
G.3 Terminal-Bench-2 90
G.4 Deep-SWE 91
H Prompts **92**
H.1 Generic SWE prompts 92
H.2 Git PR prompt with GPT-specific instructions 96
H.3 Claude specific prompt with reasoning/tool-call nudge 98
H.4 Terminal-Bench-2 prompts 99
H.5 Deep-SWE prompts 105
I Tool Definitions **108**
I.1 Generic shell and editor tools 109
I.2 OpenAI-specific tools 113
I.3 xAI-specific tools 115
J vLLM Serving Configurations **116**
J.1 Per-model serve commands 117
J.2 A note on vLLM flags 119
K Harmony-format parsing failures in the harness **120**
K.1 The symptom: empty and malformed tool calls 120
K.2 The serving boundary: stateless re-tokenization 121
K.3 Token-level evidence 121
K.4 The fix: surface a single retryable error 123

A SSA ARCHITECTURE

We use Simple Strands Agent (SSA) as the harness for all our experiments (see Figure 5 for architecture). It has the following components: tasks, prompts, models, tools, environments, and hooks. The input task is converted into prompts, with the agent alternating between model calls and tool execution. Tools act on an isolated environment (we use Docker for all our experiments) and hooks enforce control policies such as context management and content filtering.

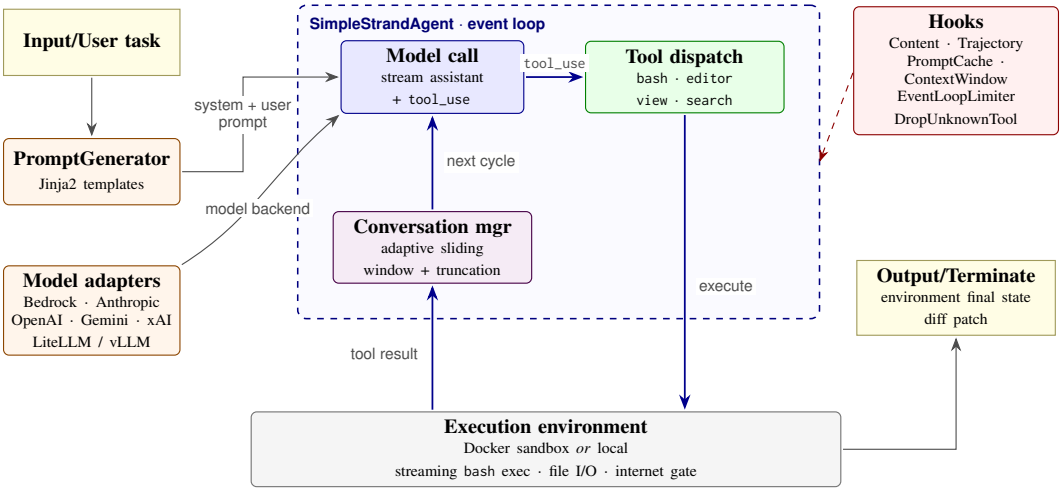


Figure 5: **SSA architecture**. Tasks and model adapters feed a cyclic `SimpleStrandAgent` loop. The loop streams model output, dispatches tools, executes them in the environment, and appends tool results to conversation state. Hooks validate bounds and record the loop; termination extracts the final environment state and diff patch.

Task and prompts. The task to be solved is converted to a user prompt and paired with system prompts that may include model-specific instructions (see Appendix H for complete set of SSA prompts). Model specific changes may include reasoning nudge (see Section 2.2) and tool-specific instructions such as `apply_patch` for GPT family.

Model interface. Adapters map model-provider APIs into a common event interface: assistant text, reasoning content, tool-use requests, stop reasons, usage, cache metadata, and recoverable errors. The event loop is the same across all models.

Agent core and conversation. The core is a thin control loop: call the model, execute requested tools, append tool results, and repeat. The conversation manager preserves valid tool-use/tool-result structure while compressing large outputs or trimming old turns when context gets too long.

Tools and environment. SSA's tool surface is deliberately small as it mainly relies on bash and file viewing/editing tools. Tools delegate command execution to the environment. For a complete list of tools, see Appendix I.

Hooks and instrumentation. Hooks validate model output, drop malformed tool calls, manage context-windows, enforce loop bounds, add cache markers, and record trajectories and metrics for analysis.

B METHODOLOGY AND METRICS

All quantitative results in Section 4 are computed from two artifacts produced by every SSA evaluation. (1) A `metrics.json` with the per-cycle token, latency, cache and tool-success counters emitted by the harness, and (2) `trajectory.json` containing every user / assistant message (with reasoning blocks, tool calls and tool results) verbatim. The trajectory is the authoritative input to all behavioral analyses, whenever counters in `metrics.json` disagree with the trajectory (e.g. retry due to exception handling) the trajectory wins.

B.1 METHODOLOGY

All behavioral analyses are driven by a per-call LLM judge (Zheng et al., 2023) that classifies every tool invocation against a fixed rubric. This subsection describes the judge pipeline, the rubric it operates on, and the system prompt under which it runs.

B.1.1 LLM JUDGE FOR TOOL-CALL CLASSIFICATION

Every individual tool call in the trajectory is classified against a fixed rubric by a Claude Opus 4.6 judge (Anthropic, 2025). The pipeline works as follows:

1. **Windowed classification.** Each trajectory is split into sliding windows of LLM calls. The judge sees the window rendered as a transcript (preceding events tagged `[context]`, and the callable event tagged with `<<CALL n>>`) and LLM labels the provided event in the window in a single response. Local context lets the judge disambiguate repeated commands by surrounding state.
2. **Forced tool-use output.** The judge cannot reply with free-form text. Its only allowed output is a single call to the `submit_classifications` tool whose schema only accepts an `enum`. Outputs are validated client-side before acceptance.

B.1.2 CLASSIFICATION RUBRIC

Table 3 lists every field the judge assigns to a call. R1–R5 carry the bulk of the behavioural signal, R6 supplies a coarse-phase label used in Gantt-style trajectory plots, R7 is a reliability self-report used to flag low-confidence buckets in downstream analyses, and R8 disambiguates the raw `error` status, because a non-zero `bash` exit (`grep` returning "no match", `pytest` reporting test failures, etc.) is often not a tool malfunction.

Table 3: Rubrics the LLM judge assigns to every tool call, with per-value definitions. R5 (`target_paths`) is the only open-valued field; the rest are picked from the listed enums. Coloured rows are field separators.

Value	Description
R1 category	
`explore_navigate`	Orient in the repo: list / inspect structure, history, or state (`ls`, `tree`, `view` a directory, `git log/status`, `find -type d`).
`search_locate`	Hunt for a symbol, string, or file across the tree (`grep -rn`, `rg`, `find -name`).
`read_code`	Read the contents of a specific source / test / config file (`cat`, `head`, `view <file>`, `sed -n '1,80p'`).
`edit_source`	Modify non-test project source code (`str_replace` on `<file>.py`, `sed -i` on a source file, `apply_patch` updating a source file).
`edit_test`	Modify the project's own test files (under `tests/`, files matching `test_*.py`).
`write_scratch`	Create or modify an agent-authored throwaway repro / debug script (e.g. `cat > /testbed/reproduce.py`, `create /testbed/check.py`).
`run_suite`	Run the project's real test suite (`pytest <project tests>`, `python -m unittest`, `./runtests.py`, `tox`).

continued on next page

Table 3 – continued

Value	Description
`run_scratch`	Execute a throwaway repro or probe (`python reproduce.py`, `python -c "<repro snippet>"`).
`inspect_runtime`	Quick non-repro runtime probe (`python -c "import x; print(x.__version__)"`, `pip show`, `-version`).
`vcs`	Version-control mutation or inspection beyond log/status (`git diff`, `git apply`, `git stash`, `git checkout -- f`, `git add`).
`env_setup`	Install / build / configure the environment (`pip install -e .`, `conda ...`, `python setup.py build_ext`, `export`).
`other`	None of the above or undeterminable (miscellaneous shell plumbing).
R2 target_kind	
`project_source`	Real, non-test source code shipped by the project (under the package tree, not under `tests/`).
`project_test`	The project's own test files (e.g. `astropy/table/tests/test_*.py`).
`scratch`	Agent-authored throwaway file — repro / verify / debug — including `/tmp/*` and files at `/testbed/` root that don't belong to the package.
`build_config`	Build, packaging, or CI config: `setup.py`, `pyproject.toml`, `conftest.py`, CI YAML.
`vcs_internal`	VCS metadata only (the `.git` directory).
`none`	No file target (e.g. `python -version`, `cd`, `echo`).
R3 mutation	
`read_only`	Pure read or inspection; no filesystem change.
`mutates_source`	Writes / changes project source or test files.
`mutates_scratch`	Writes / changes only scratch / throwaway files.
`destructive`	Broad delete / reset / overwrite: `rm -rf`, `git reset -hard`, `git checkout .`, `apply_patch *** Delete File`.
R4 test_activity	
`suite`	Runs or edits the project's real test suite.
`scratch`	Runs or writes a throwaway repro / verification script.
`none`	Not a testing action.
R5 target_paths	
list	Concrete file path(s) the call reads or writes. Resolves a leading `cd /testbed &&` into absolute paths where unambiguous; otherwise the path as written. `[]` when no file target. Deduplicated within a single call.
R6 phase	
`explore`	Understanding the repo or reproducing the bug.
`localize`	Narrowing in on the offending code.
`implement`	Writing the fix.
`verify`	Running tests or otherwise confirming the fix.
`other`	Setup, cleanup, or otherwise undeterminable.
R7 confidence	
`high`	Judge is highly confident in R1 and R2 for this call.
`medium`	Judge has moderate confidence; downstream code may down-weight or audit these.
`low`	Judge is unsure; flagged for review and excluded from precision-sensitive aggregates.
R8 error_kind	
`none`	Call did not error (result status was success or unknown). Forced to `none` when result status is not `error`.

continued on next page

Table 3 – continued

Value	Description
`tool_invocation`	The tool (or the CLI it dispatched) could not process the call — the failure happens *before* the requested operation runs: `command not found`, shell syntax error, `No such file or directory`, permission denied, a subprocess rejecting its CLI arguments (`unrecognized argument`, argparse errors), an inline `python -c SyntaxError`, or a `str_replace_editor` / `apply_patch` failure (target/`old_str` not found, patch context mismatch).
`subprocess_runtime`	The tool accepted and dispatched the call and the subprocess then failed mid-execution — it parsed fine but the program inside threw: a Python `Traceback` from a repro script, a runtime `ImportError` / `AttributeError`, a `conftest` / fixture crash while loading tests, or a `pip install` whose build step failed.
`benign`	Expected non-zero exit that is not a tool malfunction: `grep` / `rg` no-match (exit 1), `pytest` reporting test failures or "no tests ran" (the run executed fine), `[ -f x ]` test conditional, `git diff --exit-code` / `--quiet` used as a check, a pre-fix repro deliberately demonstrated to fail.
`uncertain`	Status is `error` but the output is empty or truncated and the judge cannot tell which bucket.

The LLM judge rubrics obtained from Table 3 are used to inform numerous metrics from the agent trajectories. The tool-call distribution (Figures 22-23, 35-36, 44-45) and bash sub-command taxonomy come from R1, the edit-share (Figures 14-15, 27-28, 42-43) from R3, the scratch-vs-suite testing share (Figures 14-15, 27-28, 42-43), the explore→localize→implement→verify Gantt plots (Figures 20-21, 33-34, 38-39) from R6, and the tool-error rate (Figures 18-19, 31-32, 40-41) from R8. The union of the two real-failure buckets (`tool_invocation` and `subprocess_runtime`) is substantially lower than the raw `tool_use.error_count` reported by the harness. Wherever the figures below refer to "`R8 = genuine`" they mean this union, benign and uncertain calls are excluded.

B.1.3 Solution-distance methodology details

Section 3 defines the text-level state space and the ideal solution distance over the oracle-defined solution set $\mathcal{S}_i$. This section provides the empirical approximation used in the experiments: patch-feature extraction, reference-set construction, replay indexing, and endpoint corrections.

Proofs of the level-set results.

Proof of Theorem 3.3. As defined in Section 3.1, a run has not reached θ by t exactly when $M_r(t) > \theta$. Applying this straightforward identity

$$z = \int_0^1 \mathbf{1}\{z > \theta\}\, d\theta,$$

to each $M_r(t) \in [0, 1]$, we get the following

$$\begin{aligned} \int_0^1 S_\theta(t)\, d\theta &= \int_0^1 \frac{1}{N} \sum\nolimits_r \mathbf{1}\{M_r(t) > \theta\} d\theta \\ &= \frac{1}{N} \sum\nolimits_r \int_0^1 \mathbf{1}\{M_r(t) > \theta\} d\theta \\ &= \frac{1}{N} \sum\nolimits_r M_r(t) = \bar{M}(t). \end{aligned}$$

By the definitions of $\bar{D}$ and $\bar{M}$, $\bar{D}(t) - \bar{M}(t) = N^{-1} \sum_r [D_r(t) - M_r(t)] = \bar{B}(t)$. Moreover, $M_r(t) \le D_r(t)$ for every run, hence $\bar{B}(t) \ge 0$. □

Proof of Proposition 3.4. Each difference $D_r(t) - M_r(t)$ is nonnegative, so their mean is zero exactly when every difference is zero. If $D_r = M_r$ at all times t, then D_r is non-increasing because M_r is

non-increasing. Conversely, a non-increasing trajectory always equals its running minimum, making backtracking to be zero. □

The empirical solution subset. The oracle-correct set $\mathcal{S}_i$ is unobservable, so we approximate it by a deduplicated empirical subset $\tilde{\mathcal{S}}_i \subseteq \mathcal{S}_i$ assembled from the runs we actually have. Each SWE instance (in both SWE-Bench-Verified and SWE-Bench-Pro) was attempted exactly 105 times in our sweep (21 models $\times$ 5 runs), and every resolved run contributes one verified-correct final code state. This provides a rich set of data to construct/approximate $\mathcal{S}$. For data instances where no executed run produced any resolved status, we at least have the provided gold patch. Near-identical states are merged at Jaccard similarity $\rho = 0.9$. Most resolved patches stay distinct, so $|\tilde{\mathcal{S}}_i|$ should not be read as a count of semantic solution classes. In our collective runs, on SWE-Bench-Verified, resolved-run references already cover 474 of 500 instances with a mean of ~ 82 resolved runs per covered instance, and the remaining 26 fall back to the gold patch. On SWE-Bench-Pro, the corresponding numbers are 562 of 731 instances covered, with a mean of ~ 60 resolved runs per covered instance, and the remaining 169 instances fall back to the gold patch.

Next, we represent each patch as a set of *patch features* $\tau(P) = \{(f, s, \ell) : f \in \text{files}(P),\, s \in \{+, -\},\, \ell \in \text{lines}(P)\}$, where ℓ is whitespace-stripped line content, and headers and blank lines are dropped. Features are sets, so repeated identical changed lines in the same file collapse intentionally. Patches with empty feature sets after filtering are excluded. We have used this simple representation in this work, and additional approaches such as AST-based (Alon et al., 2018; Guo et al., 2021) or semantic similarity (Moraglio et al., 2012; Silva et al., 2026) could be explored further.

Before de-duplication, we filter out agent's scratch work. Roughly 31% of raw mode features come from files that are neither present at the base commit nor in the gold patch, such as reproduction scripts, generated migrations, and build artifacts. Such features make otherwise correct modes impossible to match. We keep a feature only if its file is present at the base commit, appears in the gold patch, or is created at least 30% of the time across all runs. On SWE-Bench-Verified, the consensus clause is empty in practice, so the filter removes scratch features without dropping observed source fixes.

Divergence choices. Section 3 already gives the canonical solution distance as a recall-style divergence (Eq. 5); the implementation here just substitutes the empirical $\tilde{\mathcal{S}}_i$ for the unavailable $\mathcal{S}_i$. Writing $d_i(t) = \phi_i(x_i(t))$ for the source-projected live-diff feature set after replay step t (or $d(t)$ when the instance is clear), the per-reference recall fraction is

$$\text{recall}_P(t) \;=\; \frac{|\, d(t) \cap \tau(P) \,|}{|\, \tau(P) \,|}, \tag{9}$$

and Eq. 5 is evaluated empirically as $D(t) = 1 - \max_{P \in \tilde{\mathcal{S}}_i} \text{recall}_P(t)$.

A more symmetric alternative is a Jaccard distance to the nearest reference,

$$D_P^{\text{Jac}}(t) \;=\; 1 - \frac{|\, d(t) \cap \tau(P) \,|}{|\, d(t) \cup \tau(P) \,|}, \tag{10}$$

which penalises every live feature not in the reference as well as every missing one. We do not use it for live scoring because D^{Jac} penalizes harmless additions. If an agent reproduces all features of a verified repair but also adds m harmless features (comments, nearby cleanup, reproduction helpers, or unrelated files), recall reports 0 while Jaccard reports $m/(|\tau(P)| + m)$. From a practical viewpoint, such incidental features are common in the agent traces and difficult to filter without dropping real fix features. This is our primary motivation to keep recall for live-state scoring and sacrificing the theoretical *distance* property of Eq.10. As a reminder, we use Jaccard only when de-duplicating verified reference patches, where both operands are themselves candidate solutions. Finally, beyond these two options, one could plug in precision-style penalties on extra live features, edit-distance over patch text, or semantic divergences (AST alignment, embedding similarity, LLM-judged "fraction of fix achieved") and we leave those to future work.

Implementation specifics: Two details are worth mentioning. (1) $d(t)$ must be extracted with the *same diff algorithm* that produced the solution set elements, a unified diff (for example, `difflib.unified_diff(base, current)`), not a whole-file set difference. In an internal experiment, using set-diff inflated the residual end-distance by about 0.2, because a changed line whose text

recurs elsewhere in the file makes set-diff miss the corresponding '−' patch-feature while the git-diff mode has it. (2) Edit-step (via edit tool, shell command calls) paths must be normalized relative to repo workspace before feature extraction. Without this, models that issue absolute paths (e.g. `/testbed/django/. . .`) produce patch-features with file prefixes that never match the empirical solution relative paths, so D stays at 1.

B.1.4 Illustrative examples

We elaborate the recall-divergence (Eq. (5)) on five sample trajectories that span the qualitative shapes it should distinguish. (1) One decisive edit, (2) a gradual staircase with respect to (w.r.t.) a single element in solution set, (3) the same gradual descent w.r.t. complete solution set, (4) found-it-and-lost-it behavior, and (5) revert as A/B testing. Throughout, green tiles mark features the live diff has reproduced and gray tiles mark features still missing. Because $D = 1$ leaves the best-mode selection undefined (zero overlap with every reference in $\mathcal{S}_i$), we fix the displayed reference for any $D = 1$ panel to the benchmark gold patch as a visual tie-breaker. Note that, the agent may later reach a different oracle solution.

Example 1. Decisive single-edit fix. Figure 6 walks through the GPT-5.4 trajectory for `astropy__astropy-12907`, whose gold patch is a one-line replacement (two patch features: one deletion and one addition). A single edit at cycle position $t \approx 0.53$ reproduces both features simultaneously, so two snapshots fully characterise the run: the initial state (D=1) and the post-edit state (D=0). This is the simplest trajectory shape — one edit collapses D from 1 to 0 in a single step.

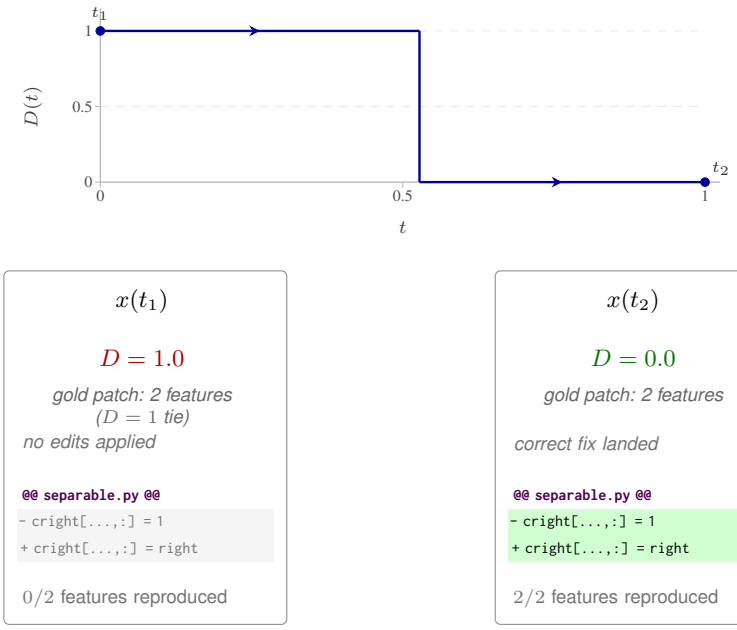
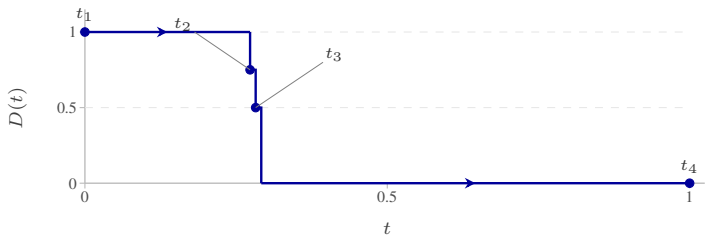


Figure 6: **Single-edit collapse.** A single correct edit can move an agent directly from the initial state to the solution set. Faithful replay of the GPT-5.4 trajectory on `astropy__astropy-12907` (SWE-Bench Verified) has only two relevant states: the initial repository state $x(t_1)$ before any edit (D=1) and the post-edit state $x(t_2)$ after the decisive edit at cycle position $t \approx 0.53$ (D=0). Green tiles mark reference patch features already present in the live diff; gray tiles mark features still absent. At t_1 every reference in $\tilde{\mathcal{S}}_i$ has zero overlap, so the displayed reference is fixed to the gold patch only as a visual tie-breaker. The curve therefore shows the cleanest possible trajectory shape: one edit moves the agent directly into the solution set.

Example 2. Gradual staircase against a single reference. Most runs descend through several transient non-zero D values before reaching the solution set. Figure 7 shows one such descent on the Opus 4.6 trajectory for `django__django-11292` (SWE-Bench-Verified), whose reference fix touches three hunks. The divergence is scored against a single 8-feature reference solution in $\mathcal{S}_i$, and D

records four non-zero values across the first three edits at normalized cycle positions $t = 0.27$, 0.28, 0.29, then holds at 0 through three refinements, each hunk landing in turn corresponds to one stair step.



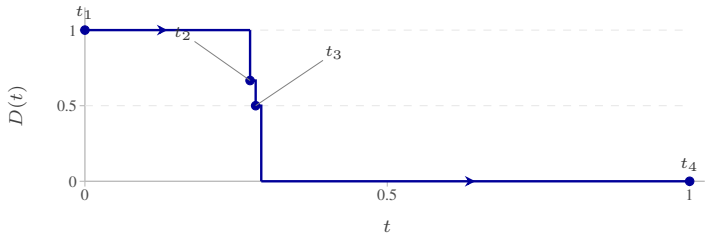
$x(t_1)$	$x(t_2)$	$x(t_3)$	$x(t_4)$
$D = 1.0$	$D = 0.75$	$D = 0.5$	$D = 0.0$

`@@ -95,7 +95,7 @@`
`- '...', '--force-color',`
`+ '...', '--skip-checks',`
`@@ -223,7 +223,7 @@`
`- ...=('skip_checks',...)`
`+ ...=('stderr','stdout')`
`@@ -286,6 +286,10 @@`
`+ parser.add_argument(`
`+ '--skip-checks',...`
`+ help='Skip ...',`
`+ )`
	`@@ -95,7 +95,7 @@`
	`- '...', '--force-color',`
	`+ '...', '--skip-checks',`
	`@@ -223,7 +223,7 @@`
	`- ...=('skip_checks',...)`
	`+ ...=('stderr','stdout')`
	`@@ -286,6 +286,10 @@`
	`+ parser.add_argument(`
	`+ '--skip-checks',...`
	`+ help='Skip ...',`
	`+ )`
		`@@ -95,7 +95,7 @@`
		`- '...', '--force-color',`
		`+ '...', '--skip-checks',`
		`@@ -223,7 +223,7 @@`
		`- ...=('skip_checks',...)`
		`+ ...=('stderr','stdout')`
		`@@ -286,6 +286,10 @@`
		`+ parser.add_argument(`
		`+ '--skip-checks',...`
		`+ help='Skip ...',`
		`+ )`
			`@@ -95,7 +95,7 @@`
			`- '...', '--force-color',`
			`+ '...', '--skip-checks',`
			`@@ -223,7 +223,7 @@`
			`- ...=('skip_checks',...)`
			`+ ...=('stderr','stdout')`
			`@@ -286,6 +286,10 @@`
			`+ parser.add_argument(`
			`+ '--skip-checks',...`
			`+ help='Skip ...',`
			`+ )`

Figure 7: **Fixed-reference staircase.** Against a fixed reference, solution distance behaves like recall over patch features. The Opus 4.6 trajectory for `django__django-11292` is scored against one 8-feature reference held constant across the four panels. Green tiles are reference features reproduced by the live repository state, and gray tiles are missing features. As the agent lands successive hunks, the matched count increases from $0/8$ to $2/8$, $4/8$, and finally $8/8$, producing the staircase $D{=}1 \to 0.75 \to 0.5 \to 0$. This fixed-reference view makes the recall-style construction concrete: distance decreases in proportion to the fraction of the target patch already committed to disk. The t_2 value is higher than the recorded trajectory value (≈ 0.67) because the actual metric maximises over all empirical reference modes, as shown in Figure 8.

Example 3. Same trajectory under the `max-over-references` selection. Figure 8 shows the same Opus 4.6 run (as in Example 2), but the displayed reference is now selected per step whenever the live state has nonzero overlap with an element in $\mathcal{S}_i$. Panel 1 is the $D = 1$ tie-breaker (gold patch), panel 2 scores against a popular 6-feature mode (22 of 104 resolved runs) which becomes closest in $\tilde{\mathcal{S}}_i$. Panels 3 and 4 score against the 8-feature three-hunk solution state shared with the Opus 4.5 runs as well (11 runs), which the live diff matches at $4/8$ after the second edit and $8/8$ after the third. Reported values $1.0 \to 0.667 \to 0.5 \to 0.0$ show the agent journey in the state space $\mathcal{T}_i$ towards $\mathcal{S}_i$ (or empirically approximated $\tilde{\mathcal{S}}_i$).

Example 4. Found-it-then-lost-it. We observed that not every trajectory that reaches $\tilde{\mathcal{S}}_i$ stays there till the end. Figure 9 shows a Gemini 3 Flash run on `django__django-14140` (SWE-Bench-Verified), where the target is `Q.deconstruct()` in `django/db/models/query_utils.py`. Edit #1 (normalized cycle $t = 0.15$) writes exactly the popular 12-feature minimal rewrite shared by 22 resolved runs and the agent reaches $\tilde{\mathcal{S}}_i$ ($D{=}0$). Edit #2 (cycle 0.46) reverts that rewrite and replaces it with a 7-line multi-condition guard that shares only $1/12$ features with that reference element, so D jumps to 0.917. Three further refinements (cycles $t = 0.63, 0.76, 0.80$) trim the guard but never restore the simple rewrite, and the run terminates at $D{=}0.9$, unresolved. Therefore, we should be aware of events where agent successfully reached the desired state (resolved in this case), but can backtrack



$x(t_1)$

$D = 1.0$

shown: gold patch
($D = 1$ tie)

```
@@ -95,7 +95,7 @@
- '...', '--force-color',
+ ..., '--skip-checks',
@@ -223,7 +223,7 @@
- ...=('skip_checks',...)
+ ...=('stderr','stdout')
@@ -286,6 +286,11 @@
+ if self.requires_...:
+ parser.add_argument(
+  '--skip-checks',...
+  help='Skip ...',
+ )
@@ -357,7 +362,7 @@
- if ... not options.get(...):
+ if ... not skip_checks:
```

$x(t_2)$

$D \approx 0.67$

best mode: 6 features
(popular, 22 runs)

```
@@ -95,7 +95,7 @@
- '...', '--force-color',
+ '...', '--skip-checks',
@@ -286,6 +286,10 @@
+ parser.add_argument(
+  '--skip-checks',...
+  help='Skip ...',
+ )
```

$x(t_3)$

$D = 0.5$

best mode: 8 features
(Opus 4.5 family, 11 runs)

```
@@ -95,7 +95,7 @@
- '...', '--force-color',
+ '...', '--skip-checks',
@@ -223,7 +223,7 @@
- ...=('skip_checks',...)
+ ...=('stderr','stdout')
@@ -286,6 +286,10 @@
+ parser.add_argument(
+  '--skip-checks',...
+  help='Skip ...',
+ )
```

$x(t_4)$

$D = 0.0$

best mode: 8 features
(Opus final)

```
@@ -95,7 +95,7 @@
- '...', '--force-color',
+ '...', '--skip-checks',
@@ -223,7 +223,7 @@
- ...=('skip_checks',...)
+ ...=('stderr','stdout')
@@ -286,6 +286,10 @@
+ parser.add_argument(
+  '--skip-checks',...
+  help='Skip ...',
+ )
```

Figure 8: **Max-over-modes view.** The full metric tracks the nearest empirical solution mode, not a single fixed patch. The same Opus 4.6 run on `django__django-11292` is re-rendered with the displayed reference chosen independently at each snapshot by the maximisation in Eq. 5. The trajectory descends $1 \to 0.667 \to 0.5 \to 0$ between cycles 0.273 and 0.292, then remains at zero through three later refinements. The best reference changes along the way: panel 1 uses the gold patch only as a zero-overlap tie-breaker, panel 2 selects a smaller 6-feature mode shared by 22 resolved runs, and panels 3–4 select an 8-feature mode shared with the Opus 4.5 cluster. This view explains why the empirical $D(t)$ values differ from the fixed-reference staircase in Figure 7: progress is measured against the closest element of $\tilde{\mathcal{S}}_i$ at each state.

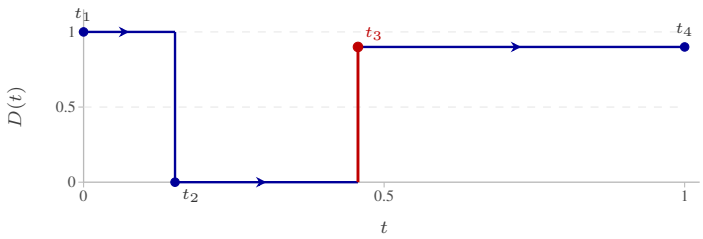
and reaches a wrong conclusion. Such dissection of model behavior is helpful to further improve the model behavior.

Example 5. Revert as discovery: a deliberate A/B test that uncovers a second bug. Not every backtrack is a mistake, and not every revert is a recovery from one. Figure 10 shows a Gemini 3.1 Pro trajectory whose middle event is a deliberate `git checkout`, a revert the agent uses as a controlled experiment. The revert surfaces a *second*, unrelated bug that the agent's first fix would otherwise have shipped past. The instance is `astropy-12907` (same as discussed in Example 1 for GPT 5.4), whose gold patch contains only one change in `astropy/modeling/separable.py` (see Figure 10 leftmost panel). The fix is enough to pass the hidden test suite. This Gemini 3.1 Pro trajectory, also identifies a second, unrelated bug in the same file: `_coord_matrix` calls `np.roll(mat, k)` without `axis=0`, which flattens before rolling and breaks compound models whose right child has multiple outputs.

Edit #1 (normalized cycle $t = 0.13$) is the obvious one: the agent replaces `cright[..] = 1` with `cright[..] = right` and its reproduction script passes. The benchmark gold solution is matched exactly, so D drops from 1 to 0. At this point the run is already oracle-correct and a submission here would pass the hidden tests.

Rather than submit, the agent constructs a sharper compound case, and notices that `separability_matrix` still returns a result that does not look block-separable. Edit #2 (cycle $t = 0.27$) is a `git checkout` on `separable.py`, and D jumps back to 1 (upward segment drawn red). The agent then walks into `_coord_matrix`, isolates the rolled coordinate matrix with a four-line probe, and observes that `np.roll(mat, k)` is flattening the 2-D array before rolling. Edit #3 (cycle 0.61) adds `, axis=0` to that call. The parseable repository state now contains the two `np.roll` features but is still missing the reverted `_cstack` features, which is later added at cycle t_5.

This is in contrast with Example 4 (Gemini 3 Flash on `django-14140`) which is a destructive backtrack, or the agent finds the solution, edits away from it, and never recovers. The present run is the opposite pattern, the revert is deliberate, and the post-revert trajectory introduces a new feature (`np.roll axis=0`) that the pre-revert state never contained. The resolved final state is strictly richer in patch features than the state at t_2 even though both achieve $D{=}0$. However, since the intended task is not regarding fixing the `np.roll(mat, k)`, this additional work has caused delayed distance reduction to reach the solution set $\mathcal{S}_i$. It is apparant from Figure 16 that $D(t)$ decays much slower for Gemini-3.1 Pro compared to counterparts (Opus 4.6, GPT 5.4).



$x(t_1)$

$D = 1.0$

shown: gold patch ($D = 1$ tie)

no edits applied

$0/12$ features reproduced

```
@@ -85,11 +85,7 @@
- args, kwargs = (), {}
- if len(c)==1 and
-   not isinst(c[0],Q):
-   child = c[0]
-   kwargs={c0:c1}
- else:
-   args=tuple(c)
-   kwargs={'_conn':...}
+ args = tuple(c)
+ kwargs = {}
+ kwargs['_conn']=conn
```

$x(t_2)$

$D = 0.0$

best mode: 12 features (popular, 22 runs)

edit #1: drop unpacking

$12/12$ features reproduced

```
@@ -85,11 +85,7 @@
- args, kwargs = (), {}
- if len(c)==1 and
-   not isinst(c[0],Q):
-   child = c[0]
-   kwargs={c0:c1}
- else:
-   args=tuple(c)
-   kwargs={'_conn':...}
+ args = tuple(c)
+ kwargs = {}
+ kwargs['_conn']=conn
```

$x(t_3)$

$D = 0.9$

best mode: 12 features (popular, 22 runs)

edit #2: revert + 7-line guard

$1/12$ features reproduced

```
@@ -85,1 +85,7 @@
- if len(c)==1 and
-   not isinst(c[0],Q):
+ if (
+   len(c)==1 and
+   not isinst(c[0],Q) and
+   hasattr(c[0],'__gi__')
+     and
+   not isinst(c[0],str) and
+   len(c[0]) == 2
+ ):
```

+7 extra non-reference features

$x(t_4)$

$D = 0.9$

best mode: 12 features (popular, 22 runs)

edits #3–#5: refine guard

$1/12$ features reproduced

```
@@ -85,1 +85,6 @@
- if len(c)==1 and
-   not isinst(c[0],Q):
+ if (
+   len(c)==1 and
+   not isinst(c[0],Q) and
+   isinst(c[0],tuple) and
+   len(c[0]) == 2
+ ):
```

final x: unresolved 'found-it-then-lost-it'

Figure 9: **Found-it-then-lost-it.** In this Gemini 3 Flash trajectory on `django__django-14140`, the first edit exactly reproduces a popular 12-feature rewrite in $\tilde{\mathcal{S}}_i$ (22 of $\sim$30 resolved runs, also the dataset gold), so D drops from 1 to 0 at cycle 0.15. The next edit reverts that rewrite and substitutes a multi-condition guard that overlaps only one reference feature, so D jumps back to 0.917 (rounded to 0.9); the upward segment in the curve is drawn red. Later edits refine the guard but never recover the matching rewrite, and the run finishes unresolved at D=0.9. The lower panels show the corresponding repository states before editing, after the correct edit, after the destructive edit, and at termination.

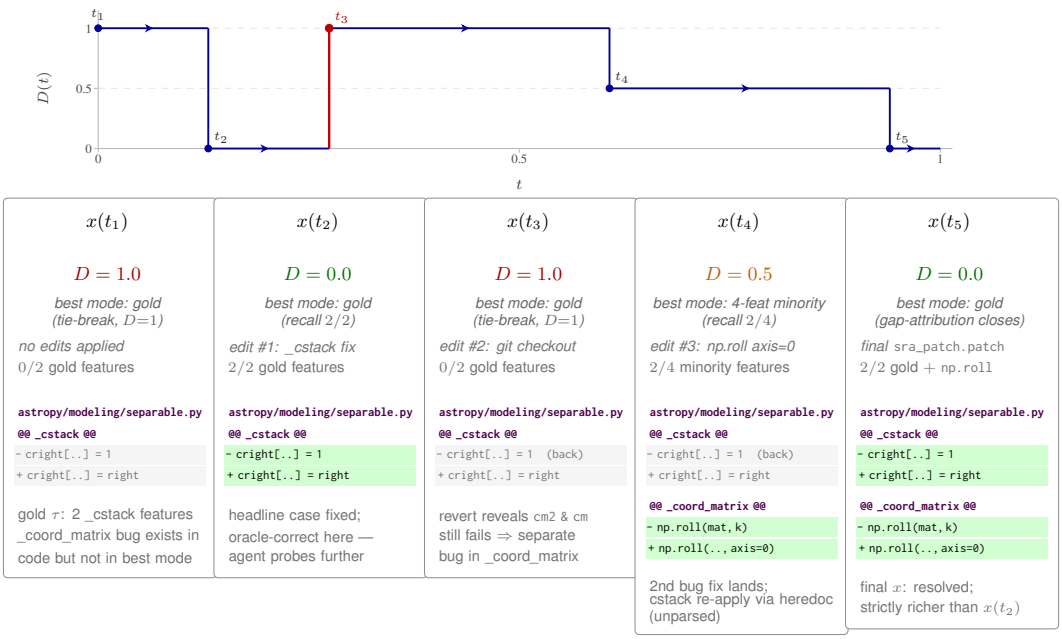


Figure 10: **Revert as discovery.** Gemini 3.1 Pro run on `astropy-12907` (SWE-Bench-Verified), the agent first reaches the gold-mode fix, deliberately reverts it to test the unfixed baseline, and uncovers an pre-existing unrelated bug. Faithful replay records the productive backtrack ($D: 1 \to 0 \to 1 \to 0.5 \to 0$): the final resolved patch combines the original `_cstack` fix with a new `np.roll(..., axis=0)` fix.

B.1.5 FAITHFUL STATE REPLAY

The empirical approximation of Eq. 5 requires the agent's actual repository state after each edit event. A simple heuristic of using a monotonic surrogate (e.g., credit each final-patch feature at the cycle position it first appears in any tool call's text) is computationally cheap but cannot increase and decrease in value. It cannot capture an agent that lands the fix and subsequently retracts it. Therefore, we reconstruct the state from scratch by parsing every edit call, applying it to in-memory file contents, and recomputing $d(t)$ at each step.

Cross-tool edit identification. In SSA, models edit the code through different tool surfaces. We map all of them to a unified operation list $[(f, \text{op}, \text{payload}), \ldots]$. Table 4 summarises the supported tool families and the corresponding parsers.

Tool / family	Models	Handled edit forms
`str_replace_editor`	Claude, Qwen, Gemini	`str_replace`, `create`, `insert`, `undo_edit`
`file_edit`	Grok	same shape as `str_replace_editor`
`file_write`	Grok	whole-file write
`apply_patch`	GPT-OSS	`Update File`, `Add File`, `Delete File`
`shell`, `bash`	GPT family and shell-tool users	`apply_patch` heredocs, `cat` heredoc overwrites. Git reverts handled below

Table 4: Edit-tool interfaces in SSA and the corresponding faithful-replay parsers.

Bash-based edit extraction. For shell-style tools, the replay parser scans the command text. For example, if a command contains one or more `apply_patch` heredocs, each heredoc body is parsed with the same `Update/Add/Delete File` parser used for the structured `apply_patch` tool. If the command contains `cat > file <<TAG ... TAG`, replay treats it as a whole-file overwrite of `file`. The bash edits identification is deliberately narrower than the behavioral edit classifier. The judge and path-auditor recognize broader bash mutations such as `sed -i`, `perl -pi`, and wrapper scripts that open a source file for writing. Faithful replay does not execute arbitrary shell or Python, so those commands are not applied as exact intermediate states. For resolved runs, any own-patch features still missing from the reconstructed state are treated as gap features. Replay assigns each missing features to the last parsed edit on its file, or, if no such edit exists, to the last bash command that textually mentioned that file. Those features are then added for distance evaluation after that cycle position. Thus, an un-parsed script edit can still reach $D(t_{\text{end}}) = 0$ through the self-anchor, but by attribution rather than by exact replay.

Fuzzy matching. The standalone editing tools in SSA, for example, `str_replace_editor` and `apply_patch`, tolerate whitespace and indent drift between the model's quoted context and the on-disk content. An exact-string applier in our faithful replay, therefore, under-applies edits that the real tool accepted. We first try an exact replacement. If that fails, `fuzzy_replace` normalizes each non-blank line by collapsing whitespace, then searches for a contiguous file-line window whose normalized non-blank lines equal the quoted old block. The same fallback is used inside `apply_patch` update hunks when the hunk's old block is not found exactly.

Revert handling. Agents undo edits through multiple channels and we track three revert categories: (1) `undo_edit` from the text-editor family (see Table 4), we maintain a per-file backup of pre-edit snapshots and pop on `undo_edit` call, (2) *file-scoped git reverts* parsed from shell/bash, such as `git checkout <file>` and `git restore <file>`, which reset the named file to its base contents, and (3) *tree-scoped git reverts*, `git reset --hard` and `git checkout .`, which resets every edited file to base state. Note, one pattern we deliberately exclude is *paired* `git stash` $\rightarrow$ `git stash pop`. This is a transient shelve-to-test maneuver (i.e., run tests on a clean tree and then restore), and hence we ignore counting its momentary dip as backtracking. A simple audit of the trajectories shows it is essentially always paired, so the final state is unaffected and the dip is purely an artefact of the intermediate state.

B.1.6 RESOLVED ENDPOINTS: SELF-ANCHOR

A resolved run for an instance r is in the solution space by construction as its final submitted patch passed the oracle. Therefore, the correct endpoint for the resolved trajectory is $D(t_{\text{end}}) = 0$. We tested strict leave-one-run-out (LOO) scoring as an intuitive diagnostic: score a run only against solutions independently observed in other runs. This did not work for our reported trajectory curves. The empirical solution set $\tilde{\mathcal{S}}_r$ is a finite sample, not a dense cover of the true solution space. Nearby correct solutions can differ in variable names, hunk boundaries, guard placement, helper structure, or surrounding context, and the nearest textual variant may simply be absent from the other observed runs. In practice, LOO left some resolved trajectories with $D(t_{\text{end}}) > 0$, which disturbed the average resolved $D(t)$ curve even though those runs solved their instances.

For the trajectory curves we therefore use a self-anchor, or any resolved run r has its corresponding generated patch alwsys in the empirical solution set $\tilde{\mathcal{S}}_r$. Unresolved runs receive no self-anchor, since their final patches are not oracle-verified solutions.

Note that this is not an extra credit for solved runs, it is the endpoint-consistency condition implied by the definition of the solution space. Hence, the nearest-reference recall at the endpoint is 1, and $D_r(t_{\text{end}}) = 0$ by construction. Without self-anchor, a positive endpoint distance can reflect sparsity in the empirical reference set $\tilde{\mathcal{S}}_r$ rather than a real failure to reach the solution set $\mathcal{S}_i$.

B.1.7 BACKTRACKING QUANTIFICATION

The main-text quantity $\bar{B}(t)$ is an instantaneous *backtracking depth*. Writing the corresponding per-run depth as $B_r(t) := D_r(t) - M_r(t)$ gives $\bar{B}(t) = N^{-1} \sum_r B_r(t)$. Interestingly, faithful replay also supports a complementary local diagnostic. For successive successful edit events at t_k and t_{k+1}, define

$$\Delta D_{r,k} := D_r(t_{k+1}) - D_r(t_k), \qquad k = 1, 2, \ldots. \tag{11}$$

The two notions of Eq.(6) and Eq.(11) are linked exactly by

$$B_r(t_{k+1}) = \big[B_r(t_k) + \Delta D_{r,k}\big]_+, \qquad \text{where, } [z]_+ := \max\{z, 0\}. \tag{12}$$

Thus, $\Delta D_{r,k} > 0$ is a *local backtracking event* that increases the depth, while a negative change reduces the depth and may leave it positive while the run recovers. Accordingly, $\bar{B}(t)$ measures current unrecovered depth, while the following summaries measure the frequency and magnitude of local uphill edits.

Backtrack fraction the share of trajectories for which some $\Delta D_{r,k} > 0$.

Backtrack-edit fraction $\#\{(r,k) : \Delta D_{r,k} > 0\} / \#\{(r,k) : |\Delta D_{r,k}| > 0\}$.

Uphill mass $\sum_{r,k} [\Delta D_{r,k}]_+ / \sum_{r,k} |\Delta D_{r,k}|$.

The first counts trajectories that contain at least one local backtracking event, while the second counts such events among distance-moving edits. The third weights them by size, so one large reversal contributes as much as many small ones. Per-model values are reported with the ΔD histograms in Figures 12 and 13 (green = progress edits, red = local backtracking events, annotated percentage = backtrack-edit fraction).

B.1.8 LIMITATIONS OF THE SOLUTION-DISTANCE METRIC

The recall-divergence metric (Eq. 5) we have described is deliberately structural rather than semantic, and inherits several limitations that we now list below.

1. **Textual overlap, not semantics.** The recall fraction in Eq. 9 counts matching changed lines. A correct-but-textually-different fix, i.e., a different identifier choice, a refactored expression, a guard placed elsewhere scores < 1 against reference modes it does not textually match. The empirical subsets and the self-anchor for resolved endpoints mitigate this substantially, but the underlying limitation remains. For unresolved runs and intermediate states, a semantic distance (AST alignment, embedding similarity, or an LLM-judge "fraction-of-fix-achieved") would score the agent's intent rather than its output text.

2. **Empirical subset $\neq$ solution space.** The space is defined by the test oracle and $\tilde{\mathcal{S}}_i$ approximates it with the patches we happened to observe (using 21 models run 5 times). A unique correct solution found by nobody else, a real possibility on novel instances, can be far from the observed empirical modes, which is why the self-anchor is reserved for resolved runs whose own patches are oracle-verified.
3. **Reference-set sparsity scales with difficulty.** $\tilde{\mathcal{S}}_i$ is dense on easy instances (for example, in SWE-Bench-Verified $80+$ reference modes) and sparse on hard ones. Instances that no model in our sweep solved have only the gold patch as a reference (or no reference if gold is also missing). D on these instances reads against a thin reference set and should be interpreted with care.
4. **Replay fidelity.** $d(t)$ is reconstructed from the edit-tool calls we parse (Table 4). Therefore, exotic shell rewrites (e.g. Python scripts that open a source file in write mode) are not fully parsed. Self-anchor fixes the resolved endpoint, but it cannot recover the exact intermediate state of an unparsed edit, so a small number of trajectories' mid-run shapes remain approximate.

B.2 METRICS

This subsection catalogs every per-trajectory and per-cycle metric either reported directly by SSA or extracted via the LLM judge or simple computations. Table 5 gives a symbol or short name used elsewhere in this report, a one-line definition, and the figure(s) in which it is plotted.

Table 5: Metrics glossary. Every per-trajectory / per-cycle quantity reported by SSA and the figure(s) in which it appears. "Judge R#" refers to the rubric field defined in Table 3.

Metric	Definition	Figures
pass@1	Mean resolution rate across the n runs $\times$ k instances per (model, benchmark); the fraction whose final state passes the benchmark's tests.	Table 1; Appendix C
Total output tokens	Sum of `output_tokens` across all $(run, instance)$ pairs for a given (model, benchmark)	Figures 11, 24, 37
$D(t)$**, solution distance**	Main-text recall divergence (Eq. 5) evaluated on the empirical solution subset $\tilde{\mathcal{S}}_i$ using Eq. 9. Distance is from the live state to the nearest verified-correct reference mode, with self-anchor for resolved endpoints.	Figures 16, 17, 29, 30
$\Delta D_{r,k}$**, local change**	$D_r(t_{k+1}) - D_r(t_k)$ for successive successful edit events. A negative change is progress; a positive change is a local backtracking event.	Figures 12, 13, 25, 26
Backtrack-edit fraction	Share of distance-moving edits with $\Delta D_{r,k} > 10^{-9}$; changes with $\lvert\Delta D_{r,k}\rvert \leq 10^{-9}$ are excluded.	Figures 12, 13, 25, 26 (annotated)
Backtrack fraction	Share of trajectories with any $\Delta D_{r,k} > 10^{-9}$, i.e., how often trajectories contain at least one local backtracking event.	cohort summary (text, §B.3)
Uphill mass	$\sum_{r,k}[\Delta D_{r,k}]_+ / \sum_{r,k} \lvert\Delta D_{r,k}\rvert$, local backtracking weighted by step size.	cohort summary (§B.3)
Edit/test ratio	Per-cycle ratio of edit-oriented calls (`edit_source`, `edit_test`, `write_scratch`) to read-oriented calls (`read_code`, `search_locate`, `explore_navigate`); aggregated as a distribution per model.	Figures 14, 15, 27, 28, 42, 43
Genuine tool-error rate	Per-cycle share of calls with R8 = `genuine`, broken out by tool family (`bash` vs. `str_replace_editor`); excludes benign non-zero exits (`grep` no-match, `pytest` reporting test failures, etc.).	Figures 18, 19, 31, 32, 40, 41
Phase composition	Share of cycles in each R6 phase (`explore`, `localize`, `implement`, `verify`, `other`); stacked-area over normalised cycle position.	Figures 20, 21, 33, 34, 38, 39

continued on next page

Table 5 – continued

Metric	Definition	Figures
Token economics	Per-cycle `input_tokens`, `output_tokens`, `reasoning_tokens`, `cache_read_input_tokens`, `cache_write_input_tokens`, plus `cache_hit_rate`.	cost / cache analyses
Latency	`ttft_ms` (time-to-first-token) and `cycle_durations_sec`; aggregated into `execution_time_seconds`.	cost analyses
Per-tool counters	`call_count`, `success_count`, `error_count`, `total_time`, `average_time`, derived `success_rate`; measured per tool name.	Figures 22, 23, 35, 36, 44, 45

The metrics fall into three families. **(i) Outcome and cost** (pass@1, total output tokens) describe what the run achieved and what it cost. They come directly from the harness and the benchmark-specific evaluator. **(ii) Trajectory geometry** ($D(t)$ and the ΔD-derived scalars) describe how the live repository state evolved over time relative to the empirical solution subset. They require the faithful-replay pipeline of Sections B.1.5–B.1.6. **(iii) Behavioral composition** (edit ratio, genuine tool-error rate, phase composition) describe *what* the agent was doing inside each cycle. They require the per-call LLM judge of Section B.1.1.

Definition of pass@k. We follow the unbiased combinatorial estimator introduced by Chen et al. (2021). Let n be the number of independent samples generated per instance and let $c \leq n$ be the number of those samples that pass the oracle. Then

$$\text{pass@k} \;=\; \mathbb{E}_{\text{instances}}\left[\,1 \;-\; \frac{\binom{n-c}{k}}{\binom{n}{k}}\,\right], \tag{13}$$

where the expectation is the mean over instances in the benchmark and $\binom{n-c}{k} = 0$ whenever $n-c < k$. The estimator is unbiased and is numerically stable to compute as $1 - \prod_{i=0}^{k-1}(n-c-i)/(n-i)$ when $n - c \geq k$. We report pass@1 in Table 1 and Appendix C, with 95% confidence intervals computed across the $5 \times |\text{instances}|$ trials per (model, benchmark). For pass@2 we use $n = 5$ runs per instance, so an instance contributes $1 - \binom{5-c}{2}/\binom{5}{2}$ for $c \in \{0, 1, \ldots, 5\}$ correct samples, then average over instances.

Sections B.3–B.5 below break the metrics down per benchmark, each in its own subsection. Plots are computed by aggregating the per-cycle counters above across the five runs we collect per (model, benchmark), then averaging at the model level (token totals are summed across all $(run, instance)$ pairs.

B.3 Metrics: SWE-Bench-Verified

Figure 11 relates total output-token spend to pass@1 across every model we evaluate, giving a rough Pareto view of token cost vs. resolution rate. Top-left is the desirable region (high accuracy with the fewest output tokens spent reasoning and acting).

The next four figures show per-model behavioral metrics across all 21 models we evaluated on SWE-Bench-Verified, ordered by family so adjacent cells share lineage. Anthropic Claude (Opus 4.6, Opus 4.5, Sonnet 4.6, Sonnet 4.5, Sonnet 4.0, Haiku 4.5), OpenAI GPTs (GPT-5.4, GPT-5.3 Codex, GPT-5.2 Codex, GPT-5.2), OpenAI open-weights (GPT-OSS 120B, GPT-OSS 20B), Google Gemini (3.1 Pro, 3 Flash), xAI (Grok 4.20-reasoning), then Qwen3-Coder variants (Next, 480B, 30B) and Qwen3.5/3.6 think variants (35B-A3B, 27B, Qwen3.5-27B).

Table 6 gives a compact numeric summary of the trajectory-geometry metrics (solution distance and local backtracking) over the resolved trajectories of each model, alongside the outcome metric pass@1. Three trajectory scalars: **local backtrack %** (share of distance-moving edits with $\Delta D > 0$ — low value means a monotone descent), **Arrival-time t@$D \leq 0.1$** (the normalized cycle position at which the mean resolved $D(t)$ curve first reaches within 0.1 of a solution — high value means the

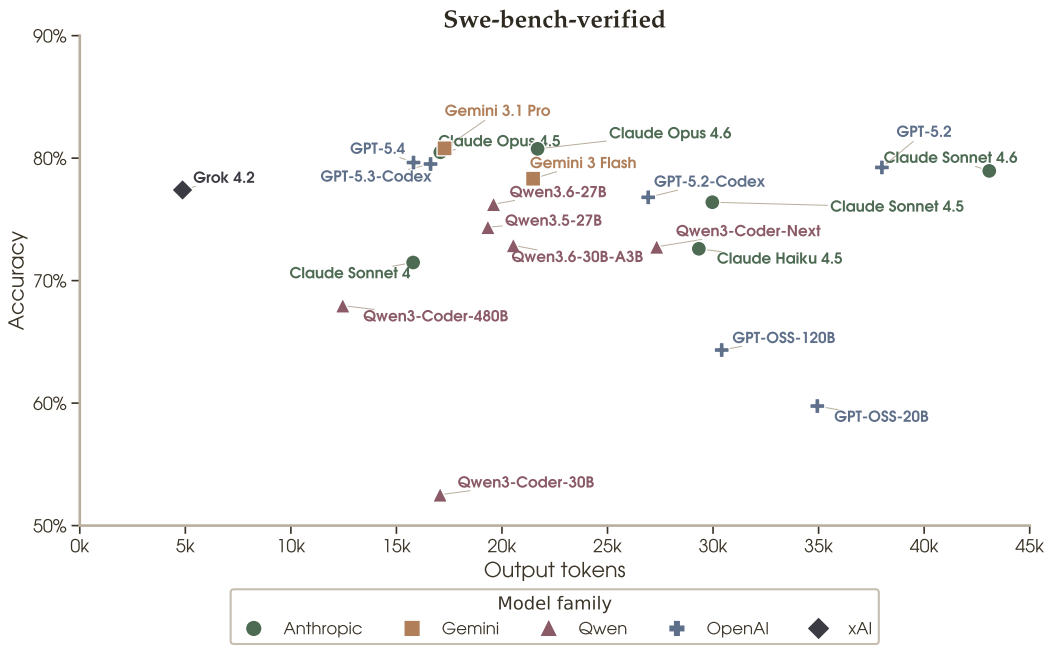


Figure 11: SWE-Bench-Verified: total output tokens vs. pass@1, one point per model. Output tokens are for each problem instance (summed over all LLM calls in an agent run), averaged across 500 instances and 5 runs per instance . Pass@1 is the mean of the five per-run resolution rates. Colors group models by family. Top-left corresponds to high accuracy with the fewest output tokens spent reasoning and acting.

model lands its solution-relevant edits late in the run), and **AUC** (area under the resolved $D(t)$ curve, lower means the live diff was, on average, closer to a reference solution throughout the run). The full outcome table including pass@2 with 95% confidence intervals is in Appendix C.

Three salient observations from Table 6 carry through the per-model grids below. The **Backtracking-edit** (the share of distance-moving edits with $\Delta D > 0$) spans an order of magnitude across the 24 models. The GPT-5 family and GPT-OSS sit under 3% (GPT-5.2-codex lowest at 0.9%), the Claude models at 4–10%, Grok 4.2 at 9.6%, Gemini 3.1 Pro at 14.2% and Gemini 3 Flash at 16.9%. The backtrack fraction (the share of trajectories that backtrack at least once) tracks this ranking closely (Gemini 3 Flash at 36%; GPT-5.2-codex at 2%). Crucially, backtracking is roughly *independent* of solve rate: Gemini 3.1 Pro backtracks more than any model except Gemini 3 Flash (14.2%) yet ties Opus 4.6 (9.4%) at 80.8% pass@1 on SWE-Bench-Verified, while GPT-5.4 is among the least-backtracked (1.3%) and sits within a point of both (80.2%). This decoupling reveals need for measuring trajectory *shape* alongside the outcome, or, *two models with the same pass@1 can have radically different confidence profiles*. For **Arrival-time** (t @ $D \leq 0.1$), Grok 4.2 (0.95) and GPT-5.2-codex / GPT-OSS 20B (0.89) push their solution-relevant edits the latest, while Opus 5.0 (0.37) and then Opus 4.8, Sonnet 5.0 and Sonnet 4.5 (0.47) converge the earliest. **AUC** (average distance) is lowest for Opus 5.0 (0.25), Sonnet 5.0 (0.27) and Sonnet 4.5 (0.28) and highest for GPT-5.2-codex (0.68), GPT-5.4 (0.63) and Opus 4.6 (0.58). This is consistent with the arrival-time signal, a model that converges late spends most of the cycle far from any solution, regardless of whether its final state is correct.

Independent corroboration via the judge. The metrics above are computed entirely from the reconstructed file state. We cross-check it against an orthogonal signal derived from the tool-call judge, the deterministic *revert label* `is_revert`. This label tags each tool-call as `undo_edit` / `git_revert_file` / `git_revert_all` / `none`. Paired `git stash` → `git stash pop` calls are excluded as transient. The judge-derived revert/edit rate (calls labeled revert divided by `edit_source` + `edit_test`) tracks the faithful-replay backtracking ranking almost exactly. Gemini-3.1-Pro scores 14.0%, Opus 4.6 scores 8.7%, Sonnet 4.6 scores 6.1% and GPT/Codex sit near 0. Overall, two

Table 6: Distance, local backtracking and outcome summary over resolved SWE-Bench-Verified trajectories per model. Local backtrack %, Arrival-time t @ $D \leq 0.1$ and area-under-curve (AUC) are computed from the faithful state-replay; pass@1 is from Appendix C.

Model	Local backtrack %	t @ $D \leq 0.1$	AUC	pass@1 %
Opus 5.0	9.0	0.37	0.25	96.60
Opus 4.8	9.3	0.47	0.31	88.80
Opus 4.6	9.4	0.79	0.58	80.80
Opus 4.5	7.5	0.58	0.41	80.48
Sonnet 5.0	9.9	0.47	0.27	84.04
Sonnet 4.6	5.4	0.74	0.50	79.00
Sonnet 4.5	6.1	0.47	0.28	76.40
Sonnet 4.0	6.2	0.68	0.41	72.00
Haiku 4.5	4.2	0.58	0.35	72.60
GPT-5.4	1.3	0.84	0.63	80.16
GPT-5.3	1.6	0.79	0.56	79.52
GPT-5.2	1.2	0.79	0.54	79.28
GPT-5.2-codex	0.9	0.89	0.68	76.80
GPT-OSS 120B	2.8	0.84	0.42	64.64
GPT-OSS 20B	1.2	0.89	0.54	59.84
Gemini 3.1 Pro	14.2	0.89	0.43	80.80
Gemini 3 Flash	16.9	0.79	0.42	78.32
Grok 4.2	9.6	0.95	0.57	77.56
Qwen3-Coder Next	5.6	0.68	0.44	72.44
Qwen3-Coder 480B	6.5	0.68	0.44	67.56
Qwen3-Coder 30B	4.2	0.68	0.45	51.36
Qwen3.6 35B-A3B	5.0	0.63	0.40	72.92
Qwen3.6 27B	5.0	0.58	0.37	76.24
Qwen3.5 27B	6.4	0.63	0.38	74.36

independent methods, (i) full state reconstruction via $\Delta D > 0$ and (ii) syntactic call labeling, agree both on which models backtrack and on how frequently.

Figure 12 shows the per-edit ΔD histogram for each model: blue bars are progress edits ($\Delta D < 0$, agent moved towards a reference mode), red bars are backtracking edits ($\Delta D > 0$, agent moved away). The annotated percentage is the backtrack-edit share. Decisive solvers (GPT-5 family) have near-empty red mass, Qwen3-coder and Gemini-3.1-Pro carry the heaviest tails.

Per-model edit/test ratio. Figure 14 shows the per-cycle ratio of edit-oriented tool calls to total ones across resolved + unresolved trajectories. A high ratio means the agent mostly edits and a low ratio means it mostly explores/test.

Per-model level families. Figures 16–17 show, for each model, the level family of its resolved trajectories together with the mean $\bar{D}(t)$ and the backtracking term $\bar{B}(t)$ of Section 3. Resolved curves are self-anchored to terminate at 0 (Section B.1.6); the dark-red curve is the unresolved mean.

Per-model genuine tool-error rate over cycles. Figure 18 plots the rate of *genuine* tool errors per cycle (R8 = `genuine`, see Section B.1.2), broken out by tool family (`bash` vs. `editor`). Benign non-zero exits (`grep` no-match, `pytest` reporting test failures, etc.) are excluded so spikes in this metric are real harness friction and not test outcomes. The standout is Grok 4.2, whose `file_write` errors run near 44%: a tool-contract mismatch rather than a reasoning failure, since `file_write` is create-only and confined to the workspace. Scratch scripts written to `/tmp/` (74%) and re-writes of existing files instead of `file_edit` (26%) account for all of it, and both are recoverable on retry.

Per-model phase composition. Figures 20–21 split each trajectory across the four R6 phases (Section 4.3). GPT-5 trajectories collapse explore/localize into implement/verify, while Qwen3-coder and Gemini spread time more evenly.

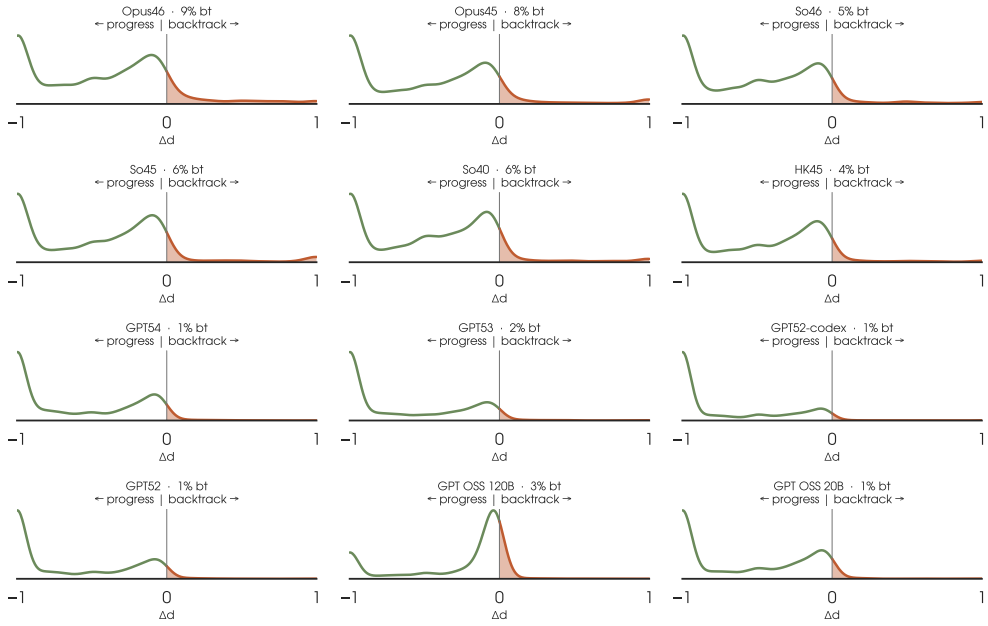


Figure 12: **Local backtracking.** Histograms of ΔD on SWE-Bench-Verified (part 1 of 2): Anthropic Claude and OpenAI families. Each cell is one model, green = progress edits, red = local backtracking edits. Annotation shows the per-model local-backtrack share.

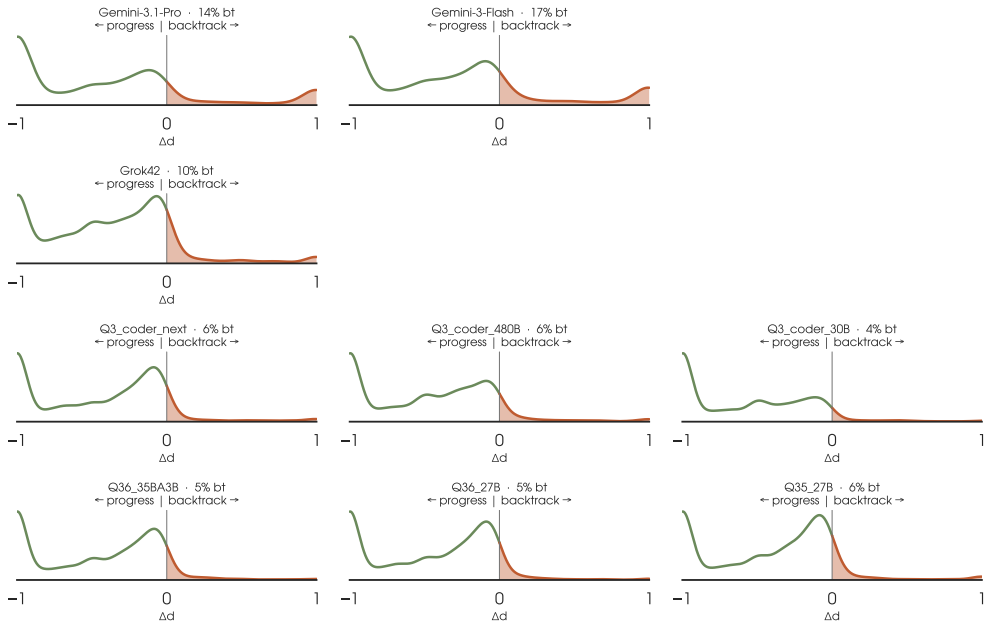


Figure 13: **Local backtracking.** Histograms of ΔD on SWE-Bench-Verified (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 12.

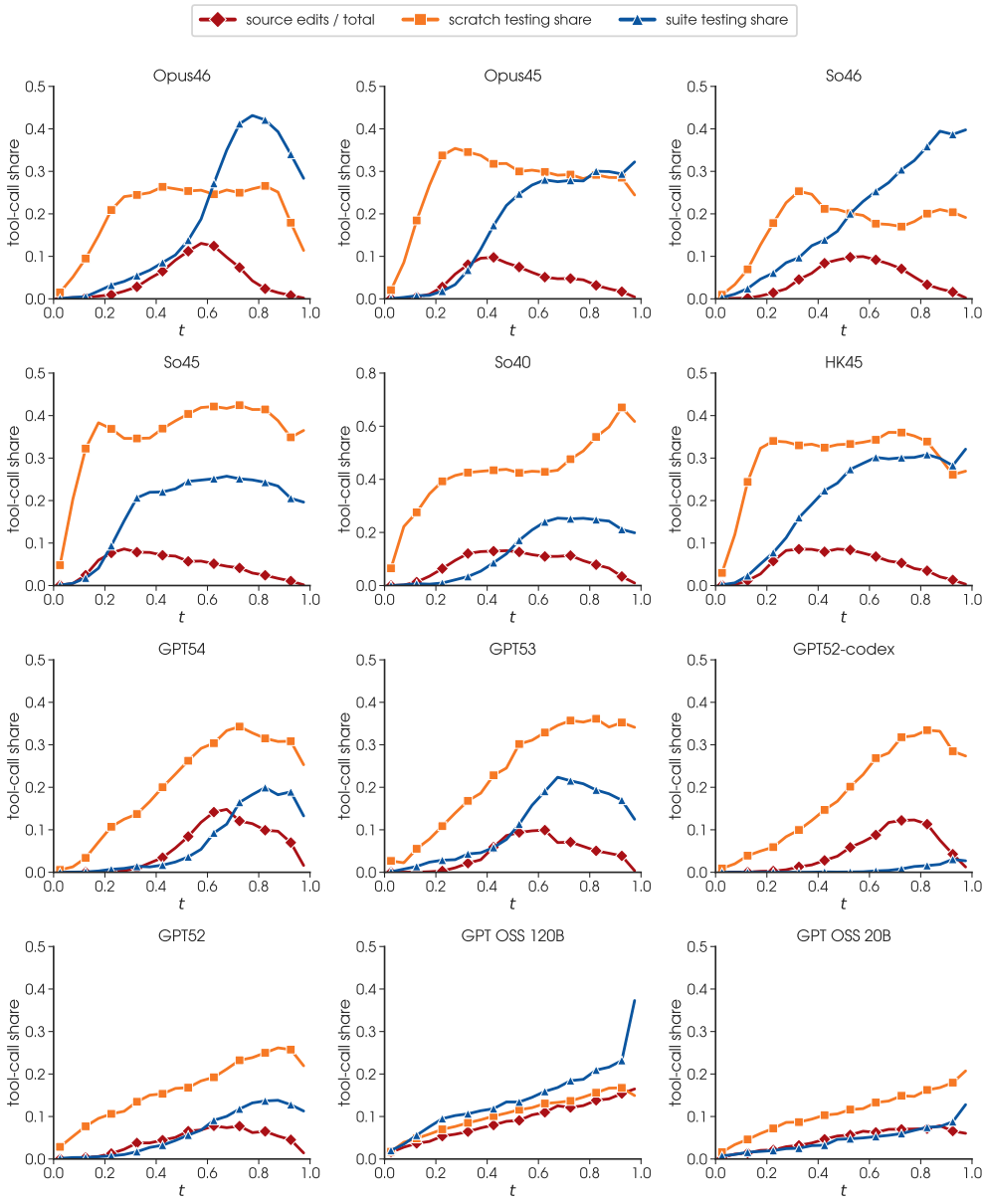


Figure 14: Edit/test ratio per cycle (part 1 of 2): Anthropic Claude and OpenAI families. Each cell shows the model's distribution of ratios.

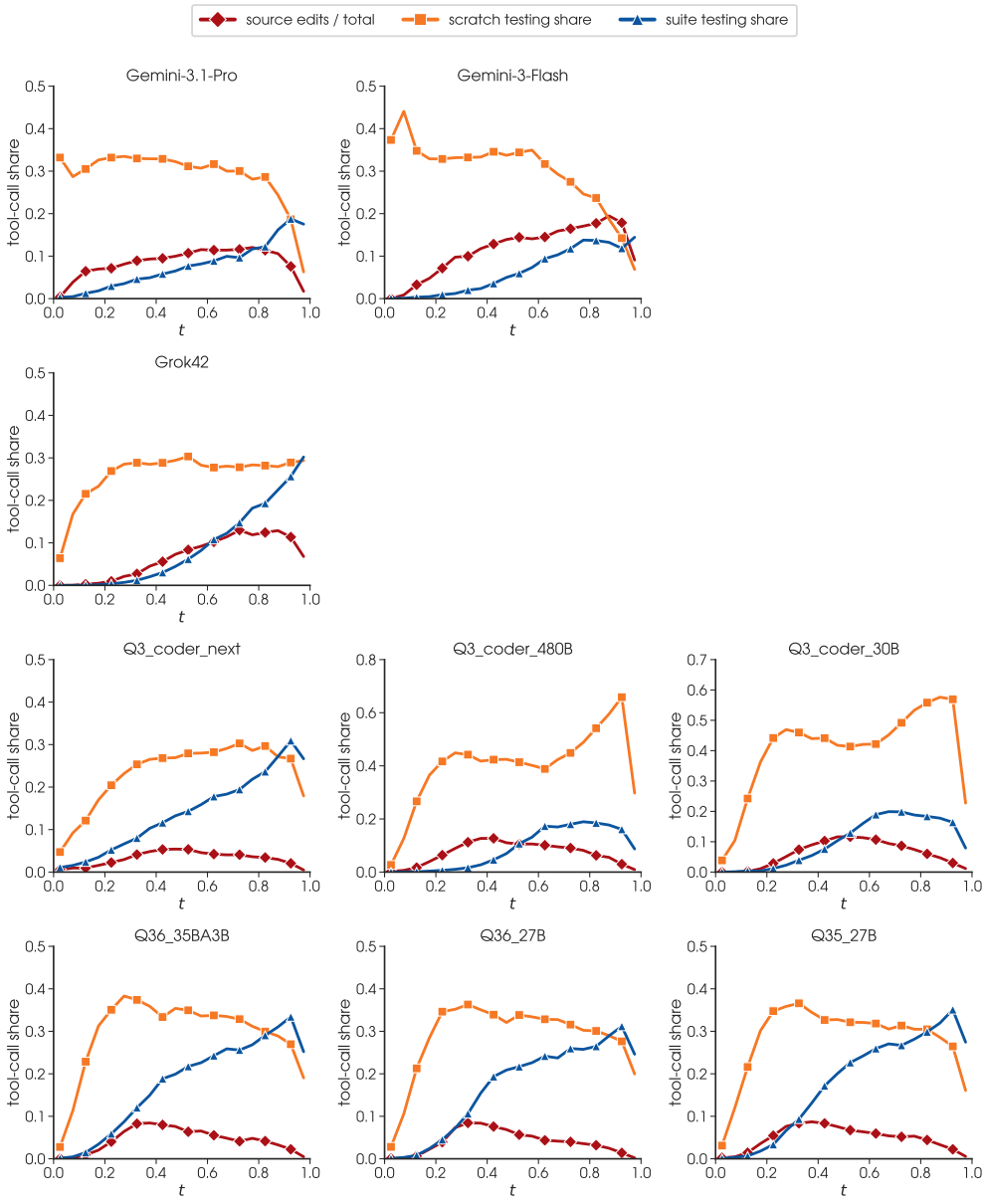


Figure 15: Edit/test ratio per cycle (part 2 of 2). Gemini, Grok and Qwen families. Continued from Figure 14.

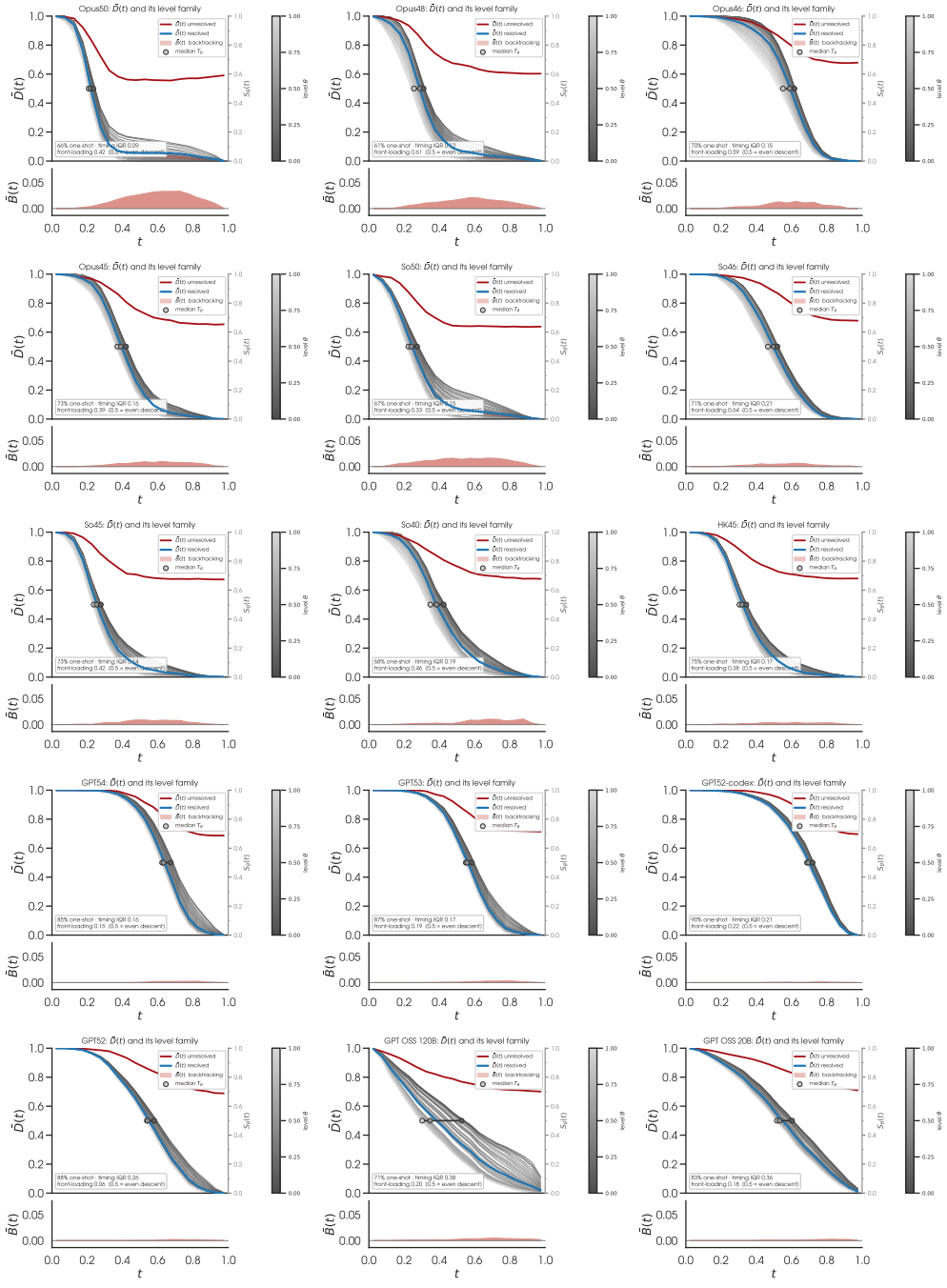


Figure 16: Level families on SWE-Bench-Verified (part 1 of 2): Claude and GPT families. Grey curves are the level sets $S_\theta(t)$ of the resolved trajectories (Section 3), the blue curve is their mean $\bar{D}(t)$, the dark-red curve is the mean $\bar{D}(t)$ of the unresolved runs (which plateaus where they stop), the markers give the median crossing time T_θ (the median over runs of the first-entry time $T_r(\theta)$, i.e. where S_θ crosses $1/2$) at $\theta = 0.5$ with its interquartile range, and the strip beneath each panel is the backtracking term $\bar{B}(t)$. The inset reports the one-shot share (runs that land the patch in a single edit); the timing IQR, the interquartile range of the runs' half-way times $T_r(0.5)$, i.e. how dispersed solve timing is across runs; and the front-loading index $(T_{0.5} - T_{0.9})/(T_{0.1} - T_{0.9})$, the position of the median half-way time between the median crossing times of levels 0.9 and 0.1 (0.5 = even descent; lower = most of the distance falls early).

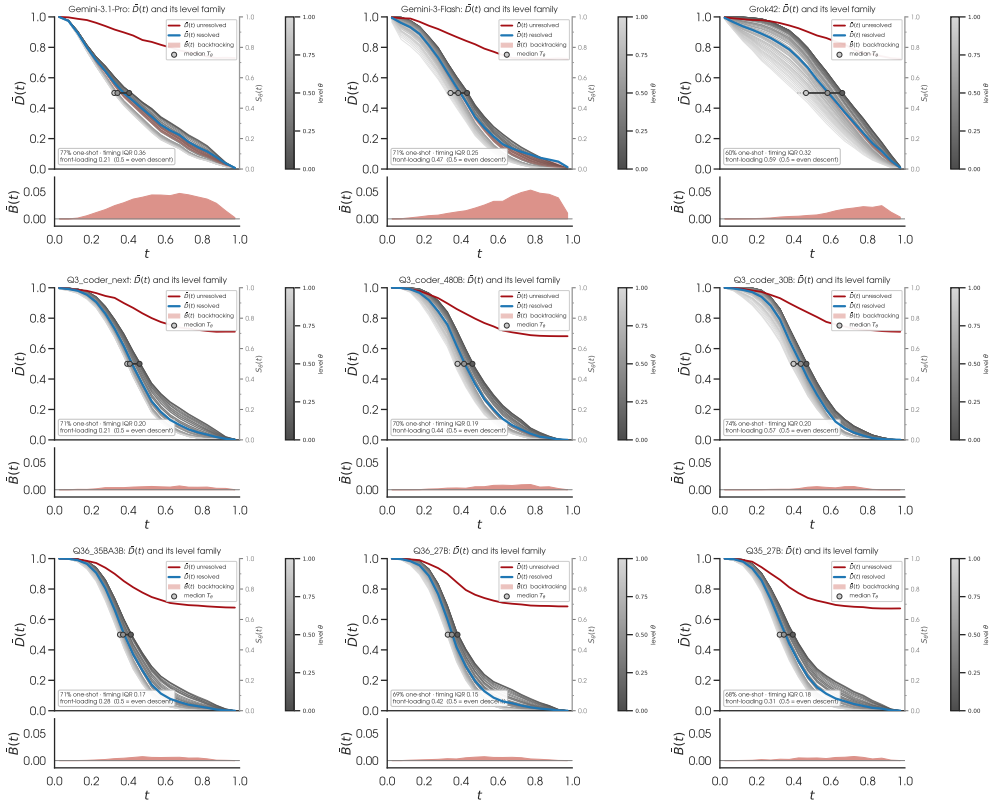


Figure 17: Level families on SWE-Bench-Verified (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 16.

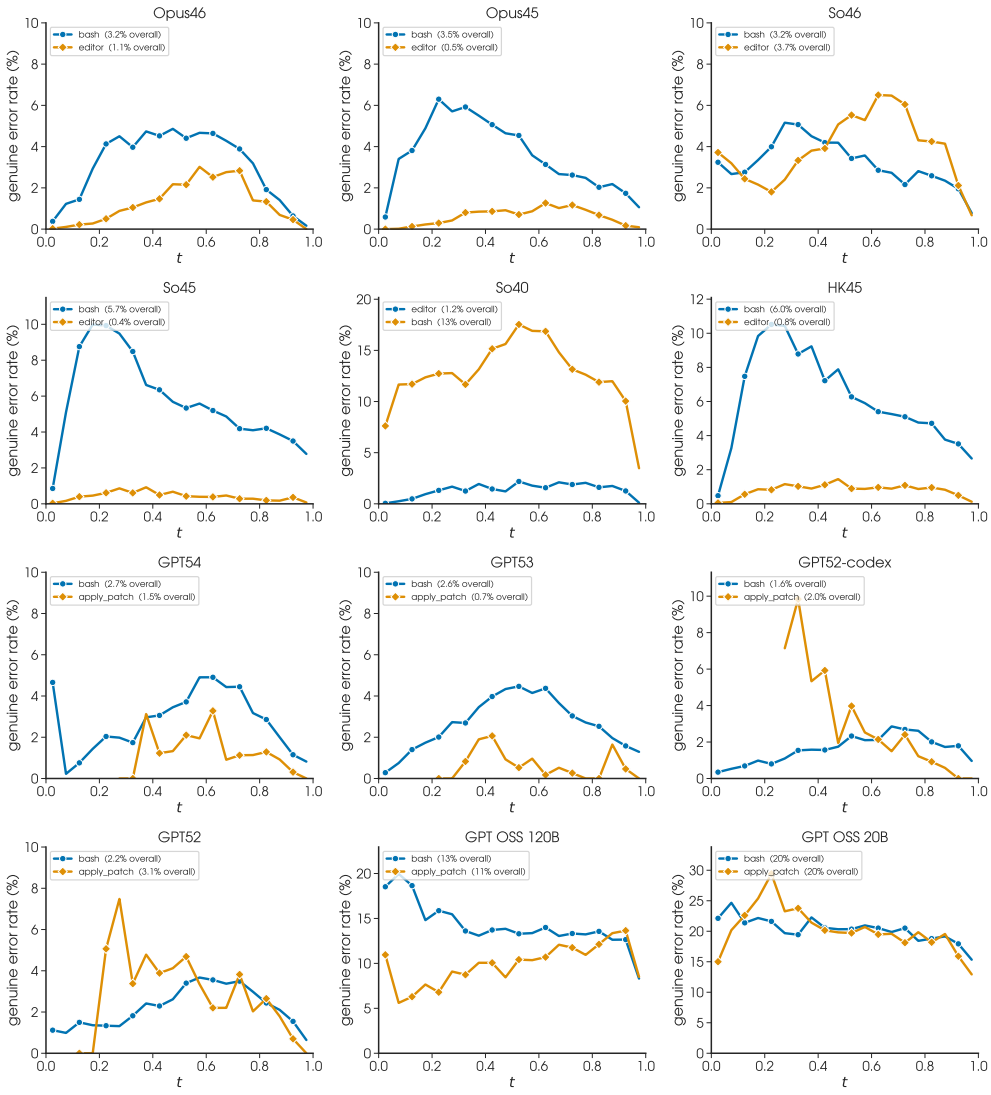


Figure 18: Genuine tool-error rate per cycle (part 1 of 2): Anthropic Claude and OpenAI families. Benign exits (`grep` no-match, `pytest` test failures, etc.) are excluded. Tool families broken out within each sub-plot.

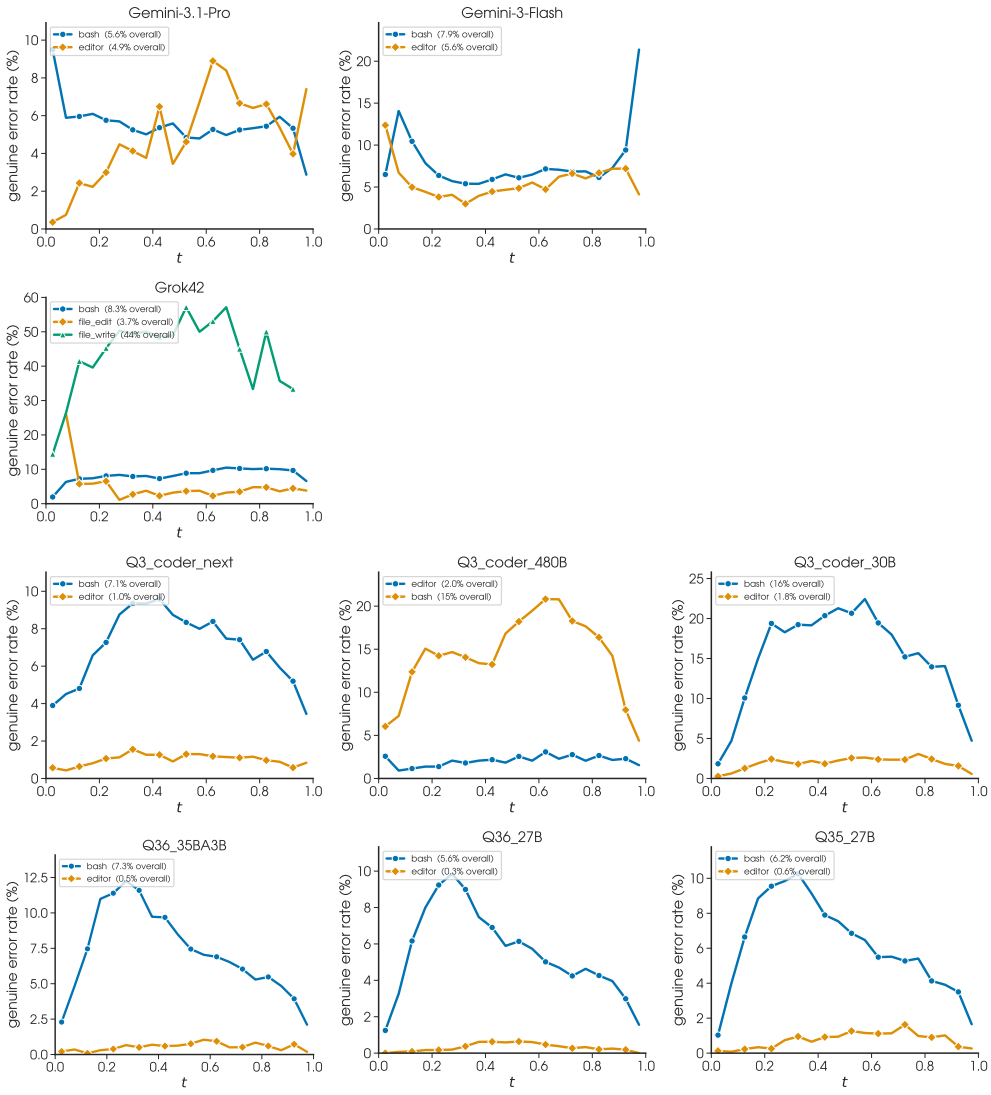


Figure 19: Genuine tool-error rate per cycle (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 18.

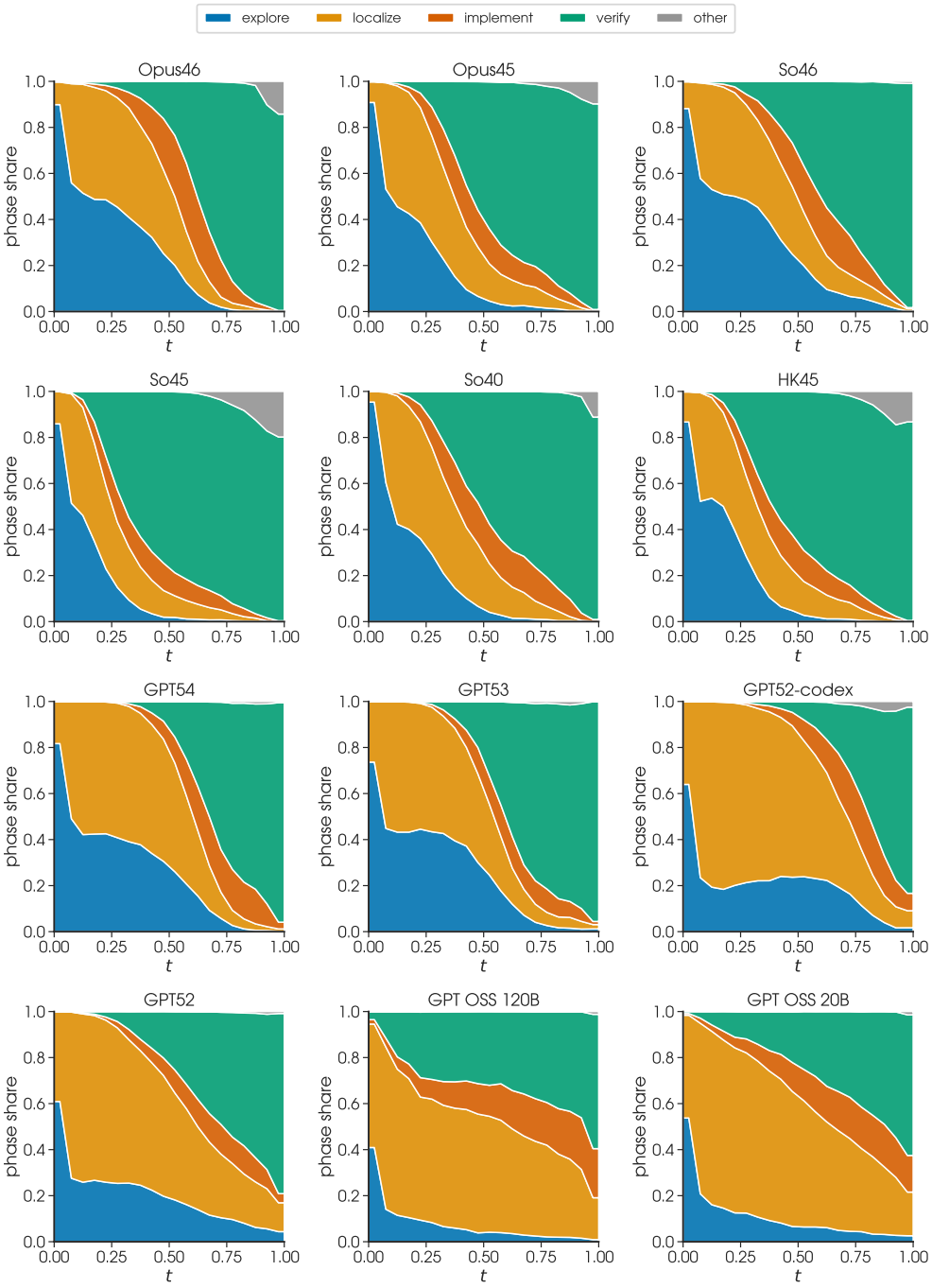


Figure 20: **Phase composition surfaces model-specific schedules.** Per-cycle share of the four R6 (see LLM judge rubrics in Table 3) phases (`explore`, `localize`, `implement`, `verify`) vs. normalised cycle position t on SWE-Bench-Verified, for Claude and GPT families (part 1 of 2). Given the same tasks, models schedule their cycles differently. The implementation peak shifts in position and width, and the verify phase carries different long-tail mass.

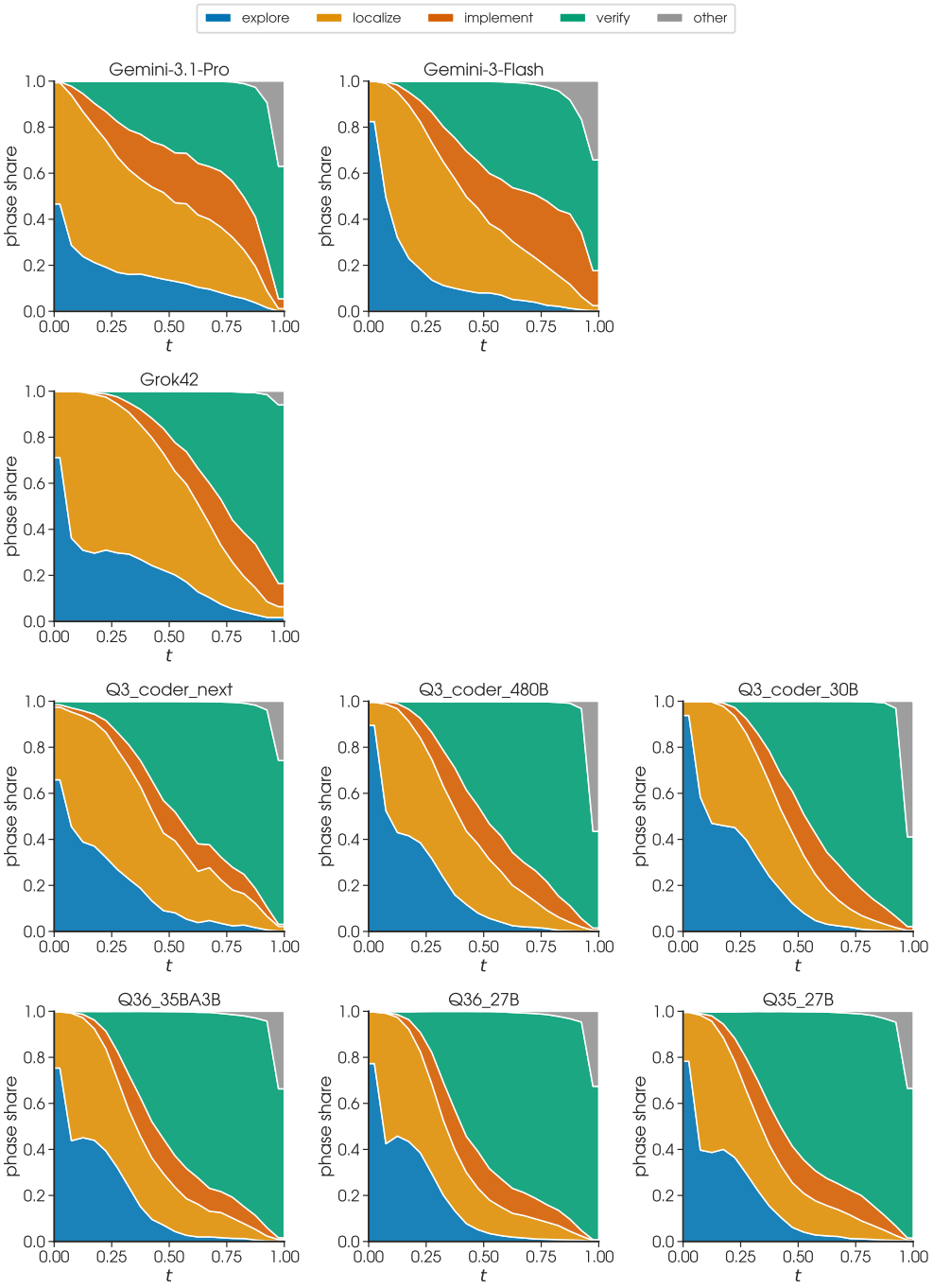


Figure 21: Phase composition (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 20.

Per-model tool-call distribution. Figures 22–23 show how each model's tool calls split across the R1 categories (`edit_source`, `read_code`, `run_suite`, `search_locate`, `vcs`, etc.). Decisive models concentrate mass on a small number of categories (e.g. Gemini family is heavy on `edit_source`), while exploratory models spread mass across many stages). Gemini carrying larger `edit_source` is consistent with higher backtracking as we observed before. The GPT and Gemini families show almost no `write_scratch` share not because they skip reproduction, but because they run them as a single bundled command. Specifically, they pipe the script into `python` through an stdin heredoc, or writing and running it in one call, which the judge scores as one `run_scratch` action. Claude and Qwen-Coder instead author the scratch file in a separate `write_scratch` step before running it, so their reproduction surfaces in both buckets.



Figure 22: Tool-call distribution per model on SWE-Bench-Verified (part 1 of 2): Anthropic Claude and OpenAI families. Each cell shows the share of calls in each R1 category (see Table 3 for LLM judge rubrics) for one model.

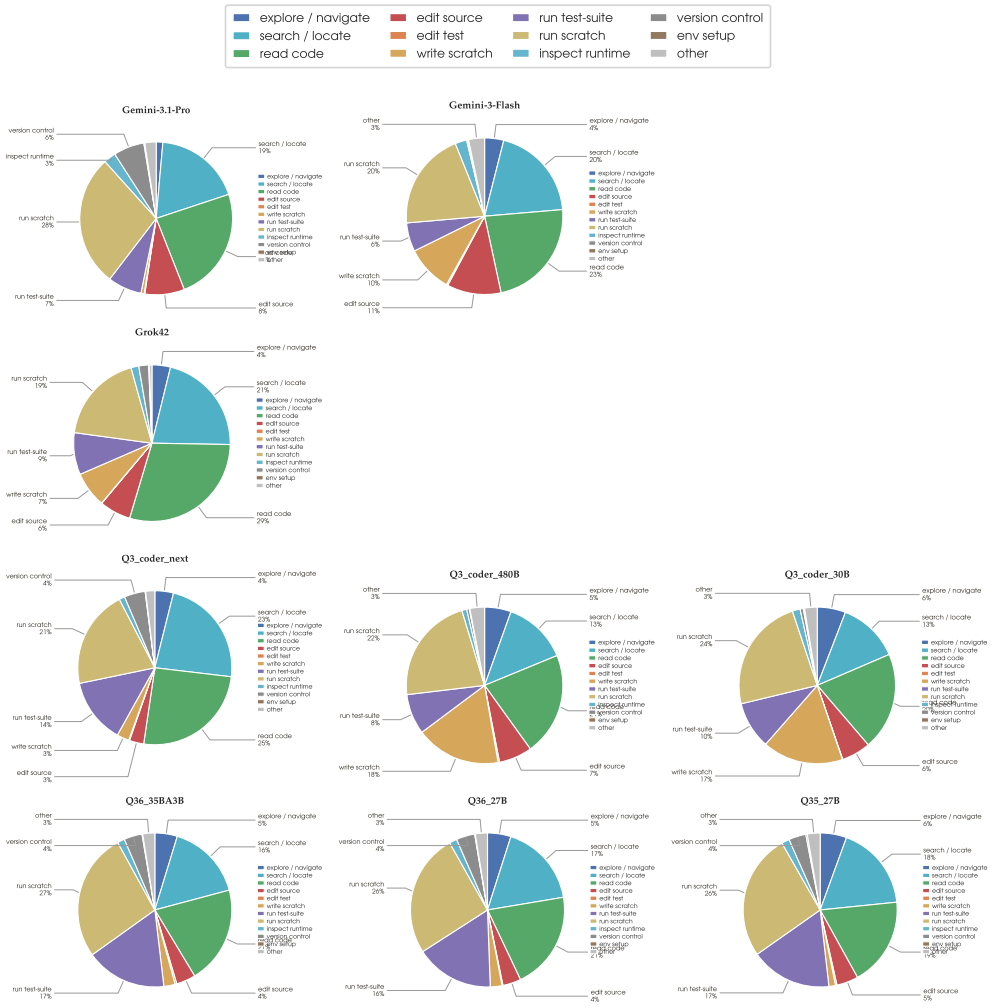


Figure 23: Tool-call distribution per model on SWE-Bench-Verified (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 22.

B.4 METRICS: SWE-BENCH-PRO

Figure 24 reports the same token-vs-pass@1 Pareto view for SWE-Bench-Pro that Figure 11 reports for SWE-Bench-Verified. The x-axis is total output tokens for a problem instance (summed over all LLM calls in an agent run), averaged across 731 instances and 5 runs per instance; the y-axis is the mean pass@1 across those five runs. Models are colored by family.

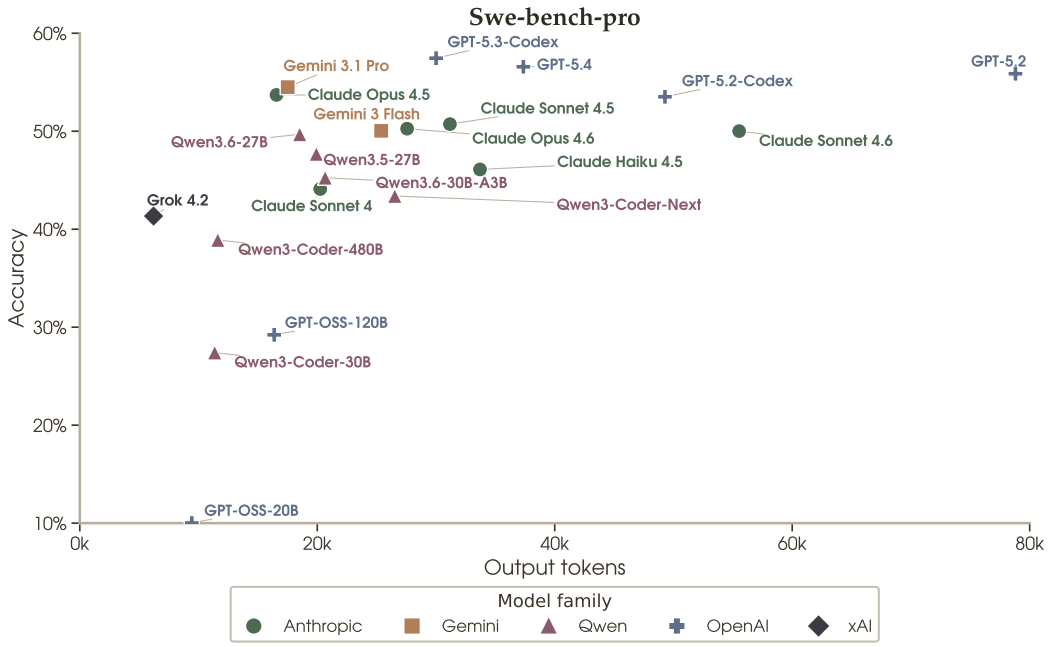


Figure 24: SWE-Bench-Pro (Deng et al., 2025): total output tokens vs. pass@1, one point per model. Output tokens are for each problem instance (summed over all LLM calls in an agent run), averaged across 731 instances and 5 runs per instance. Pass@1 is the mean of the five per-run resolution rates. Colors group models by family. Top-left corresponds to high accuracy with the fewest output tokens spent reasoning and acting. All Pass@1 values reported here are from the Base public containers; see Appendix F for the Sanitized comparison that controls for git-history leakage.

Table 7 gives the same compact distance-and-local-backtracking summary on SBP that Table 6 reports on SBV. Two observations stand out compared to SWE-Bench-Verified: **backtracking reduces across the board on SWE-Bench-Pro** — the entire field sits under 6%, from 0.3% in GPT-5.2-codex to 5.9% in Grok 4.2. The Gemini has a much lower backtracking rate here compared to SWE-Bench-Verified (3.1 Pro goes from $14.2 \rightarrow 1.7\%$; Flash from $16.9 \rightarrow 2.7\%$); Grok 4.2 has the highest backtracking rate. **Arrival-time is higher on average compared to SBV** ($t@D{\leq}0.1$ is 0.53–0.95 on SBP vs. 0.37–0.95 on SBV), consistent with SBP's larger and longer-horizon repository changes. Note that, since SBP tasks require a large set of changes to the source code, and are more involved (spread across more files), compared to atomic changes in SBV, large backtracking steps are less likely as models usually make edits one file at a time. We can also see this distinction in backtracking density Figures 12-13 where SBV carries significant mass for $\Delta D \sim 1$ which is visibly absent in SBP Figures 25-26

Figures 25–36 show the same six per-model behavioral metrics on SWE-Bench-Pro that Section B.3 reports for SWE-Bench-Verified. Note that all trajectories were for the Base public containers and not the Sanitized SBP trajectories, so per-model breakdowns in this subsection inherit the Base minus Sanitized caveat of Appendix F.

Per-model backtracking distributions (SBP). Figure 25–26 show the per-edit ΔD histogram for each model on SBP: green bars are progress edits ($\Delta D < 0$), red bars are backtracking edits ($\Delta D > 0$); the annotated percentage is the per-model backtrack-edit share.

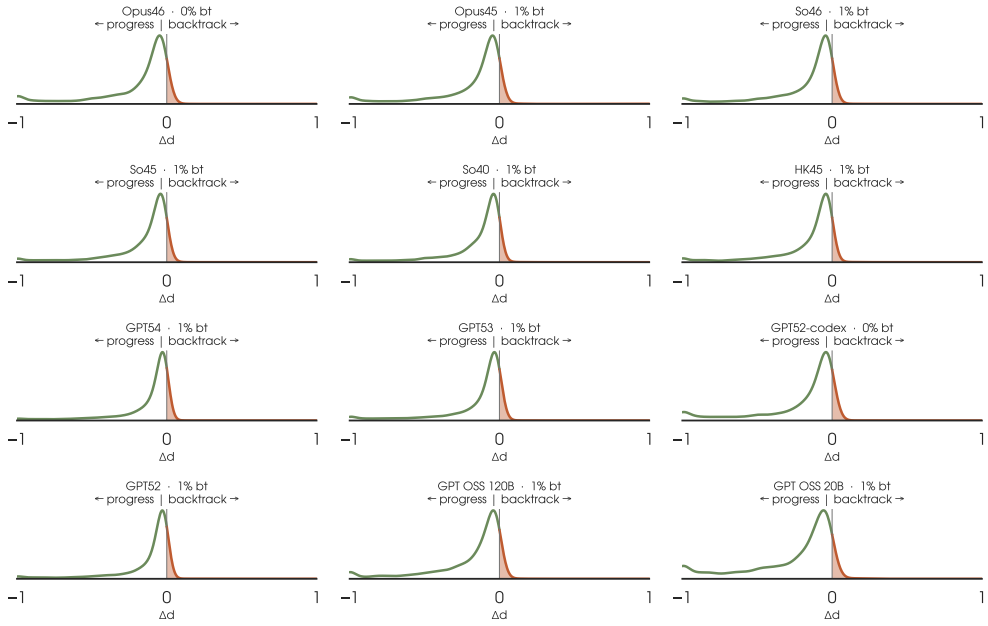


Figure 25: **Local backtracking.** Histograms of ΔD on SWE-Bench-Pro (part 1 of 2): Anthropic Claude and OpenAI families. Each cell is one model; green = progress edits, red = local backtracking edits. Annotation shows the per-model local-backtrack share.

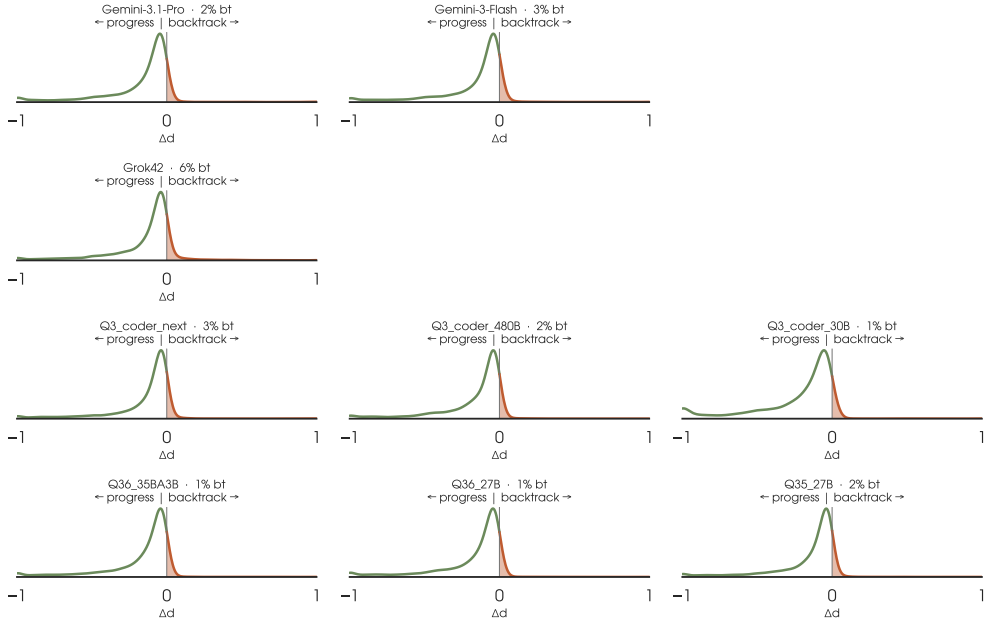


Figure 26: **Local backtracking.** Histograms of ΔD on SWE-Bench-Pro (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 25.

Table 7: Distance, local backtracking and outcome summary over resolved SWE-Bench-Pro trajectories per model. Local backtrack %, arrival-time t @ $D \leq 0.1$ and area-under-curve (AUC) are computed from the faithful state-replay curves over 731-instance (5 runs each) per model. Pass@1 is the SSA published value (Base containers; matches Appendix C).

Model	Local backtrack %	t @ $D \leq 0.1$	AUC	pass@1 %
Opus 4.8	0.8	0.63	0.44	68.48
Opus 4.6	0.4	0.74	0.55	57.45
Opus 4.5	1.1	0.68	0.50	54.58
Sonnet 4.6	0.6	0.84	0.68	53.67
Sonnet 4.5	0.9	0.53	0.36	50.15
Sonnet 4.0	1.0	0.68	0.44	44.26
Haiku 4.5	1.0	0.63	0.42	46.37
GPT-5.5	1.6	0.74	0.46	59.42
GPT-5.4	0.5	0.84	0.61	59.90
GPT-5.3	0.6	0.84	0.57	58.57
GPT-5.2	0.8	0.89	0.63	56.47
GPT-5.2-codex	0.3	0.95	0.71	53.87
GPT-OSS 120B	1.1	0.95	0.57	43.61
GPT-OSS 20B	0.7	0.95	0.63	39.67
Gemini 3.1 Pro	1.7	0.89	0.51	54.00
Gemini 3 Flash	2.7	0.79	0.46	49.43
Grok 4.2	5.9	0.95	0.70	51.57
Qwen3-Coder Next	2.7	0.74	0.47	57.75
Qwen3-Coder 480B	1.8	0.68	0.45	38.90
Qwen3-Coder 30B	0.7	0.68	0.48	27.41
Qwen3.6 35B-A3B	1.3	0.63	0.45	46.70
Qwen3.6 27B	0.9	0.68	0.48	51.40
Qwen3.5 27B	2.0	0.68	0.46	47.44

Per-model edit/read ratio (SBP). Figure 27–28 shows the per-cycle ratio of edit-oriented tool calls to total ones across resolved + unresolved trajectories. A high ratio means the agent mostly edits and a low ratio means it mostly explores/test.

Per-model level families (SBP). Figures 29–30 show the level family, mean $\bar{D}(t)$ and backtracking term $\bar{B}(t)$ of the resolved trajectories for each model on SBP, in the format of Figure 16.

Per-model genuine-error rate over cycles (SBP). Figure 31–32 plot the rate of genuine tool errors per cycle (R8 = `genuine`, see Section B.1.2), broken down by tool family.

Per-model phase composition (SBP). Figure 33–34 show how each trajectory's time is split across the four R6 phases on SBP.

Per-model tool-call distribution (SBP). Figures 35–36 show how each model's tool calls split across the R1 categories on SWE-Bench-Pro.

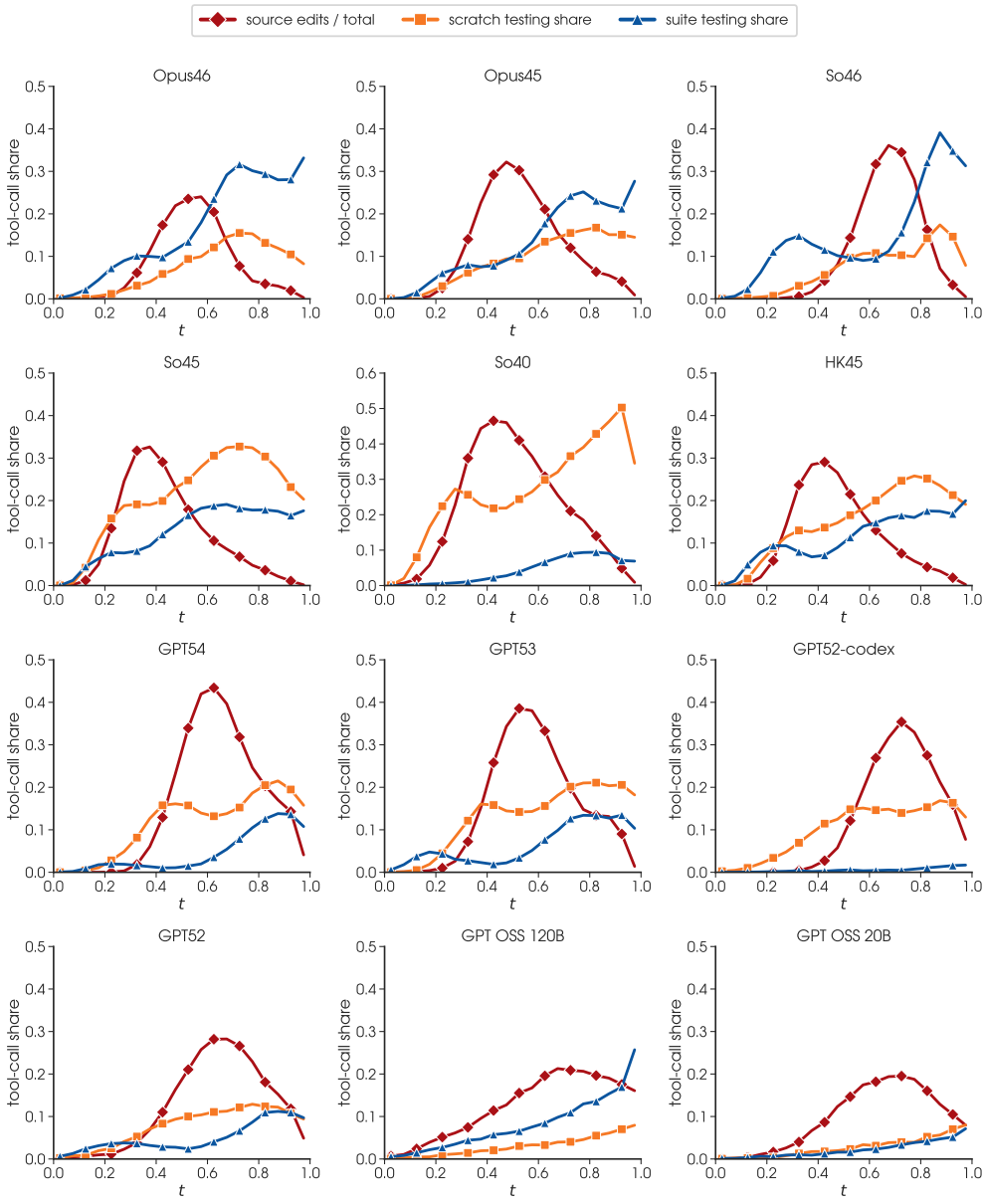


Figure 27: Edit/test ratio per cycle on SWE-Bench-Pro (part 1 of 2): Anthropic Claude and OpenAI families. Each subplot shows the model's distribution of ratios.

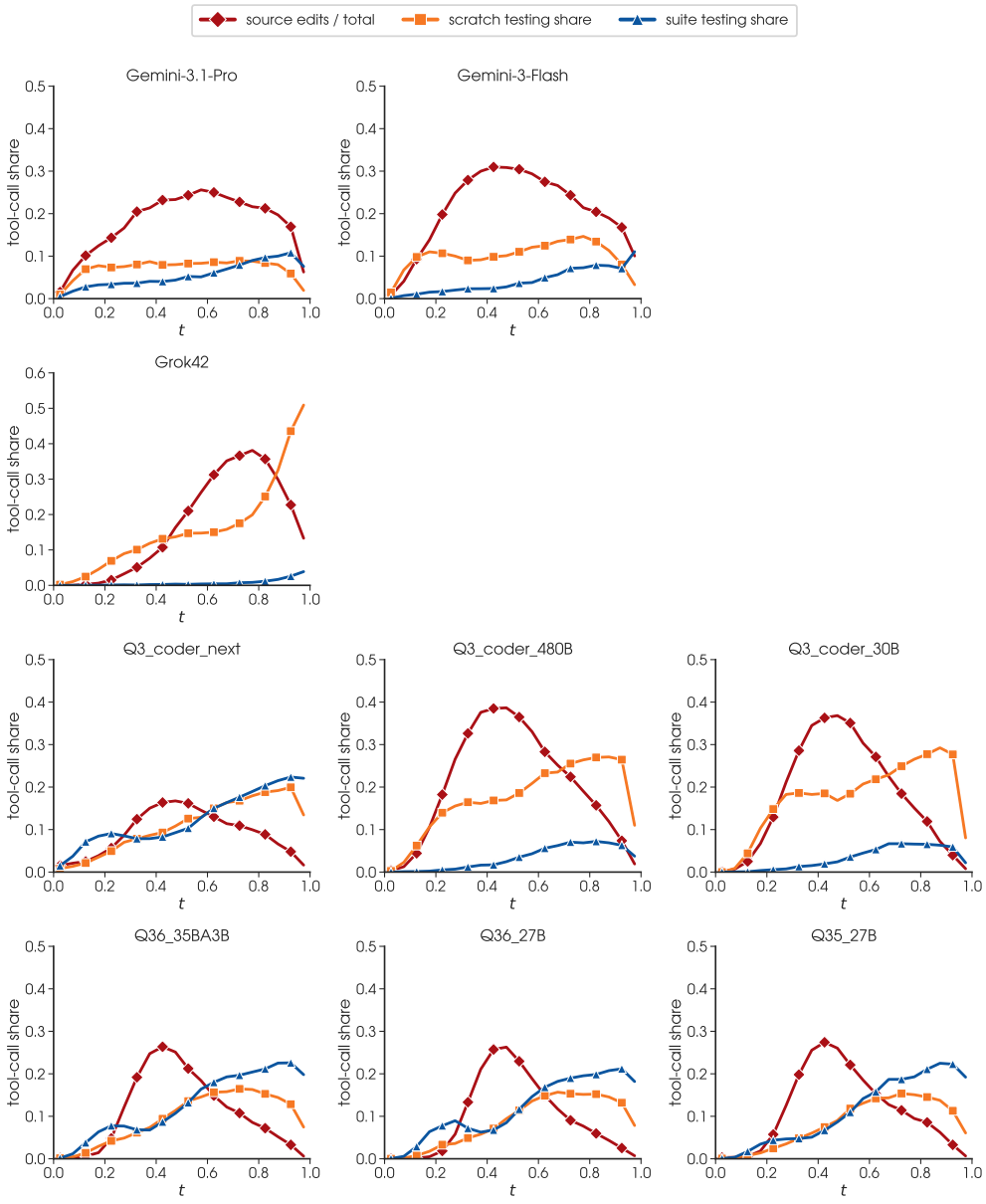


Figure 28: Edit/test ratio per cycle on SWE-Bench-Pro (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 27.

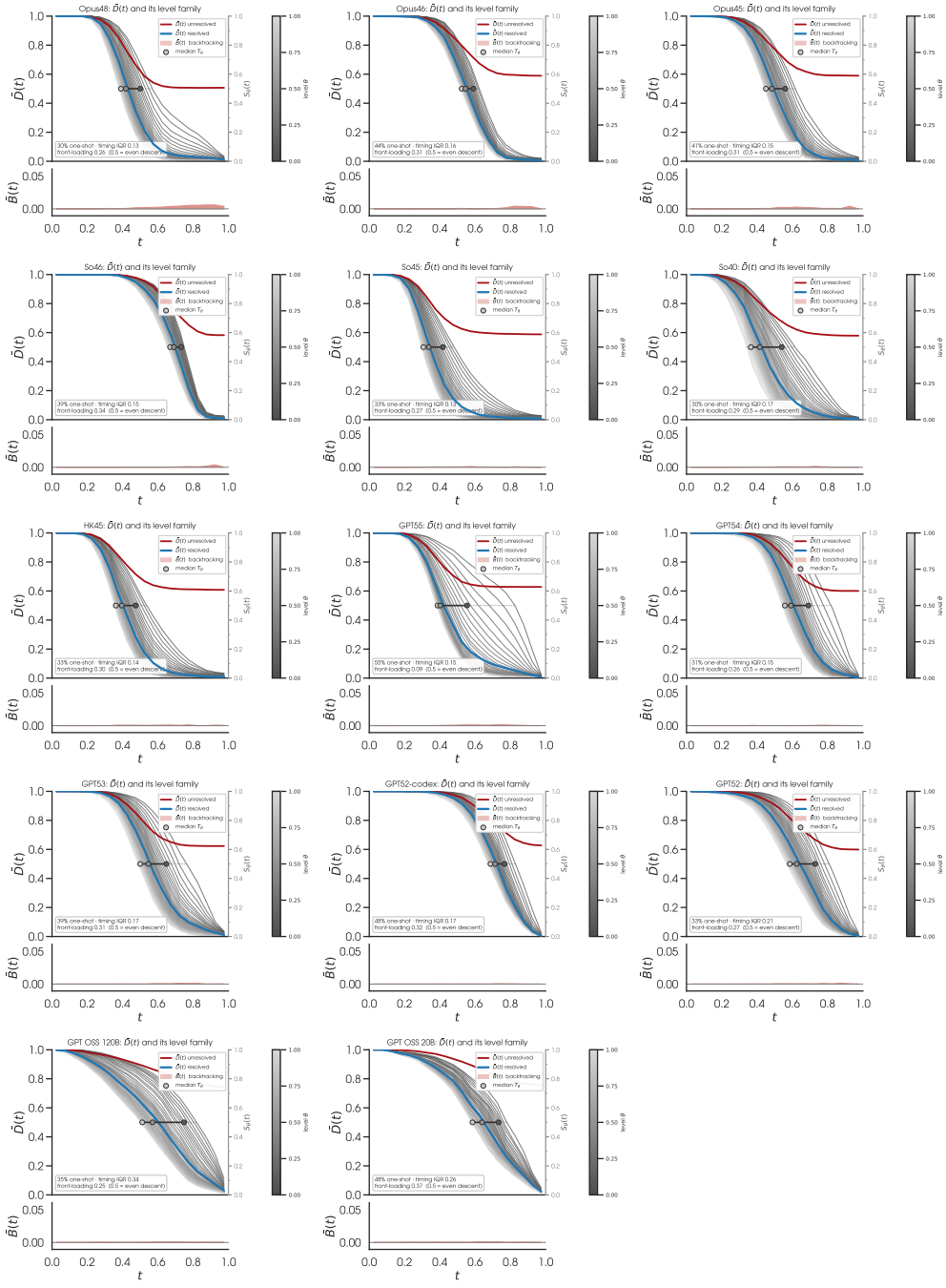


Figure 29: Level families on SWE-Bench-Pro (part 1 of 2): Claude and GPT families. Grey curves are the level sets $S_\theta(t)$ of the resolved trajectories (Section 3), the blue curve is their mean $\bar{D}(t)$, the dark-red curve is the mean $\bar{D}(t)$ of the unresolved runs (which plateaus where they stop), the markers give the median crossing time T_θ (the median over runs of the first-entry time $T_r(\theta)$, i.e. where S_θ crosses $1/2$) at $\theta = 0.5$ with its interquartile range, and the strip beneath each panel is the backtracking term $\bar{B}(t)$. The inset reports the one-shot share (runs that land the patch in a single edit); the timing IQR, the interquartile range of the runs' half-way times $T_r(0.5)$, i.e. how dispersed solve timing is across runs; and the front-loading index $(T_{0.5} - T_{0.9})/(T_{0.1} - T_{0.9})$, the position of the median half-way time between the median crossing times of levels 0.9 and 0.1 (0.5 = even descent; lower = most of the distance falls early).

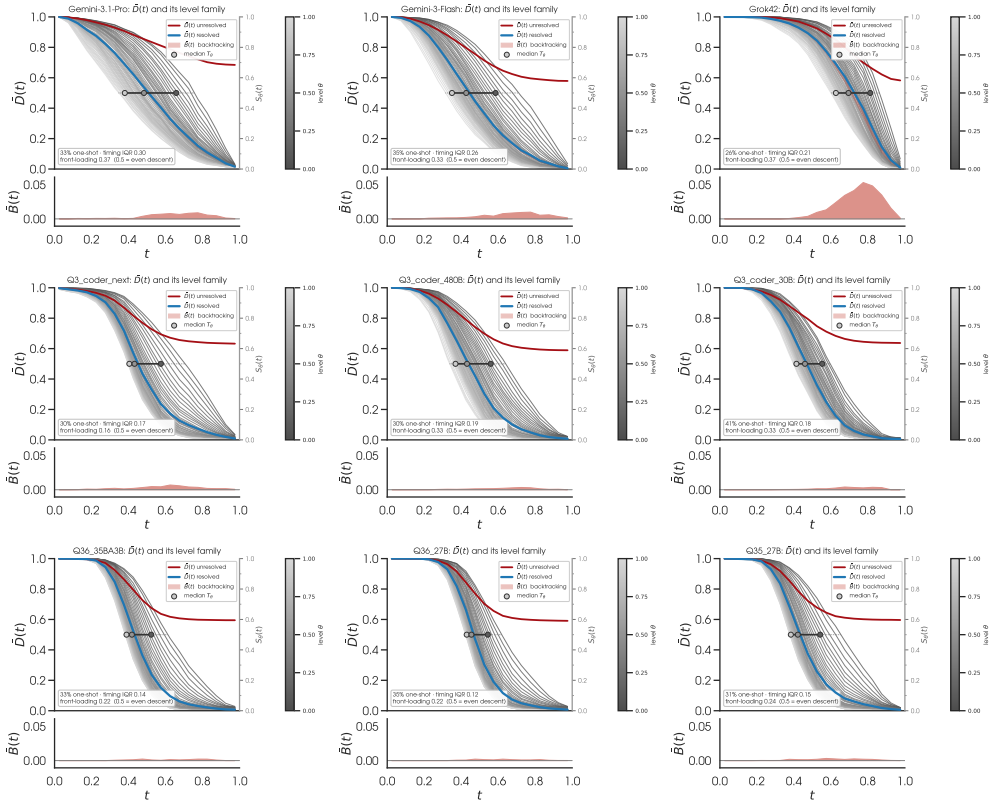


Figure 30: Level families on SWE-Bench-Pro (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 29.

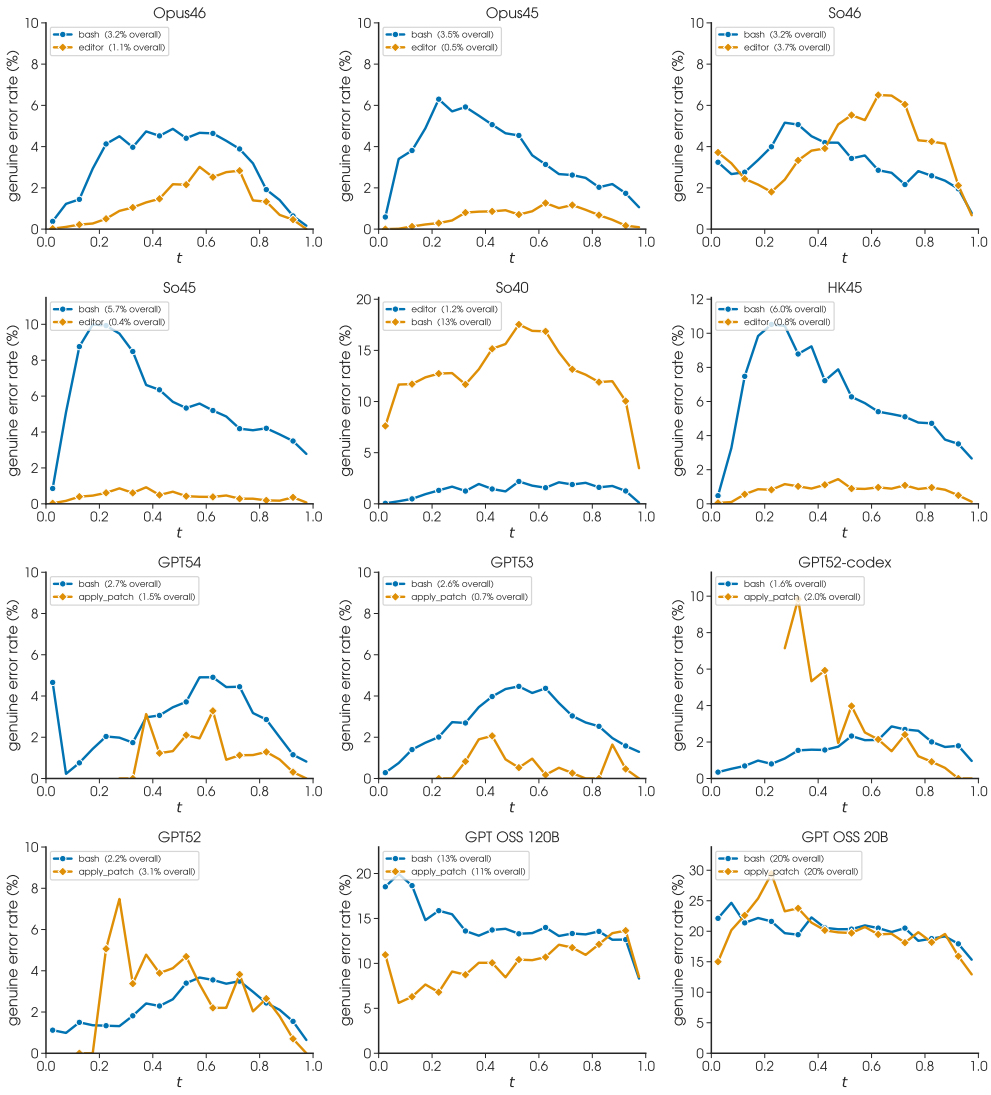


Figure 31: Genuine tool-error rate per cycle on SWE-Bench-Pro (part 1 of 2): Anthropic Claude and OpenAI families. Benign exits (`grep` no-match, `pytest` test failures, etc.) are excluded.

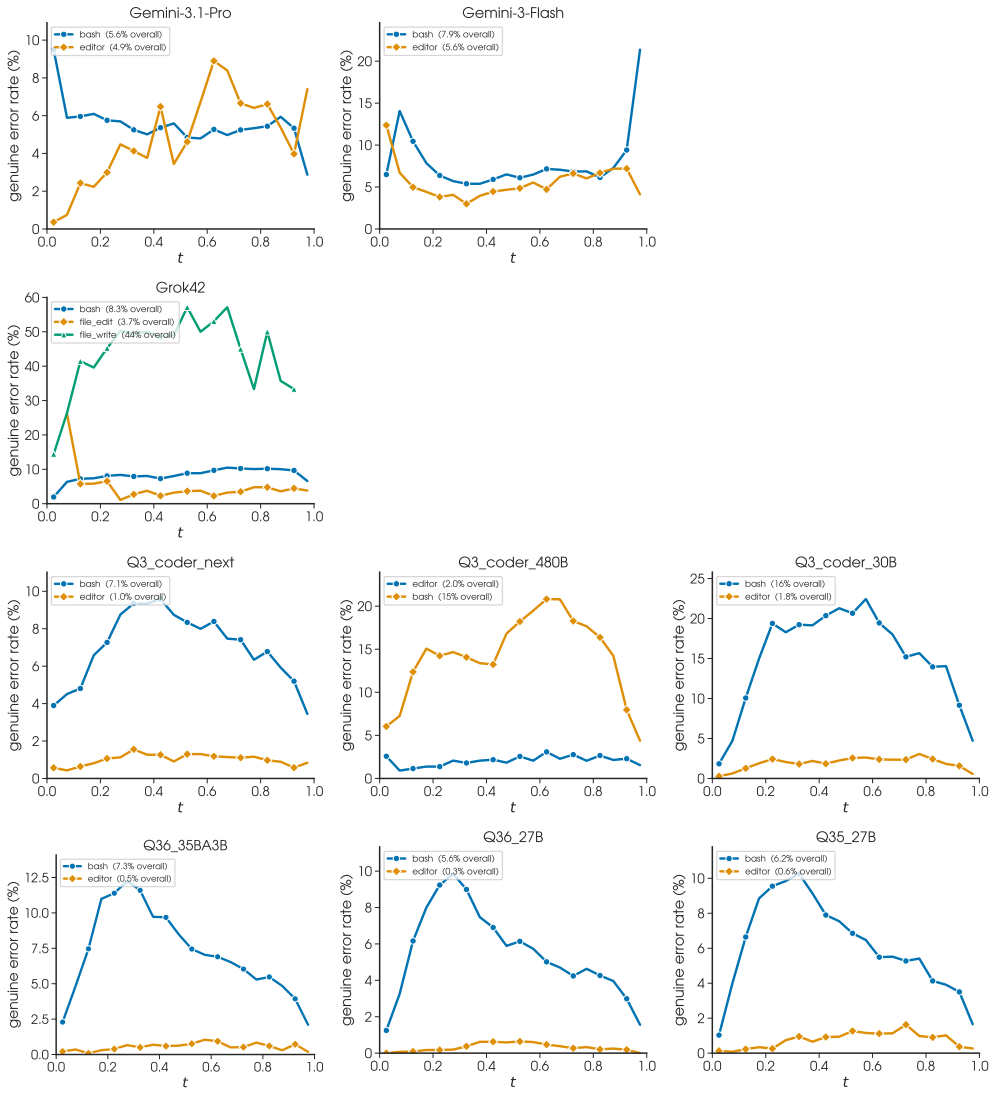


Figure 32: Genuine tool-error rate per cycle on SWE-Bench-Pro (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 31.

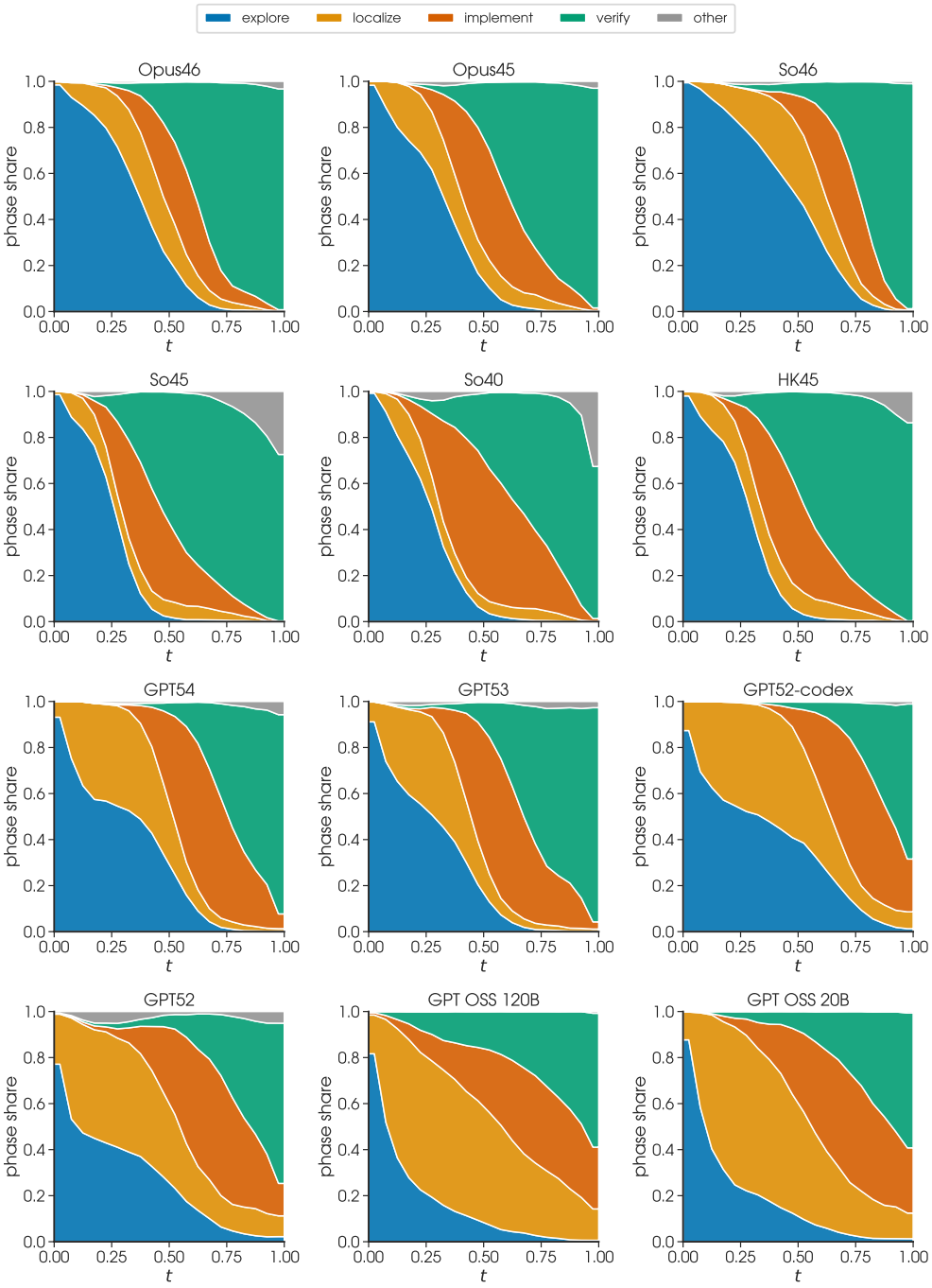


Figure 33: **Phase composition surfaces model-specific schedules.** Per-cycle share of the four R6 (see LLM judge rubrics in Table 3) phases (`explore`, `localize`, `implement`, `verify`) vs. normalised cycle position t on SWE-Bench-Pro, for Claude and GPT families (part 1 of 2). Given the same tasks, models schedule their cycles differently. The implementation peak shifts in position and width, and the verify phase carries different long-tail mass.

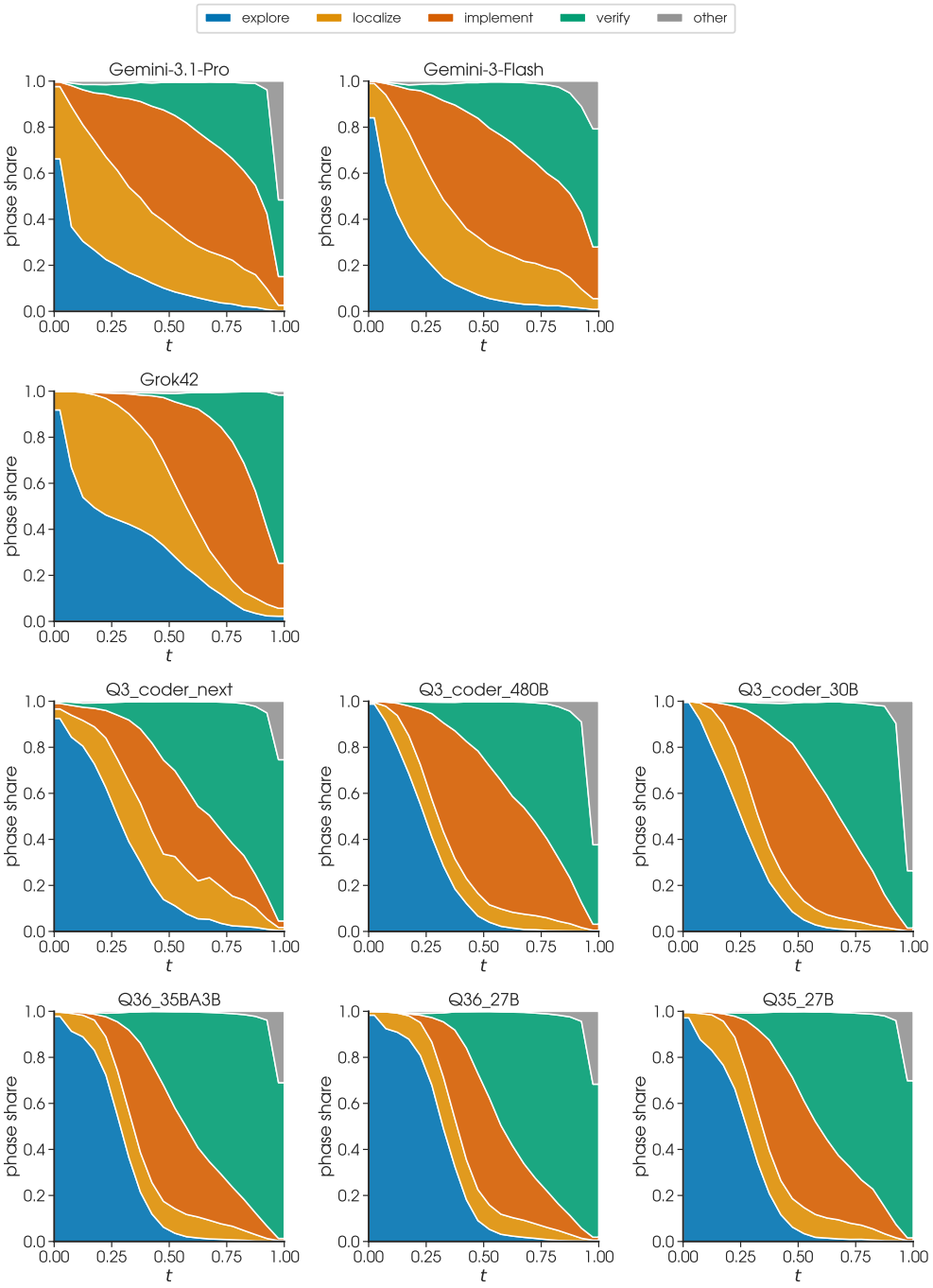


Figure 34: Phase composition on SWE-Bench-Pro (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 33.

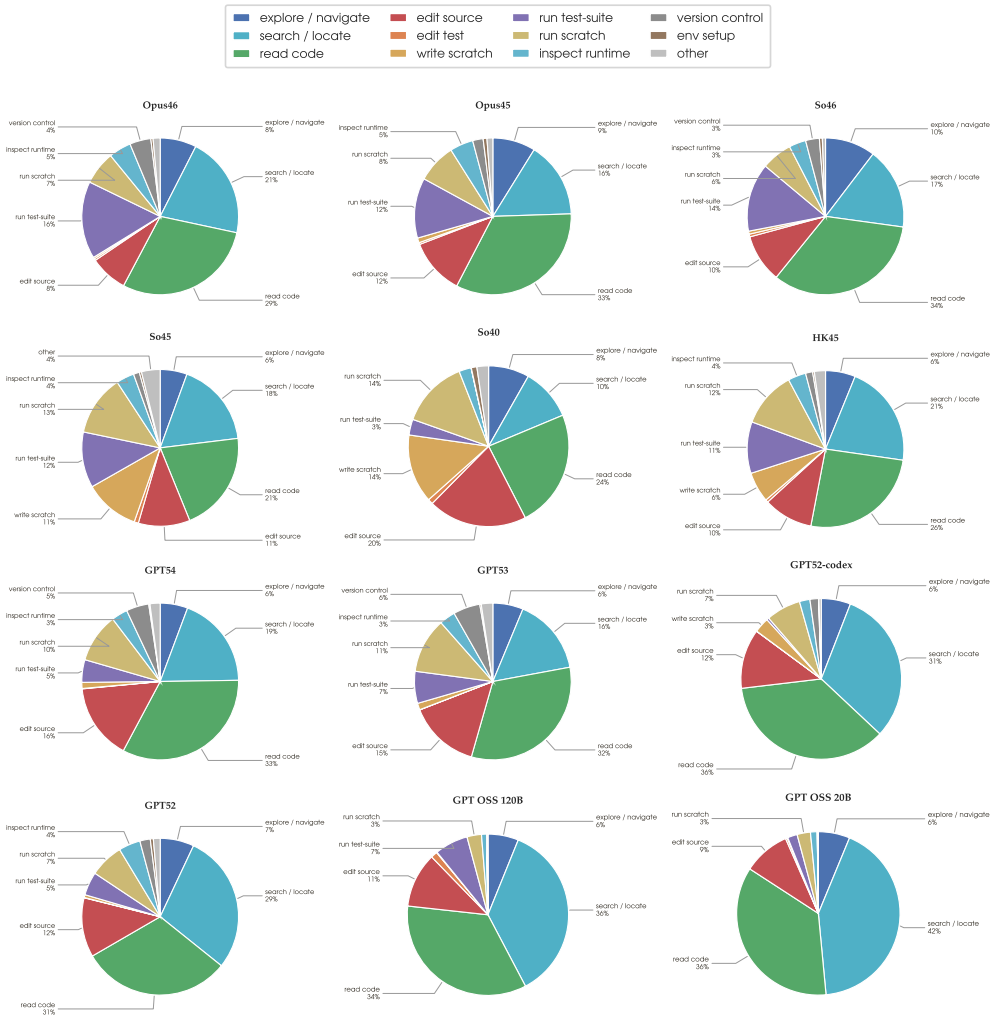


Figure 35: Tool-call distribution per model on SWE-Bench-Pro (part 1 of 2): Anthropic Claude and OpenAI families. Each cell shows the share of calls in each R1 category (see Table 3 for LLM judge rubrics) for one model.

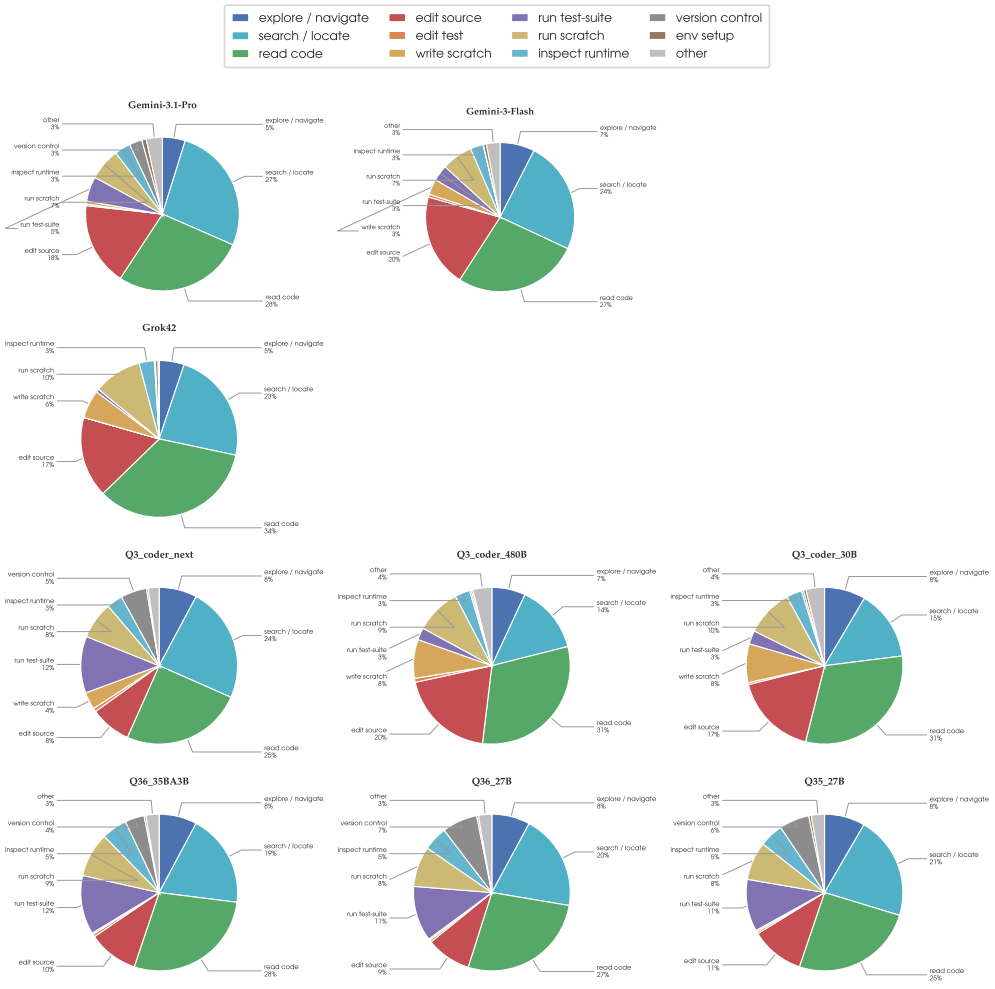


Figure 36: Tool-call distribution per model on SWE-Bench-Pro (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 35.

B.5 METRICS: TERMINAL-BENCH-2

The Terminal-Bench-2 (TB2) metrics in this section are computed by the same windowed judge pipeline as on SWE (Section B.1.1), but with a TB2-specific rubric variant. TB2 tasks *build or author* a deliverable under `/app` rather than patch a checked-out repo under `/testbed`, ship no project test suite, and are graded by an externally-run success command on the same container. The judge, therefore, reuses the `R1`, `R3`, `R5`, `R7`, and `R8` fields of Table 3 unchanged and only re-shapes the task-dependent entries. Table 8 lists the differences.

Table 8: Terminal-Bench-2 judge rubric: the only fields that differ from the SWE rubric of Table 3. `R1`, `R3`, `R5`, `R7`, `R8` are unchanged.

Field	SWE rubric (Table 3)	TB2 rubric
Workspace	`/testbed` (repo at `base_commit`)	`/app` (task workspace; no golden patch)
`R2 target_kind`	`project_source` | `project_test` | `scratch` | `build_config` | `vcs_internal` | `none`	`task_artifact` | `upstream_source` | `scratch` | `build_config` | `vcs_internal` | `none` (splits the edited artifact into the deliverable vs. 3rd-party source being built/patched)
`R4 test_activity`	`suite` | `scratch` | `none`	`success_check` | `scratch` | `none` (no project suite; `success_check` is the task's declared success oracle)
`R6 phase`	`explore` | `localize` | `implement` | `verify` | `other`	`setup_env` | `build` | `explore` | `implement` | `verify` | `cleanup` | `other` (drops `localize`; promotes `setup_env`/`build` out of `other`; adds post-solve `cleanup`)
New-file default	A freshly created `/testbed` file is usually `scratch`; the SWE-Pro addendum carves out new package-tree source as `edit_source`.	Inverted: a new `/app` file is the deliverable by default (`task_artifact`/`edit_source`) unless it is clearly probe-shaped or under `/tmp`.

Figure 37 reports the same token-vs-pass@1 Pareto view for Terminal-Bench-2 that Figures 11 and 24 report for SWE-Bench-Verified and SWE-Bench-Pro respectively. Open-source families plotted here use the Unconstrained setting (the only one we ran for them). Closed-source models use the Constrained setting to match Appendix C.

Figures 38–45 present per-model behavioral breakdowns from Section B.3 on Terminal-Bench-2. The two trajectory-replay panels (ΔD histograms and solution-distance curves) are omitted because TB2 does not expose per-instance gold patches in a unified-diff form, so the empirical solution manifold $\tilde{S}_i$ used in Sections 3–B.3 cannot be constructed. The tool-use, phase and error metrics shown here are computed with the TB2-specific rubric variant introduced at the top of this section (Table 8).

Per-model phase composition (TB2). Figures 38–39 show how each TB2 trajectory's time is split across the TB2 R6 phases (Table 8) — `setup_env` (install/configure the environment), `build` (run build tooling against upstream/system code), `explore` (orient and read docs/source), `implement` (write or patch the deliverable), `verify` (run the container/self tests), and `cleanup` (post-solve teardown). TB2 tasks are heavier on `setup_env` / `build` than SWE-Bench instances, which have no analogous phases.

Per-model genuine-error rate over cycles (TB2). Figures 40–41 plot the rate of *genuine* tool errors per cycle (R8 = `genuine`, see Section B.1.2), broken out by tool family. Benign non-zero exits (`grep` no-match, expected non-zero exit codes, etc.) are excluded so spikes here reflect real harness friction, not task outcomes.

Per-model edit/read ratio (TB2). Figures 42–43 shows the per-cycle ratio of edit-oriented tool calls to total ones across resolved + unresolved trajectories. A high ratio means the agent mostly edits and a low ratio means it mostly explores/test.

Per-model tool-call distribution (TB2). Figures 44–45 show how each model's tool calls split across the R1 categories on Terminal-Bench-2. The categories are identical to the SWE figures, with

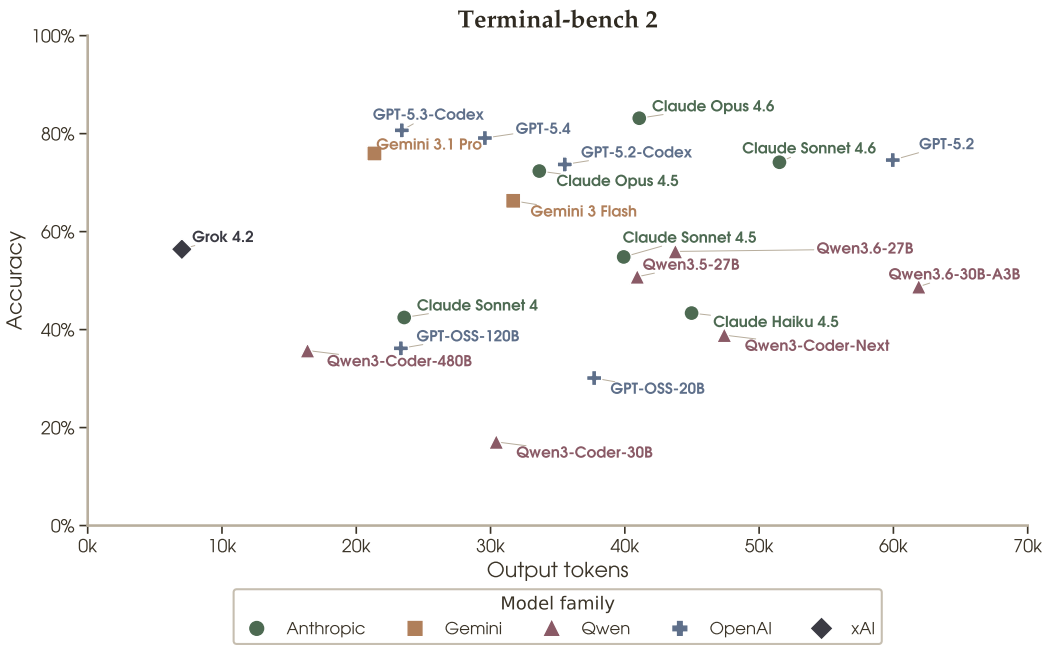


Figure 37: Terminal-Bench-2: total output tokens vs. pass@1, one point per model. Output tokens are for each problem instance (summed over all LLM calls in an agent run), averaged across 89 instances and 5 runs per instance . Pass@1 is the mean of the five per-run resolution rates. Colors group models by family. Top-left corresponds to high accuracy with the fewest output tokens spent reasoning and acting.

`env_setup` and `inspect_runtime` carrying notably more mass on TB2 as the containers often do not contain all the dependencies (e.g. `pytorch`, `R` etc) required to solve the task.

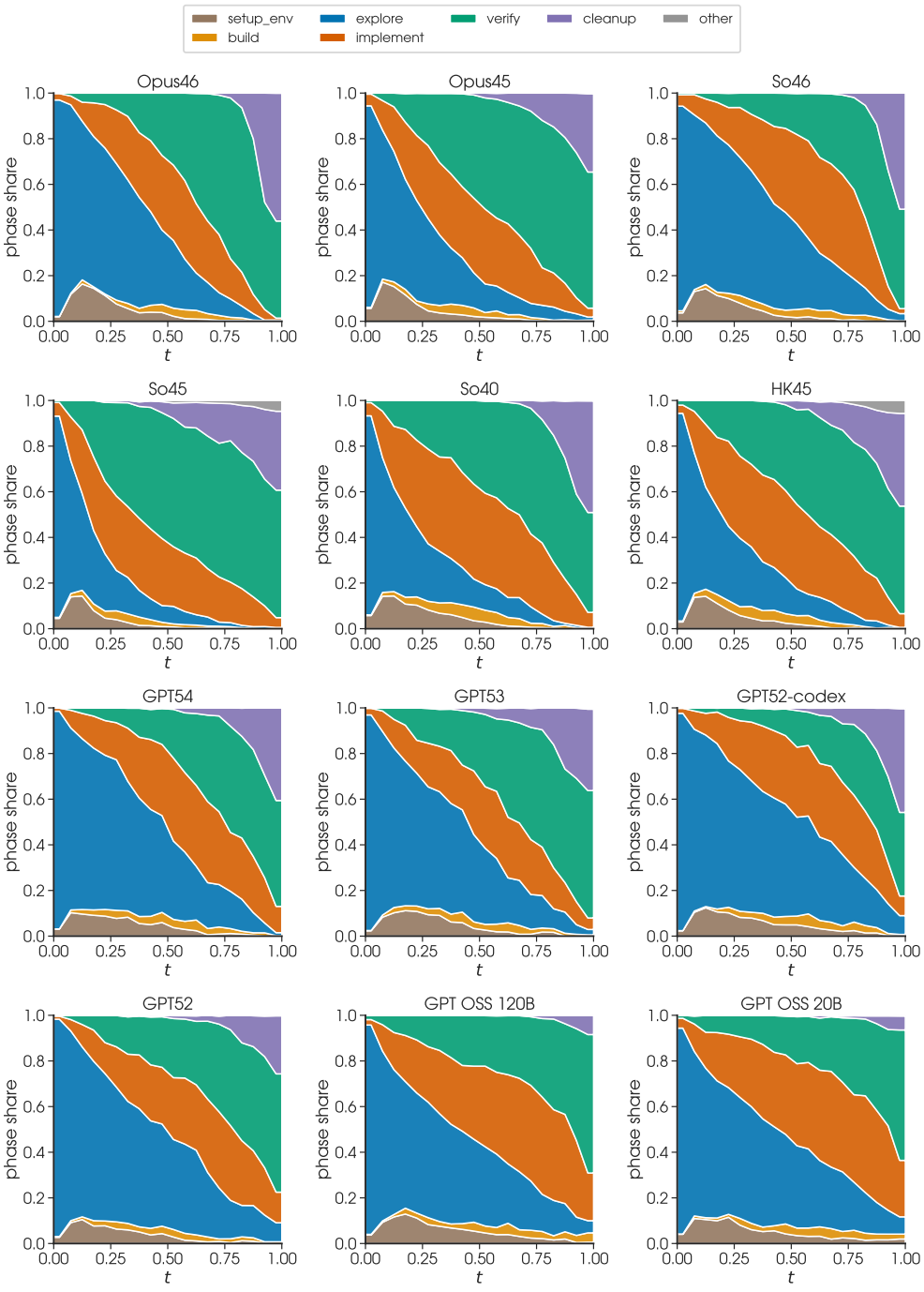


Figure 38: **Phase composition surfaces model-specific schedules.** Per-cycle share of the four R6 (see LLM judge rubrics in Table 3) phases (`explore`, `localize`, `implement`, `verify`) vs. normalised cycle position t on Terminal-Bench-2, for Claude and GPT families (part 1 of 2). Given the same tasks, models schedule their cycles differently. The implementation peak shifts in position and width, and the verify phase carries different long-tail mass.

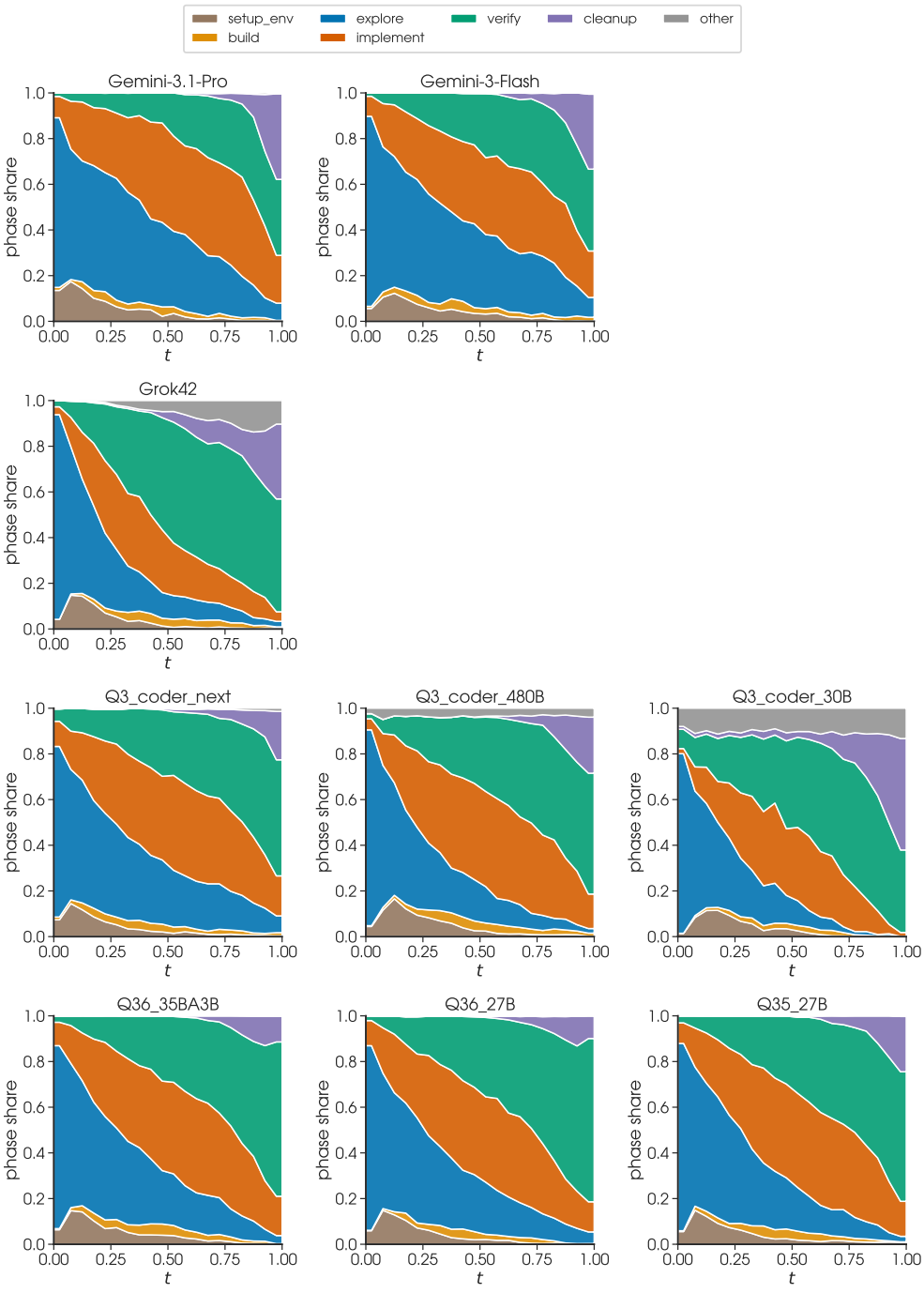


Figure 39: Phase composition on Terminal-Bench-2 (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 38.

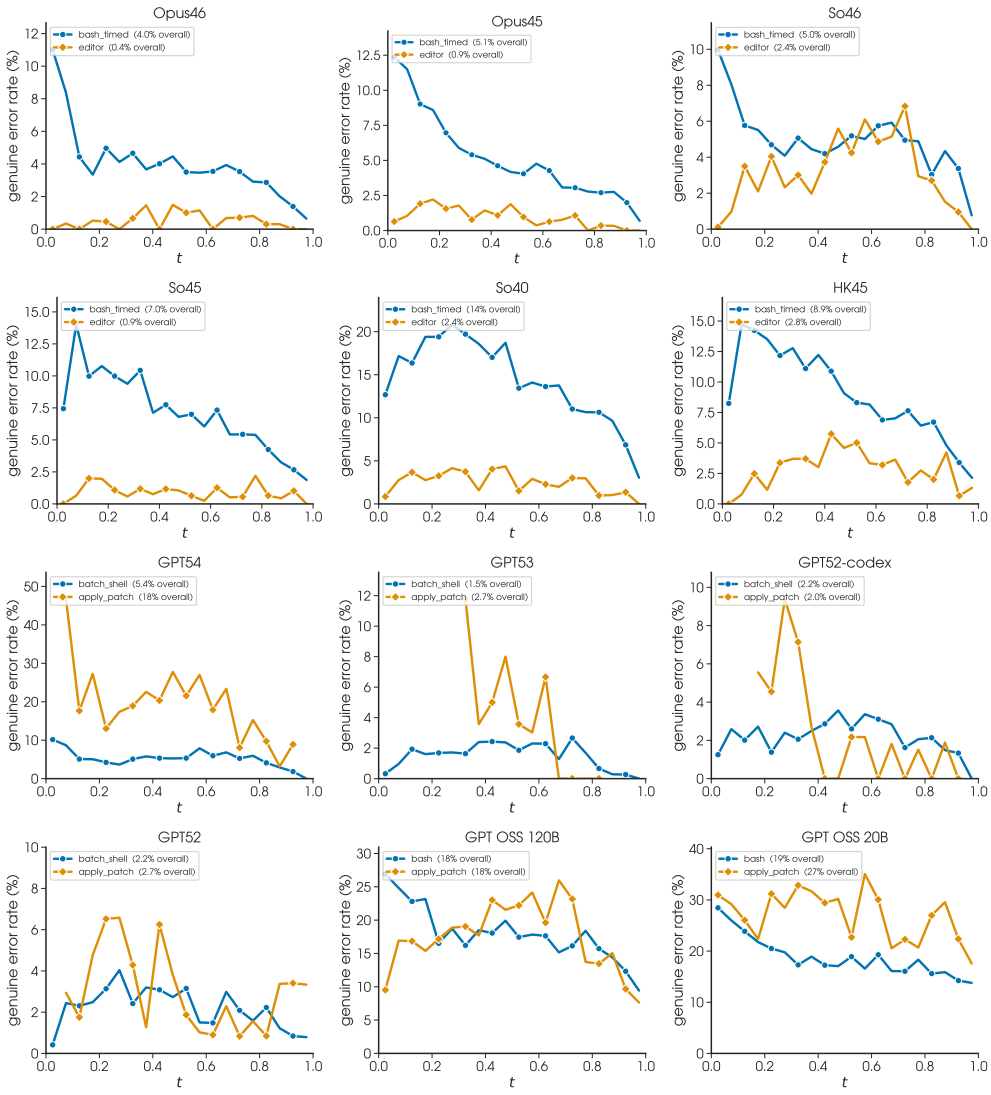


Figure 40: Genuine tool-error rate per cycle on Terminal-Bench-2 (part 1 of 2): Anthropic Claude and OpenAI families. Benign exits excluded; tool families broken out within each sub-plot.

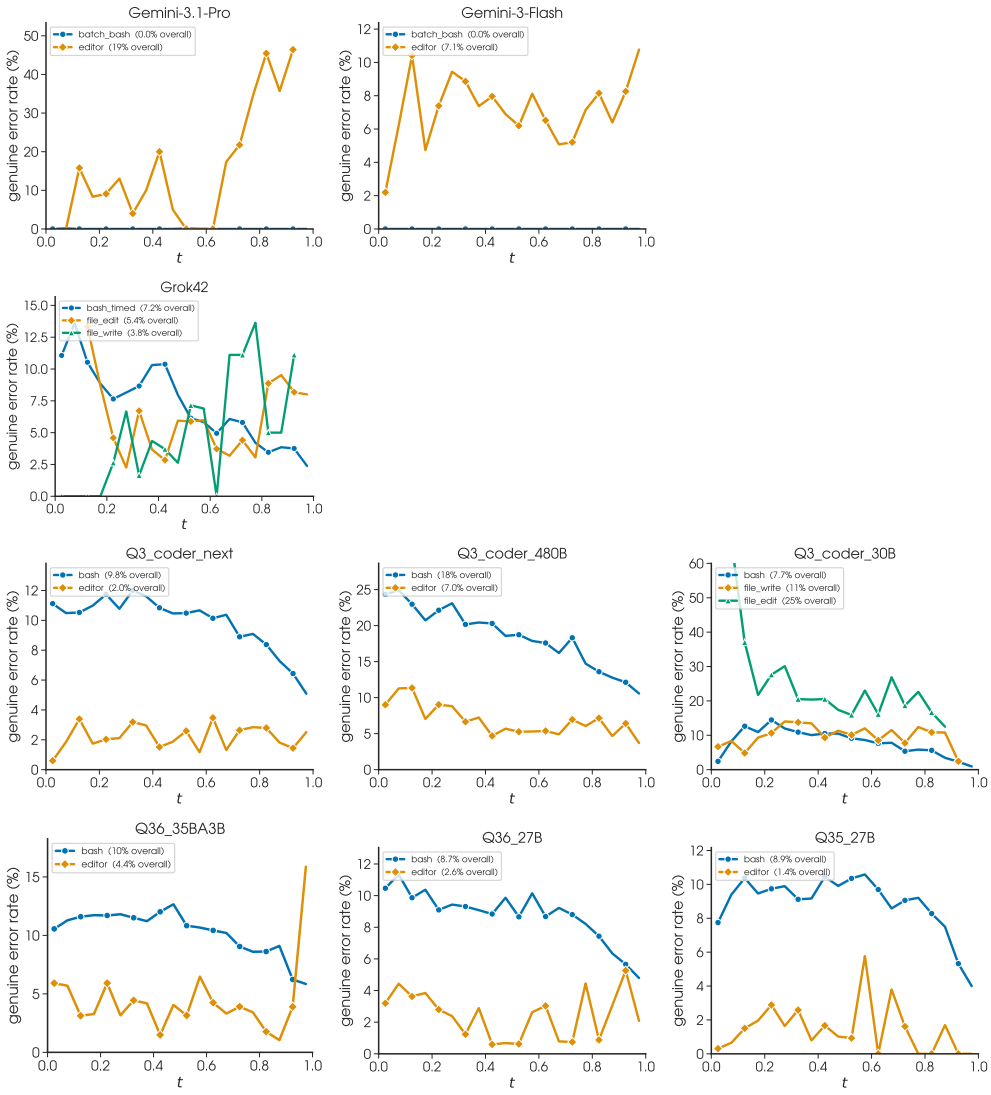


Figure 41: Genuine tool-error rate per cycle on Terminal-Bench-2 (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 40.

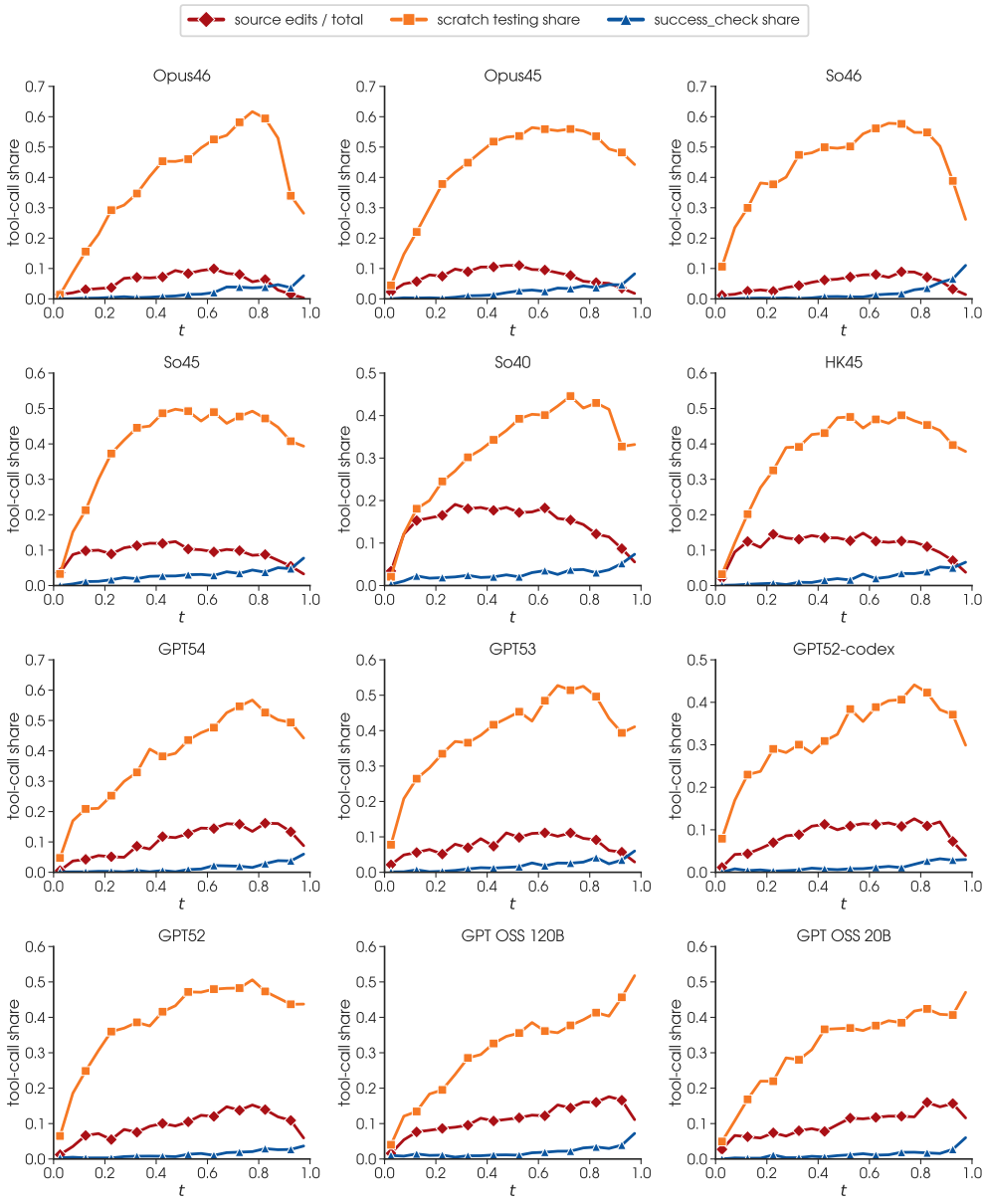


Figure 42: Edit/test ratio per cycle on Terminal-Bench-2 (part 1 of 2): Anthropic Claude and OpenAI families. Each cell shows the model's distribution of ratios.

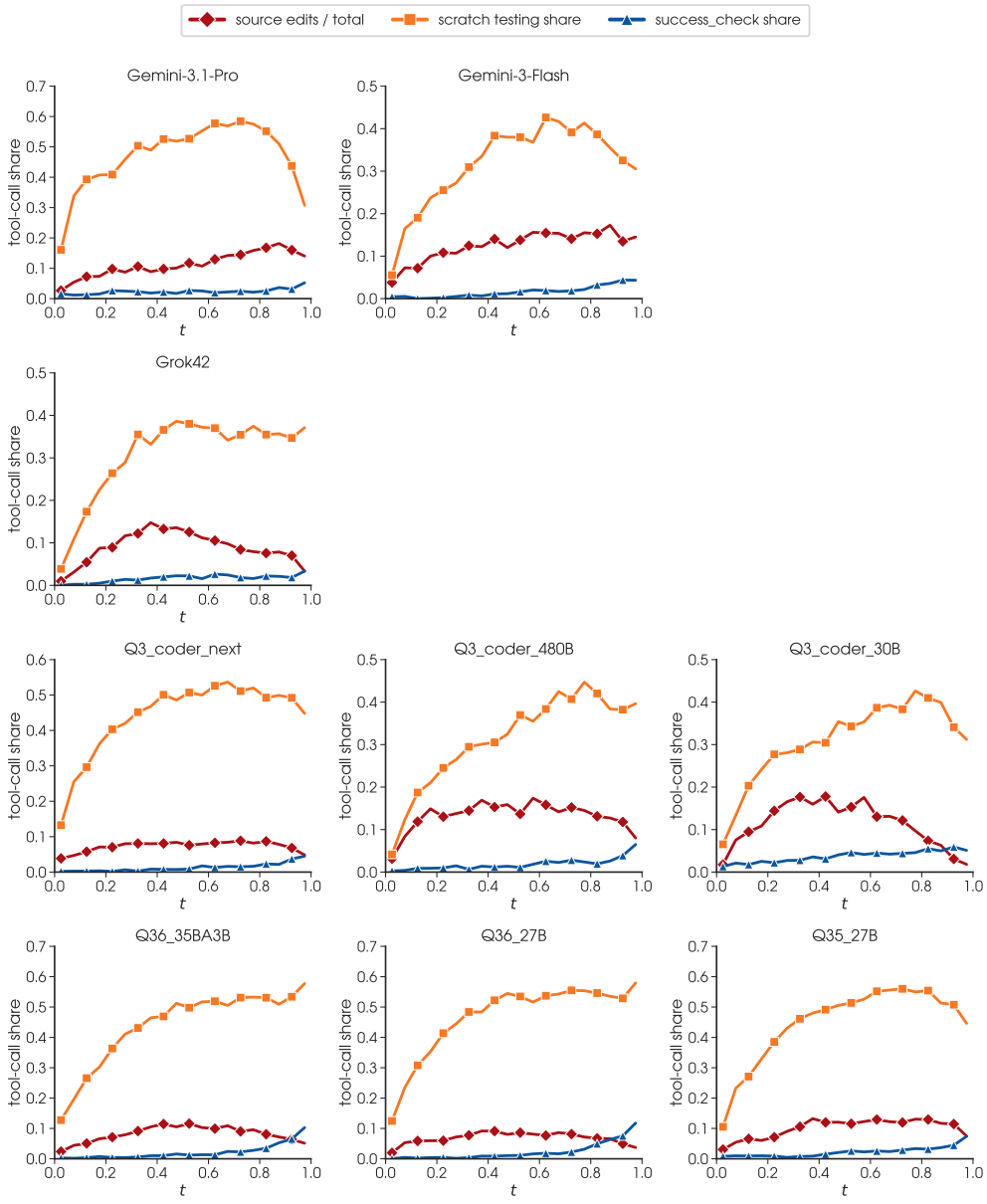


Figure 43: Edit/test ratio per cycle on Terminal-Bench-2 (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 42.



Figure 44: Tool-call distribution per model on Terminal-Bench-2 (part 1 of 2): Anthropic Claude and OpenAI families. Each cell shows the share of calls in each R1 category (see Table 3 for LLM judge rubrics) for one model.



Figure 45: Tool-call distribution per model on Terminal-Bench-2 (part 2 of 2): Gemini, Grok and Qwen families. Continued from Figure 44.

B.6 METRICS: DEEP-SWE

The figures in this section show the per-model behavioral metrics of Appendix B.3 for the 17 models evaluated on Deep-SWE, in the same order and format, so that each metric can be read against its SWE-Bench-Verified counterpart. Models are ordered by family, with families in alphabetical order: Claude (Opus 5.5, Opus 5.0, Opus 4.8, Sonnet 5.0, Fable 5), Gemini (3.8 Flash), GPT (GPT-6 Astra, Sol and Luna; GPT-5.6 Sol, Terra and Luna; GPT-5.5), Grok (4.7), Kimi (K3), Muse Spark (1.3) and Qwen (Qwen3.8 27B). Deep-SWE tasks require larger repairs than SWE-Bench-Verified, and the differences are visible in every metric: the patch is built over many edits rather than one, the level families fan out over most of the trajectory, and the backtracking-edit share is well below the SWE-Bench-Verified range for every model.

Table 9 gives the same compact distance-and-local-backtracking summary on Deep-SWE that Table 6 reports on SWE-Bench-Verified. Local backtracking sits an order of magnitude below the SWE-Bench-Verified range (0.2–2.0% against 0.9–16.9%) and arrival is late for every model (t @ $D \leq 0.1$ between 0.74 and 0.95), since a Deep-SWE patch is assembled over many edits rather than landed in one; the spread in AUC (0.39–0.70) is nonetheless as wide as on SWE-Bench-Verified.

Table 9: Distance, local backtracking and outcome summary over resolved Deep-SWE trajectories per model. Local backtrack %, arrival-time t @ $D \leq 0.1$ and area-under-curve (AUC) are computed from the faithful state-replay curves over 113 instances (5 runs each) per model; pass@1 is from Appendix C. Families in alphabetical order.

Model	Local backtrack %	t @ $D \leq 0.1$	AUC	pass@1 %
Opus 5.5	2.0	0.95	0.58	74.33
Opus 5.0	1.6	0.89	0.47	72.56
Opus 4.8	1.2	0.79	0.50	61.28
Sonnet 5.0	1.7	0.79	0.39	57.25
Fable 5	1.1	0.79	0.47	68.84
Gemini 3.8 Flash	0.6	0.79	0.56	73.10
GPT-6-Astra	0.2	0.79	0.52	77.70
GPT-6-Sol	0.6	0.84	0.48	76.10
GPT-6-Luna	1.2	0.74	0.45	64.24
GPT-5.6-Sol	1.1	0.74	0.46	72.56
GPT-5.6-Terra	1.2	0.74	0.45	69.20
GPT-5.6-Luna	1.4	0.79	0.43	65.13
GPT-5.5	0.5	0.74	0.47	74.51
Grok 4.7	1.2	0.95	0.70	72.39
Kimi-K3	0.9	0.79	0.45	77.00
Muse Spark 1.3	0.5	0.79	0.54	75.22
Qwen3.8 27B	1.3	0.84	0.44	44.77

Per-model figures (Deep-SWE). Each figure pair below follows the format of its SWE-Bench-Verified counterpart. Figures 46–47 show the per-edit ΔD histogram for each model, as in Figure 12: green bars are progress edits ($\Delta D < 0$), red bars are backtracking edits ($\Delta D > 0$), and the annotation is the backtrack-edit share. Figures 48–49 show the per-cycle share of source-editing, scratch-testing and suite-testing tool calls across resolved and unresolved trajectories, as in Figure 14. Figures 50–51 show the level family, mean $\bar{D}(t)$ and backtracking term $\bar{B}(t)$ of the resolved trajectories for each model, as in Figure 16. Figures 52–53 plot the rate of *genuine* tool errors per cycle (R8 = `genuine`, see Section B.1.2), broken out by tool family (`bash` vs. `editor`), as in Figure 18; benign non-zero exits are excluded. Figures 54–55 show how each trajectory's time is split across the four R6 phases (`explore`, `localize`, `implement`, `verify`; see Table 3), as in Figure 20.

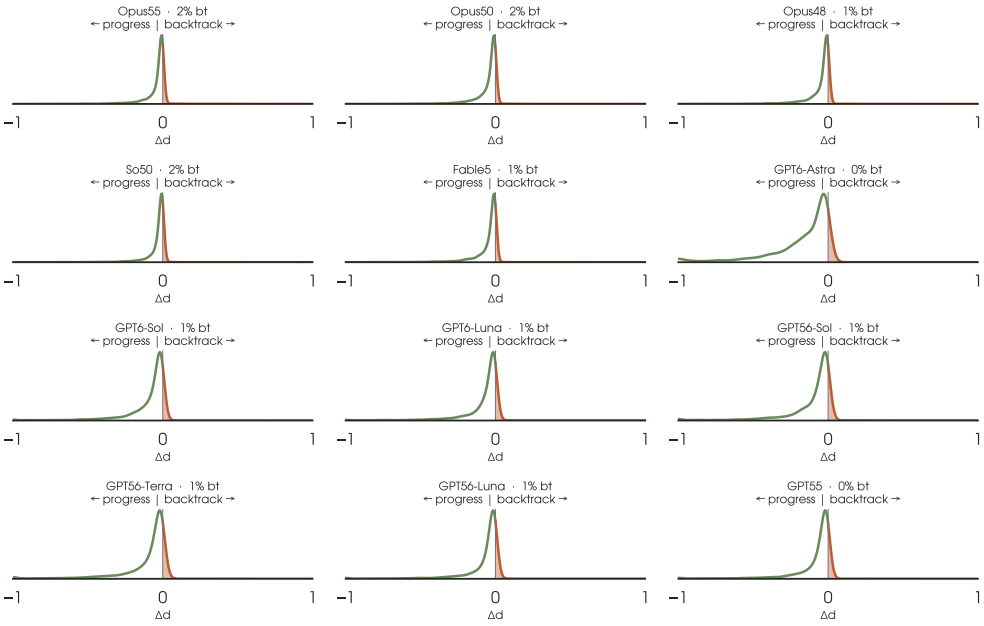


Figure 46: **Local backtracking.** Histograms of ΔD on Deep-SWE (part 1 of 2): Claude and GPT families. Each cell is one model, green = progress edits, red = local backtracking edits. Annotation shows the per-model local-backtrack share.

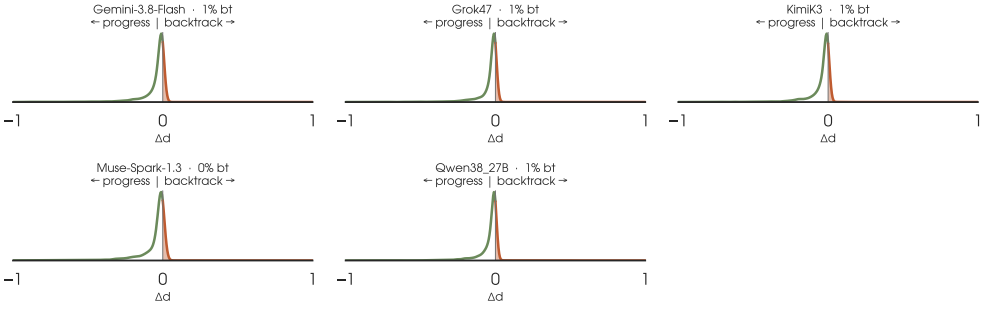


Figure 47: **Local backtracking.** Histograms of ΔD on Deep-SWE (part 2 of 2): Gemini, Grok, Kimi, Muse Spark and Qwen families. Continued from Figure 46.

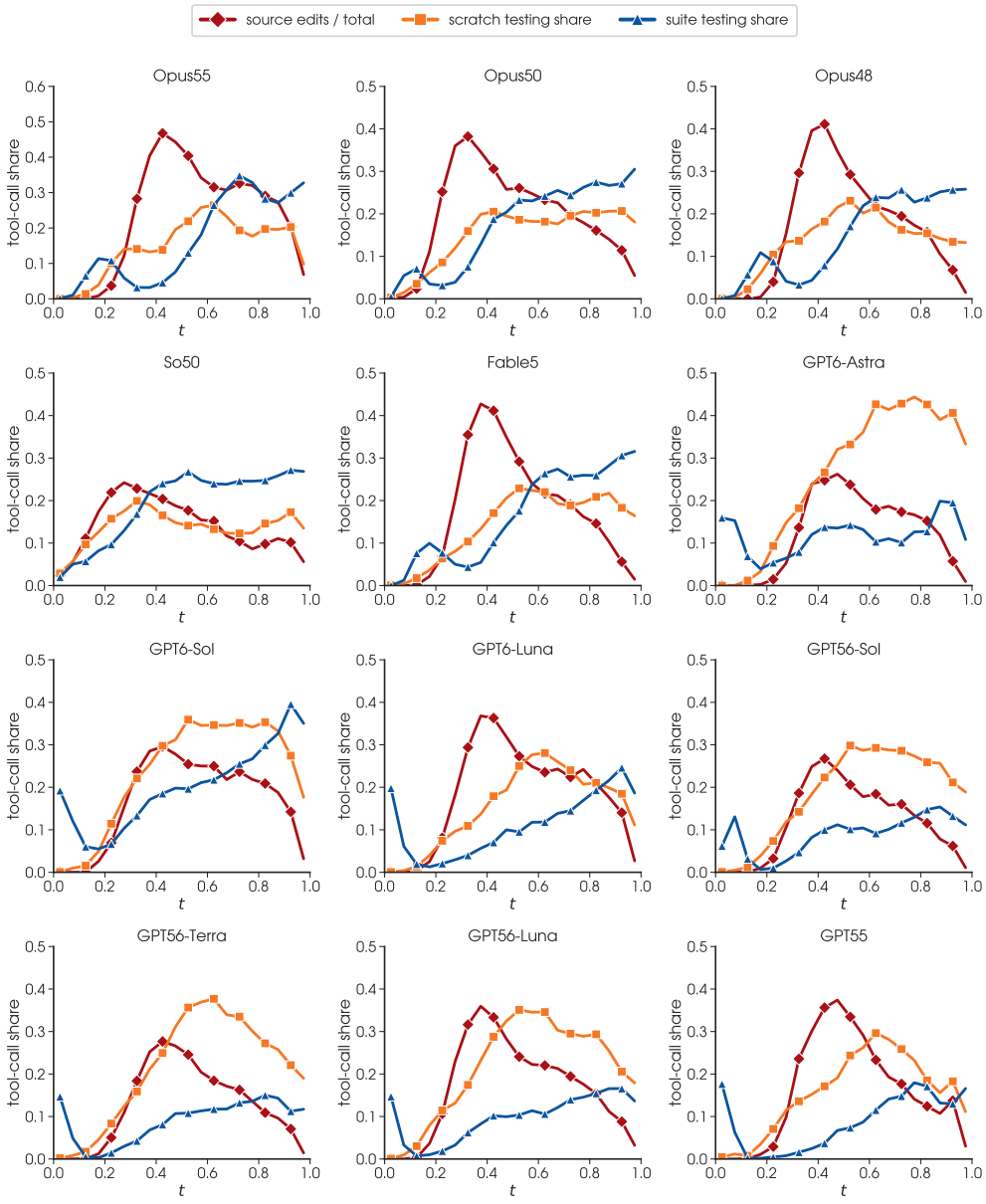


Figure 48: Edit/test ratio per cycle on Deep-SWE (part 1 of 2): Claude and GPT families. Each cell shows the model's share of source edits, scratch tests and suite tests over normalized cycle position.

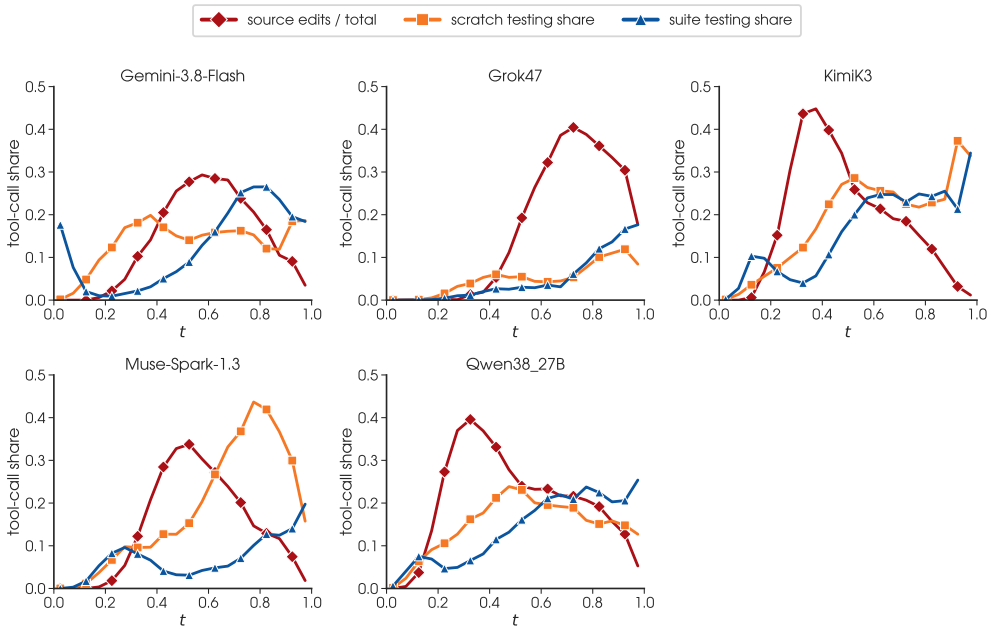


Figure 49: Edit/test ratio per cycle on Deep-SWE (part 2 of 2): Gemini, Grok, Kimi, Muse Spark and Qwen families. Continued from Figure 48.

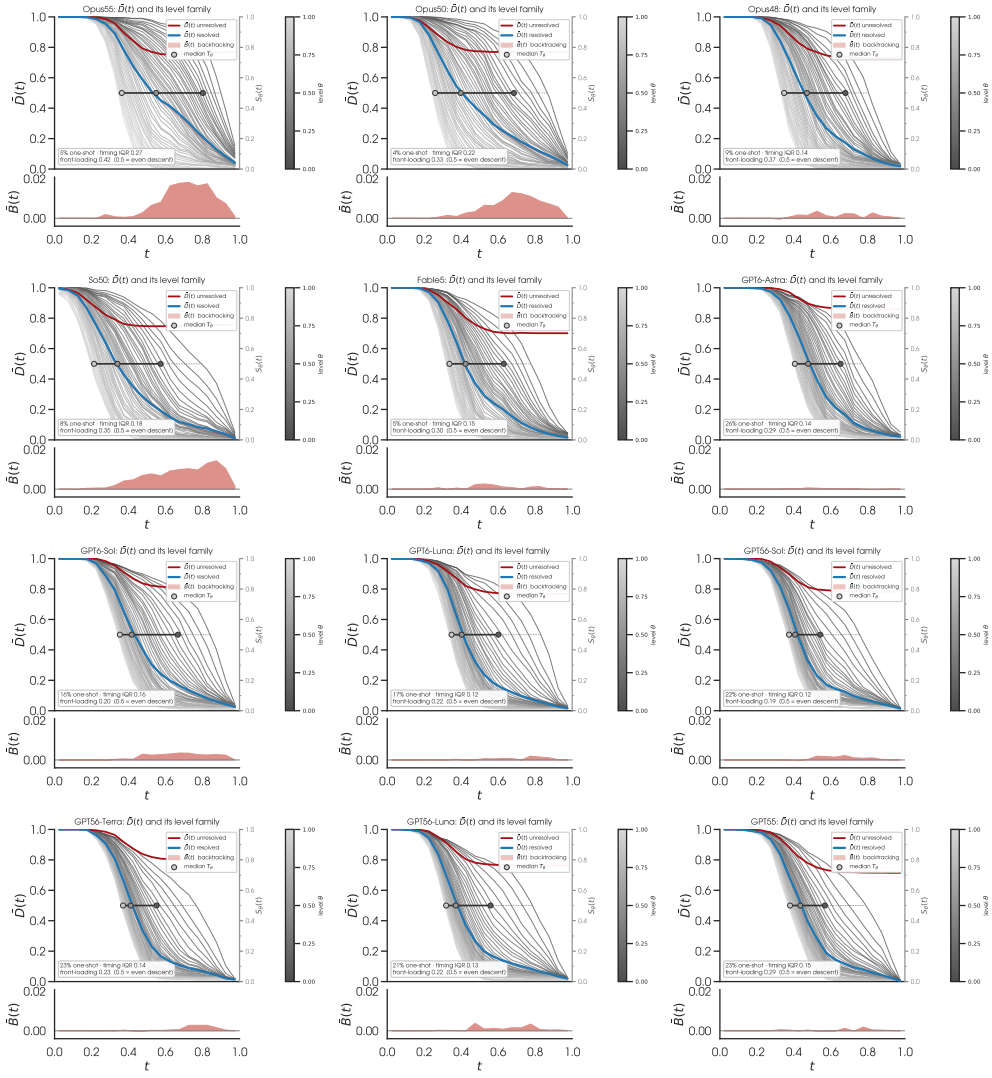


Figure 50: Level families on Deep-SWE (part 1 of 2): Claude and GPT families. Grey curves are the level sets $S_\theta(t)$ of the resolved trajectories (Section 3), the blue curve is their mean $\bar{D}(t)$, the dark-red curve is the mean $\bar{D}(t)$ of the unresolved runs (which plateaus where they stop), the markers give the median crossing time T_θ (the median over runs of the first-entry time $T_r(\theta)$, i.e. where S_θ crosses $1/2$) at $\theta = 0.5$ with its interquartile range, and the strip beneath each panel is the backtracking term $\bar{B}(t)$. The inset reports the one-shot share (runs that land the patch in a single edit); the timing IQR, the interquartile range of the runs' half-way times $T_r(0.5)$, i.e. how dispersed solve timing is across runs; and the front-loading index $(T_{0.5} - T_{0.9})/(T_{0.1} - T_{0.9})$, the position of the median half-way time between the median crossing times of levels 0.9 and 0.1 (0.5 = even descent; lower = most of the distance falls early).

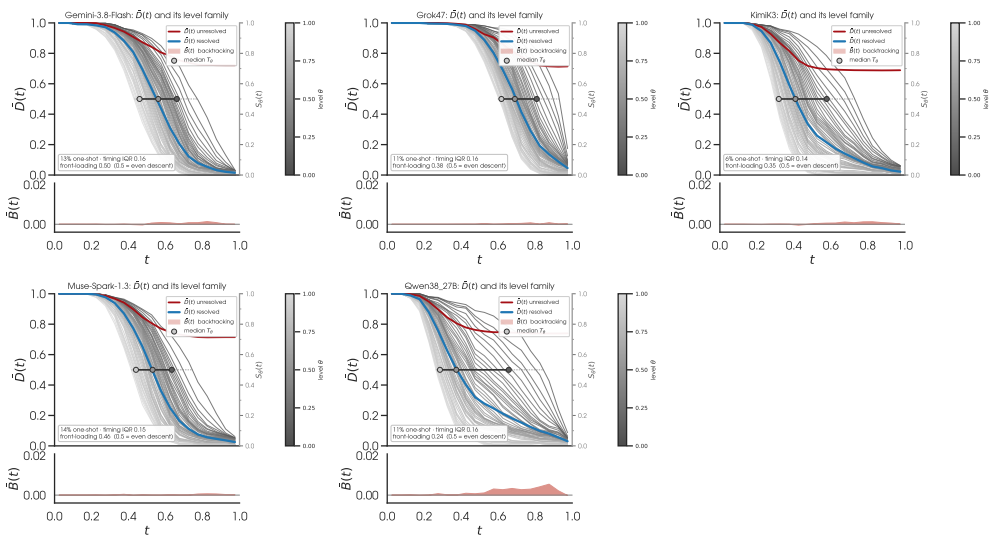


Figure 51: Level families on Deep-SWE (part 2 of 2): Gemini, Grok, Kimi, Muse Spark and Qwen families. Continued from Figure 50.

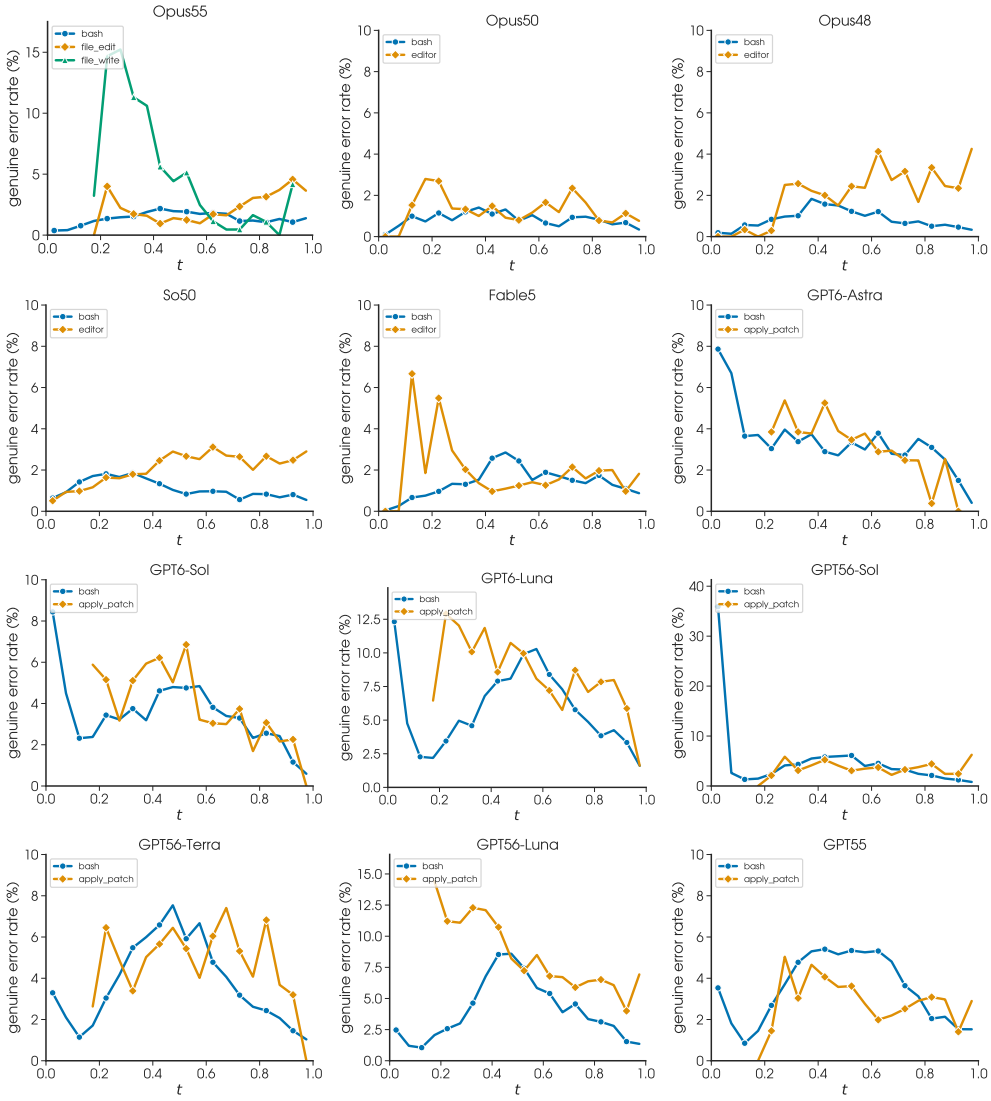


Figure 52: Genuine tool-error rate per cycle on Deep-SWE (part 1 of 2): Claude and GPT families. Benign exits (`grep` no-match, `pytest` test failures, etc.) are excluded. Tool families broken out within each sub-plot.

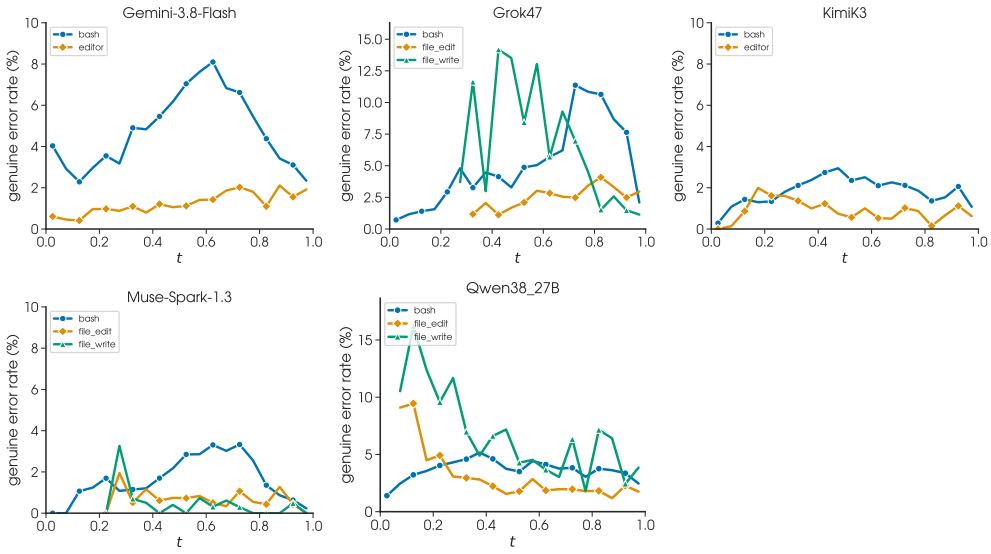


Figure 53: Genuine tool-error rate per cycle on Deep-SWE (part 2 of 2): Gemini, Grok, Kimi, Muse Spark and Qwen families. Continued from Figure 52.

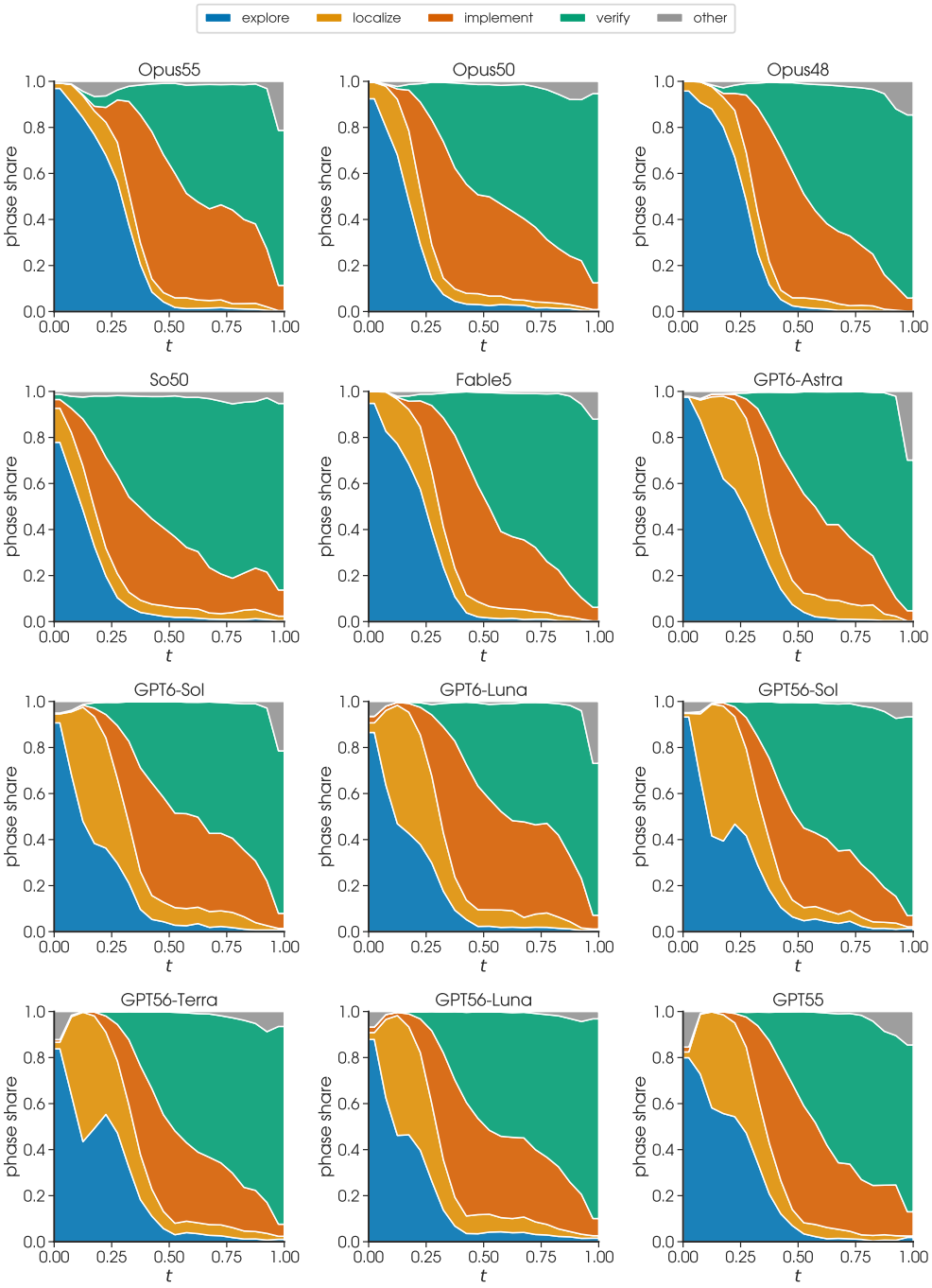


Figure 54: **Phase composition surfaces model-specific schedules.** Per-cycle share of the four R6 (see LLM judge rubrics in Table 3) phases (`explore`, `localize`, `implement`, `verify`) vs. normalised cycle position t on Deep-SWE, for Claude and GPT families (part 1 of 2). Given the same tasks, models schedule their cycles differently. The implementation peak shifts in position and width, and the verify phase carries different long-tail mass.

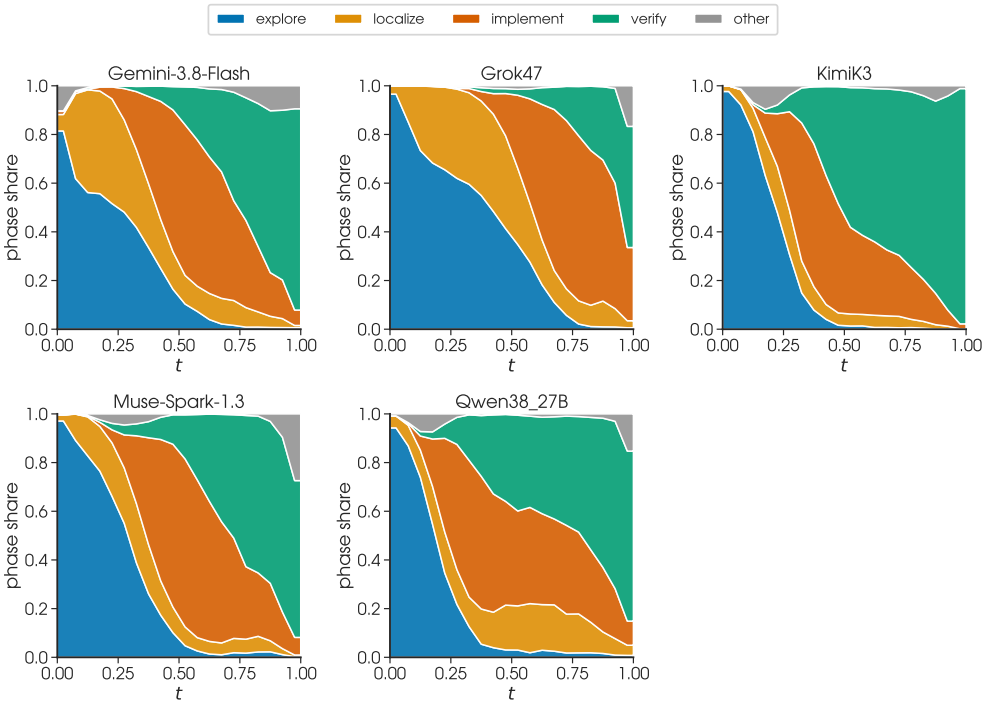


Figure 55: Phase composition on Deep-SWE (part 2 of 2): Gemini, Grok, Kimi, Muse Spark and Qwen families. Continued from Figure 54.

C PER-MODEL PASS RATES BY FAMILY

This appendix gives the full numbers summarized by Tables 1 and 2: for every model and every benchmark it was run on, SSA's five-run pass@1 with its 95% confidence interval, pass@2 where reported, and the figure the provider publishes together with its source. Tables are organized by model family, in alphabetical order, so that each can be read against its own provider's published figures; they are not intended for cross-family comparison. Terminal-Bench-2 is reported under both the Constrained setting (the benchmark's memory and time budgets) and the Unconstrained setting (budgets removed, an upper bound), with the published figure placed under the setting the provider reports. Opus 5.5, Opus 5.0, Opus 4.8, Sonnet 5.0, Fable 5 and GPT-5.5 were added after the main sweep: they were run on Deep-SWE and, where a row appears below, on SWE-Bench-Verified or SWE-Bench-Pro, but not on Terminal-Bench-2, so they do not appear in Table 1.

Table 10: Per-model pass rates, Claude family, grouped by benchmark. SSA pass@1 is the mean of five per-run resolution rates with a 95% confidence interval over all per-instance trials; pass@2 is the fraction of instances resolved by at least one of two attempts (reported for the SWE benchmarks only). "Published" is the provider's model-card figure; for Deep-SWE it is the DataCurve v1.1 artifact (`mini-swe-agent`, best reasoning-effort setting, pass@1 over four attempts) or, where marked *, the vendor's own system or model card. "—" no figure published; "n/a" not reported. Mark as in Table 1. SWE-Bench-Pro figures are from the Base public containers.

Model	SSA pass@1 (%)	pass@2 (%)	Published (%)	Mark
SWE-Bench-Verified				
Opus 5.0	96.60 ± 1.85	97.28	96.0	✓
Opus 4.8	88.80 ± 1.23	91.66	88.6	✓
Opus 4.6	80.80 ± 1.54	82.54	80.8	✓
Opus 4.5	80.48 ± 1.55	83.30	80.9	✓
Sonnet 5.0	84.04 ± 1.43	86.96	85.0	✓
Sonnet 4.6	79.00 ± 1.59	82.06	79.6	✓
Sonnet 4.5	76.40 ± 1.66	79.94	77.2	✓
Sonnet 4.0	72.00 ± 1.76	76.82	72.7	✓
Haiku 4.5	72.60 ± 1.74	77.30	73.3	✓
SWE-Bench-Pro				
Opus 5.0	79.53 ± 1.30	82.67	79.2	✓
Opus 4.8	68.48 ± 1.50	73.02	—	—
Opus 4.6	57.45 ± 1.60	63.10	53.4	✓✓
Opus 4.5	54.58 ± 1.61	58.72	52.0	✓✓
Sonnet 4.6	53.67 ± 1.61	59.83	—	—
Sonnet 4.5	50.15 ± 1.62	56.22	—	—
Sonnet 4.0	44.26 ± 1.61	51.09	—	—
Haiku 4.5	46.37 ± 1.61	53.14	—	—
Fable 5	79.12 ± 1.31	82.85	80.0	✓
Terminal-Bench-2 (Constrained)				
Opus 4.6	73.93 ± 4.08	n/a	65.4	✓✓
Opus 4.5	65.39 ± 4.40	n/a	59.8	✓✓
Sonnet 4.6	64.71 ± 4.44	n/a	59.1	✓✓
Sonnet 4.5	49.21 ± 4.63	n/a	51.0	✓
Sonnet 4.0	41.80 ± 4.56	n/a	—	—
Haiku 4.5	41.57 ± 4.56	n/a	41.0	✓
Terminal-Bench-2 (Unconstrained)				
Opus 4.6	83.14 ± 3.48	n/a	—	—
Opus 4.5	72.35 ± 4.15	n/a	—	—
Sonnet 4.6	74.15 ± 4.07	n/a	—	—
Sonnet 4.5	54.83 ± 4.62	n/a	—	—
Sonnet 4.0	42.47 ± 4.59	n/a	—	—
Haiku 4.5	43.37 ± 4.60	n/a	—	—
Deep-SWE				
Opus 5.5	74.33 ± 3.60	80.70	74.2*	✓
Opus 5.0	72.56 ± 3.68	81.77	68.8*	✓✓
Opus 4.8	61.28 ± 4.02	75.66	59.0	✓
Sonnet 5.0	57.25 ± 4.07	73.18	53.9	✓
Fable 5	68.84 ± 3.82	77.78	69.9	✓

Table 11: Per-model pass rates, Gemini family, grouped by benchmark. SSA pass@1 is the mean of five per-run resolution rates with a 95% confidence interval over all per-instance trials; pass@2 is the fraction of instances resolved by at least one of two attempts (reported for the SWE benchmarks only). "Published" is the provider's model-card figure; for Deep-SWE it is the DataCurve v1.1 artifact (`mini-swe-agent`, best reasoning-effort setting, pass@1 over four attempts) or, where marked *, the vendor's own system or model card. "—" no figure published; "n/a" not reported. Mark as in Table 1. SWE-Bench-Pro figures are from the Base public containers.

Model	SSA pass@1 (%)	pass@2 (%)	Published (%)	Mark
SWE-Bench-Verified				
Gemini 3.1 Pro	80.80 ± 1.54	84.14	80.6	✓
Gemini 3 Flash	78.32 ± 2.08	81.76	78.0	✓
SWE-Bench-Pro				
Gemini 3.1 Pro	54.00 ± 1.61	58.72	54.2	✓
Gemini 3 Flash	49.43 ± 1.62	55.40	—	—
Terminal-Bench-2 (Constrained)				
Gemini 3.1 Pro	72.13 ± 4.17	n/a	68.5	✓
Gemini 3 Flash	63.59 ± 4.47	n/a	47.6	✓✓
Terminal-Bench-2 (Unconstrained)				
Gemini 3.1 Pro	76.00 ± 4.00	n/a	—	—
Gemini 3 Flash	66.29 ± 4.39	n/a	—	—
Deep-SWE				
Gemini 3.8 Flash	73.10 ± 3.65	81.76	73.7*	✓

Table 12: Per-model pass rates, GPT family, grouped by benchmark. SSA pass@1 is the mean of five per-run resolution rates with a 95% confidence interval over all per-instance trials; pass@2 is the fraction of instances resolved by at least one of two attempts (reported for the SWE benchmarks only). "Published" is the provider's model-card figure; for Deep-SWE it is the DataCurve v1.1 artifact (`mini-swe-agent`, best reasoning-effort setting, pass@1 over four attempts) or, where marked *, the vendor's own system or model card. "—" no figure published; "n/a" not reported. Mark as in Table 1. SWE-Bench-Pro figures are from the Base public containers.

Model	SSA pass@1 (%)	pass@2 (%)	Published (%)	Mark
SWE-Bench-Verified				
GPT-5.4	80.16 ± 1.56	83.40	—	—
GPT-5.3	79.52 ± 1.58	82.20	—	—
GPT-5.2	79.28 ± 1.58	81.72	80.0	✓
GPT-5.2-codex	76.80 ± 1.65	79.98	—	—
GPT-OSS 120B	64.64 ± 1.87	71.90	62.0	✓✓
GPT-OSS 20B	59.84 ± 1.92	68.54	60.0	✓
SWE-Bench-Pro				
GPT-5.5	59.42 ± 1.60	61.79	58.6	✓
GPT-5.4	59.90 ± 1.58	64.77	57.7	✓✓
GPT-5.3	58.57 ± 1.59	62.59	56.8	✓✓
GPT-5.2	56.47 ± 1.60	60.00	56.0	✓
GPT-5.2-codex	53.87 ± 1.61	62.59	56.4	▲
GPT-OSS 120B	43.61 ± 1.60	51.10	—	—
GPT-OSS 20B	39.67 ± 1.58	48.22	—	—
Terminal-Bench-2 (Constrained)				
GPT-5.4	72.58 ± 4.02	n/a	75.1	✓
GPT-5.3	75.05 ± 4.02	n/a	77.3	✓
GPT-5.2	62.69 ± 4.49	n/a	62.2	✓
GPT-5.2-codex	68.76 ± 4.63	n/a	64.7	✓
Terminal-Bench-2 (Unconstrained)				
GPT-5.4	79.10 ± 3.78	n/a	—	—
GPT-5.3	80.67 ± 3.67	n/a	—	—
GPT-5.2	74.60 ± 4.04	n/a	—	—
GPT-5.2-codex	73.70 ± 4.07	n/a	—	—
GPT-OSS 120B	36.17 ± 4.46	n/a	—	—
GPT-OSS 20B	30.11 ± 4.26	n/a	—	—
Deep-SWE				
GPT-6-Astra	77.70 ± 3.43	81.85	74.1*	✓✓
GPT-6-Sol	76.10 ± 3.51	81.95	68.8*	✓✓
GPT-6-Luna	64.24 ± 3.95	76.01	66.6*	✓
GPT-5.6-Sol	72.56 ± 3.68	81.32	72.7*	✓
GPT-5.6-Terra	69.20 ± 3.80	77.52	69.6	✓
GPT-5.6-Luna	65.13 ± 3.93	77.25	67.2	✓
GPT-5.5	74.51 ± 3.51	85.04	67.0	✓✓

Table 13: Per-model pass rates, Grok family, grouped by benchmark. SSA pass@1 is the mean of five per-run resolution rates with a 95% confidence interval over all per-instance trials; pass@2 is the fraction of instances resolved by at least one of two attempts (reported for the SWE benchmarks only). "Published" is the provider's model-card figure; for Deep-SWE it is the vendor's own system card, marked *. "—" no figure published; "n/a" not reported. Mark as in Table 1. SWE-Bench-Pro figures are from the Base public containers.

Model	SSA pass@1 (%)	pass@2 (%)	Published (%)	Mark
SWE-Bench-Verified				
Grok 4.2	77.56 ± 1.63	80.76	—	—
SWE-Bench-Pro				
Grok 4.2	51.57 ± 1.62	57.31	—	—
Terminal-Bench-2 (Constrained)				
Grok 4.2	52.35 ± 4.64	n/a	—	—
Terminal-Bench-2 (Unconstrained)				
Grok 4.2	56.40 ± 4.61	n/a	—	—
Deep-SWE				
Grok 4.7	72.39 ± 3.68	83.18	71.0*	✓

Table 14: Per-model pass rates, Kimi family, grouped by benchmark. SSA pass@1 is the mean of five per-run resolution rates with a 95% confidence interval over all per-instance trials; pass@2 is the fraction of instances resolved by at least one of two attempts (reported for the SWE benchmarks only). "Published" is the provider's model-card figure; for Deep-SWE it is the DataCurve v1.1 artifact (`mini-swe-agent`, best reasoning-effort setting, pass@1 over four attempts) or, where marked *, the vendor's own system or model card. "—" no figure published; "n/a" not reported. Mark as in Table 1. SWE-Bench-Pro figures are from the Base public containers.

Model	SSA pass@1 (%)	pass@2 (%)	Published (%)	Mark
Deep-SWE				
Kimi-K3	77.00 ± 3.47	86.46	67.5*	✓✓

Table 15: Per-model pass rates, Muse Spark family, grouped by benchmark. SSA pass@1 is the mean of five per-run resolution rates with a 95% confidence interval over all per-instance trials; pass@2 is the fraction of instances resolved by at least one of two attempts (reported for the SWE benchmarks only). "Published" is the provider's model-card figure; for Deep-SWE it is the DataCurve v1.1 artifact (`mini-swe-agent`, best reasoning-effort setting, pass@1 over four attempts) or, where marked *, the vendor's own system or model card. "—" no figure published; "n/a" not reported. Mark as in Table 1. SWE-Bench-Pro figures are from the Base public containers.

Model	SSA pass@1 (%)	pass@2 (%)	Published (%)	Mark
Deep-SWE				
Muse Spark 1.3	75.22 ± 3.56	86.10	75.4*	✓

Table 16: Per-model pass rates, Qwen family, grouped by benchmark. SSA pass@1 is the mean of five per-run resolution rates with a 95% confidence interval over all per-instance trials; pass@2 is the fraction of instances resolved by at least one of two attempts (reported for the SWE benchmarks only). "Published" is the provider's model-card figure; for Deep-SWE it is the vendor's own model card, marked *. "—" no figure published; "n/a" not reported. Mark as in Table 1. SWE-Bench-Pro figures are from the Base public containers.

Model	SSA pass@1 (%)	pass@2 (%)	Published (%)	Mark
SWE-Bench-Verified				
Qwen3-Coder Next	72.44 ± 1.75	78.20	71.3	✓
Qwen3-Coder 480B	67.56 ± 1.83	73.66	69.6	▲
Qwen3-Coder 30B	51.36 ± 1.95	59.46	51.6	✓
Qwen3.6 35B-A3B	72.92 ± 1.74	77.76	73.4	✓
Qwen3.6 27B	76.24 ± 1.67	80.52	77.2	✓
Qwen3.5 27B	74.36 ± 1.71	79.50	72.4	✓✓
SWE-Bench-Pro				
Qwen3-Coder Next	57.75 ± 1.60	69.82	44.3	✓✓
Qwen3-Coder 480B	38.90 ± 1.58	46.42	—	—
Qwen3-Coder 30B	27.41 ± 1.44	33.96	—	—
Qwen3.6 35B-A3B	46.70 ± 1.61	56.47	49.5	▲
Qwen3.6 27B	51.40 ± 1.62	58.49	53.5	▲
Qwen3.5 27B	47.44 ± 1.61	66.06	51.2	▲
Terminal-Bench-2 (Unconstrained)				
Qwen3-Coder Next	38.87 ± 4.53	n/a	36.2	✓
Qwen3-Coder 480B	35.73 ± 4.45	n/a	—	—
Qwen3-Coder 30B	17.07 ± 3.50	n/a	—	—
Qwen3.6 35B-A3B	48.76 ± 4.64	n/a	51.5	✓
Qwen3.6 27B	56.00 ± 4.61	n/a	59.3	✓
Qwen3.5 27B	50.78 ± 4.65	n/a	41.6	✓✓
Deep-SWE				
Qwen3.8 27B	44.77 ± 4.10	61.68	42.2*	✓

D TERMINAL-BENCH-2: INFRASTRUCTURE SENSITIVITY

Terminal-Bench-2 imposes both compute limits (task memory, storage, CPU) and time limits (agent and verifier timeouts). While these limits standardize the use of compute to make benchmark scores comparable, they also have the unintended side-effect of conflating agent quality with infrastructure stability. Dependency installation, backend latency, and retries draw on the same budget as problem solving (Merrill et al., 2026; Kwon et al., 2023). From looking at the logs, we noticed two system factors that stood out. First, the inference backend's throughput (tokens-per-minute and requests-per-minute) must cover all concurrent projects for the full duration of evaluation. High invoker-latency variance, timeouts, and retries eat into the total time budget allocated for problem-solving and reduce the resolution rate. Second, packing multiple projects onto one node (e.g. Harbor with `n_concurrent` > 1) splits available network bandwidth across them, slowing the dependency downloads required by nearly every task and leaving less time before the agent is interrupted.

Command batching interacts with each family's reasoning budget differently. This split is a direct consequence of the reasoning/tool-use relation. Since most tool calls are shell commands, an obvious mitigation of high interaction overhead due to sequential command execution is a batch interface that executes several commands per turn. The effect is mixed. Batching forces the model to maintain a coherent terminal state across steps, which raises reasoning complexity. For Claude, batching shifts effort from round-trips into in-context reasoning; for Gemini and Grok, batch execution helps because it does not trigger additional reasoning. Under the constrained setting, batching is therefore not a uniform win, and it mirrors the failure mode of Section 2: the right operating point on the reasoning-versus-interaction trade-off is family-specific, not global.

Relaxing the time and memory limits raises absolute pass@1 by 5**–**10 **points.** We interpret the Unconstrained setting (Appendix C) as an upper-bound of model performance on Terminal-Bench-2. Across models, removing the memory and agent-time limits lifts pass@1 by about 5–10 absolute points, with most of the gains coming from a handful of projects that repeatedly time out under the constraints but solve reliably without them — `make-doom-for-mips`, `torch-pipeline-parallelism`, `gpt2-codegolf`, `caffe-cifar-10`, and `train-fasttext`.

E EVALUATION INTEGRITY

The public SWE-Bench-Pro containers leak future git history. Distinct from training-data contamination (Magar & Schwartz, 2022; Sainz et al., 2023; Jain et al., 2024), the public images leave post-`base_commit` objects reachable even though `HEAD` is checked out at the instance's `base_commit`. An agent can recover the intended fix through ordinary commands such as `git log -p` or `git show <future-sha>`. We compare the public *Base* containers against *Sanitized* ones whose working tree is identical but whose future commits and dev-branches are pruned before the agent starts. The containers expose the fix to any agent that inspects git history in the course of ordinary exploration, so removing them lowers measured pass@1 most for the models whose exploration includes git history: Opus 4.6 by 6.87 points ($57.45\% \rightarrow 50.58\%$, 4.67% of instances including a `git show` of the gold-patch commit), GPT-5.4 by 3.32 points (5.33% of instances with `git show`), and Sonnet 4.6 by 3.66 points. Other models are flat or move within trajectory noise, confirming the leakage channel exists but is exercised unevenly. In an ablation where the model is explicitly instructed to "inspect git history" in its system prompt, leakage effect is significantly larger: it raises Qwen3.6 27B by $+17.13\%$ (compared to sanitized) and Qwen3-Coder Next to a $+21.12\%$ (compared to sanitized). We, therefore, treat Sanitized as the cleaner estimate of model ability and recommend that SWE-Bench-Pro comparisons either use Sanitized scores or explicitly report leakage exposure. The sanitization protocol and full per-model Base-Sanitized table are in Appendix F.

A few residual evaluation artifacts cap the achievable ceiling. On SWE-Bench-Verified, instances such as `astropy-8872` and `astropy-8707` fail even when the agent's patch is correct, because of setup inconsistencies that require fixes to the evaluation environment; some `psf_requests` instances fail intermittently on nonresponsive external test URLs and need manual patching for reliable scoring (Jimenez et al., 2024). On SWE-Bench-Pro, run on Amazon ECS[2], 3 of 731 instances consistently fail under AWS-specific environment assumptions, and an approximate 0.41% ceiling loss is suffered uniformly across all SSA runs.

F SWE-BENCH-PRO GIT LEAKAGE

The public SWE-Bench-Pro (Deng et al., 2025) containers are checked out at the instance's `base_commit`, but they can still retain future git objects, branches, tags, or reflogs. This creates a runtime leakage channel, i.e., an agent can inspect repository history during evaluation and place post-`base_commit` information into its context. This differs from training-data contamination (Magar & Schwartz, 2022; Sainz et al., 2023; Jain et al., 2024) as the model need not memorize a solution if the benchmark environment itself can reveal it through ordinary commands such as `git log -p`, `git branch -a`, or `git show <future-sha>`.

F.1 SANITIZED EVALUATION

For a SWE-Pro instance with cutoff point `base_commit`, a temporally isolated environment should expose only the checked-out source tree, tests, dependencies, and repository history up to that commit. We therefore compare two regimes. **Base** uses the official public container as released: `HEAD` is at `base_commit`, but future git state may remain reachable. **Sanitized** keeps the working tree identical to Base while removing future history before the agent starts: remotes and refs are removed and a single local `main` branch is recreated at the original `HEAD`.

The main impact statistic is **Base–Sanitized Pass@1**. If future information does not matter to the agent, removing it should not systematically reduce Pass@1 beyond ordinary trajectory variance. If the agent benefited from future information, Base accuracy should degrade under Sanitized evaluation.

F.2 DIRECT LEAKAGE METRIC

Let $\mathcal{I}_{\text{SBP}}$ be the 731 public SWE-Pro instances, and let s_i be the gold fix commit for instance i. For each model, we count a *confirmed direct leak* when a resolved Base trajectory contains `git show` applied to s_i in the agent context before termination. Let L_i be the leakage indicator that is exactly 1

[2]https://aws.amazon.com/ecs/

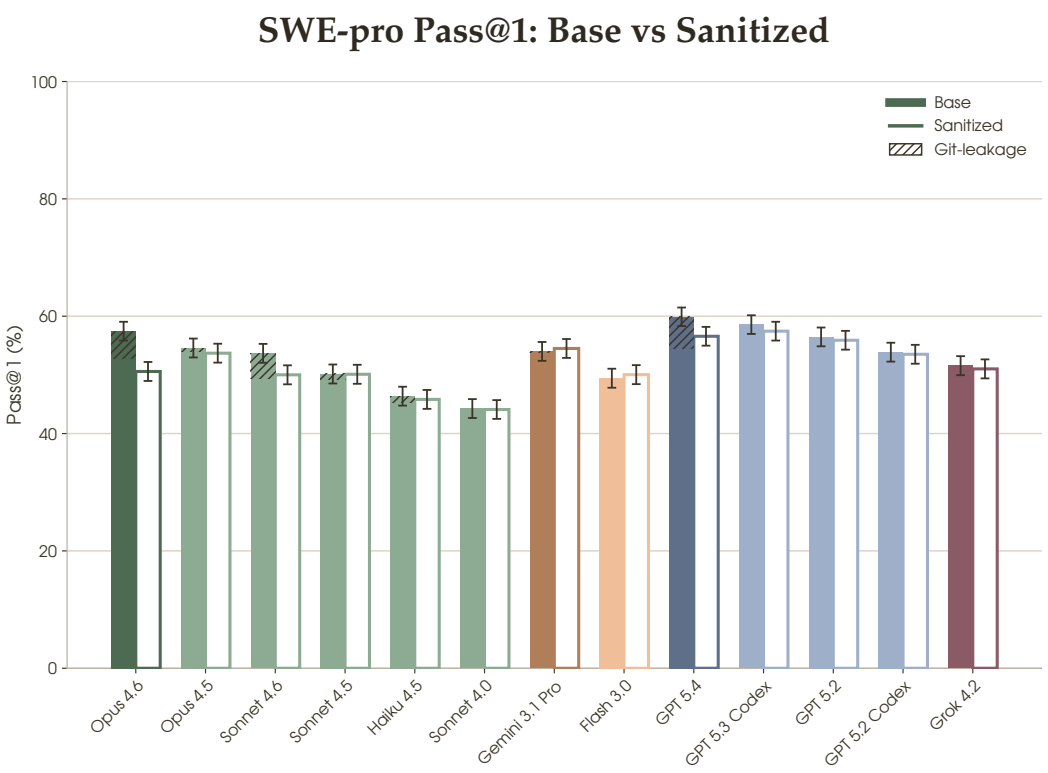


Figure 56: SWE-Pro Pass@1 under Base and Sanitized containers. The shaded overlay is the confirmed direct leakage fraction: resolved Base instances whose agent context contains `git show <gold-patch-sha>`, normalized by all 731 SWE-Bench-Pro instances.

when the context contains `git show <gold-patch-sha>` for that instance AND the run solves the Base task. The reported direct leakage percentage is

$$\text{DirectLeakage} = \mathbb{E}[L_i].$$

The empirical expectation is over all public SWE-Bench-Pro instances (731), so the number is directly comparable to Pass@1.

This metric is intentionally a lower bound. It captures only exact-SHA exposure. It misses future information obtained through `git log -p`, `git diff` against branches or tags, `git blame`, merge commits, cherry-picks, release commits, nearby commits that contain the same edit, follow-up commits that reveal the intended invariant, and commit messages that localize the fix without exposing the exact gold SHA. Thus, direct leakage is confirmed evidence of exposure, not an estimate of all possible leakage. The Base minus Sanitized Pass@1 gap is the broader aggregate measure.

F.3 Empirical impact

Figure 56 shows that sanitization can materially change measured SWE-Bench-Pro performance. Opus 4.6 drops from 57.45% Pass@1 in Base to 50.58% under Sanitized evaluation, a 6.87% decrease, with 4.67% of its Base instances include a `git show` of the gold-patch commit. GPT 5.4 drops from 59.90% to 56.58% and has the largest direct leakage fraction, 5.33%. Sonnet 4.6 also shows a clear signal: 53.67% Base, 50.01% Sanitized, and 4.32% confirmed direct leakage.

Table 17 gives the numerical view. The gap and the direct leakage fraction are related but not interchangeable. The gap can be larger than direct leakage because the metric misses indirect and nearby-commit leakage. It can also be smaller because an exact `git show` exposure may not be necessary for the final solve, or the agent may not realize to overwrite its current solution with the exposed one. Near-zero gaps or small Sanitized improvements, such as Gemini 3.1 Pro and Flash 3.0,

Table 17: SWE-Pro Base and Sanitized Pass@1, with a fraction of instances in which a Base run accessed the gold-patch commit. Leakage is counted when a resolved Base trajectory contains `git show <gold-patch-sha>` in the agent context; percentages use all 731 SWE-Pro instances as the denominator.

Model	Base %	Sanitized %	Gap %	Direct leakage %
Opus 4.6	57.45	50.58	6.87	4.67
Opus 4.5	54.58	53.70	0.88	0.65
Sonnet 4.6	53.67	50.01	3.66	4.32
Sonnet 4.5	50.15	50.10	0.05	0.90
Haiku 4.5	46.37	45.82	0.55	1.09
Sonnet 4.0	44.26	44.10	0.16	0.00
Gemini 3.1 Pro	54.00	54.50	-0.50	0.21
Flash 3.0	49.43	50.04	-0.61	0.00
GPT 5.4	59.90	56.58	3.32	5.33
GPT 5.3 Codex	58.57	57.45	1.12	0.00
GPT 5.2	56.47	55.90	0.57	0.00
GPT 5.2 Codex	53.87	53.51	0.36	0.00
Grok 4.2	51.57	51.02	0.55	0.00

Table 18: Qwen models on SWE-Bench-Pro under Base, Git Instructions, and Sanitized regimes. Pass@1 cells are percentages with bootstrap 95% confidence intervals where available; leakage columns are percentage-point estimates attributable to git leakage.

Model	Base (%)	Base leak. (%)	Git Instr. (%)	Git leak. (%)	Sanitized (%)
Qwen3-Coder Next	57.75 ± 1.60	18.11	64.51 ± 1.55	27.60	43.39 ± 1.60
Qwen3.5 27B	47.44 ± 1.61	0.01	55.51 ± 1.61	11.79	47.68 ± 1.61
Qwen3.6 27B	51.40 ± 1.62	2.60	66.81 ± 1.52	23.14	49.68 ± 1.62
Qwen3.6 35B-A3B	46.70 ± 1.61	1.17	48.91 ± 1.62	4.02	45.22 ± 1.61

are best read as ordinary trajectory variance. The central finding is that Base containers expose a real future-information channel, and its measured effect depends on each model's tool-use behavior.

F.4 Open-source models and explicit git instructions

The results above use each model's default instructions (prompts and tool spec). To stress-test the channel, we also evaluate four Qwen models with **Git Instructions**, a prompt variant that explicitly asks the agent to inspect git history while debugging. This regime estimates how much Pass@1 can increase when the leakage path is made salient rather than discovered naturally.

The Git Instructions experiment appends the following block to the agent's system prompt, directing it to consult version-control history before editing:

Git Instructions (appended to system prompt)

```
Additional instructions:
- Before editing ANY file <file>, you MUST first run ‘git log -p -- <file>‘ AND ‘git
blame <file>‘. Reading history in your head does not count.
- For each symbol named in the <pr_description>, you MUST run ‘git log -S’<symbol>’
--all‘ to find when/why it was introduced or changed.
- If a git command returns no useful history, state that explicitly and proceed --
do not silently skip the call.
```

Figure 57 and Table 18 show the expected pattern. Explicit git instructions raise Pass@1 for Qwen3.6 27B by 17.13% and for Qwen3.5 27B by 7.83%. Qwen3-Coder Next also gains 21.12% under Git Instructions, carries the largest Git-Instructions leakage estimate (27.60%), and drops to 43.39% after sanitization. Across all four models, Sanitized scores are below Git Instructions scores, confirming that future git information can account for a substantial part of measured SWE-Bench-Pro performance.

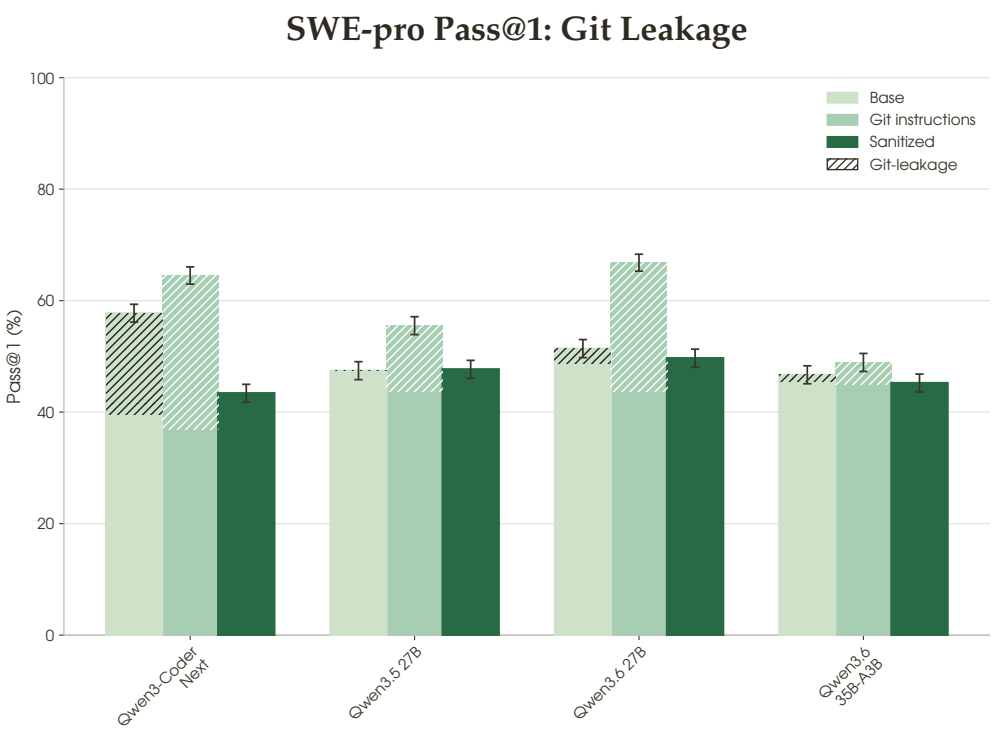


Figure 57: SWE-Bench-Pro Pass@1 for Qwen models under Base, Git Instructions, and Sanitized regimes. Explicit git instructions raise Pass@1 for all four models; Sanitized scores remove future git history while keeping the working tree fixed.

For reporting, Sanitized is the cleaner estimate of SWE-Bench-Pro repair ability because it preserves the checked-out task state while removing future repository states that should be unobservable at the benchmark cutoff point. Base scores are still useful for documenting the public containers as released, but model comparisons should either use Sanitized results or explicitly report git-leakage exposure.

G CONFIGURATIONS

All runs use an AWS PCS[3] cluster of `c7.48xlarge` instances with a maximum concurrency of 10. Claude models are served through Amazon Bedrock[4] at production capacity; OpenAI, Gemini, and Grok models are accessed through their commercial APIs. Open-weights models are self-served via vLLM (Kwon et al., 2023) (see exact serving commands in Appendix J). We use vendor-recommended high-reasoning settings wherever a model card documents them (e.g. `adaptive` thinking at `max` effort for the Claude 4.x family, reasoning effort `xhigh` for the GPT-5 family, thinking level `high` for Gemini); the complete per-model inference settings, provider clients, prompt tags, bound tools, and reasoning controls for every benchmark configuration are summarized below, and shipped with the open-sourced SSA package for reproducibility.

We enforce strict evaluation hygiene. Internet access is disabled for SWE-Bench-Verified and SWE-Bench-Pro and enabled for Terminal-Bench-2, as its tasks require it. For both SWE benchmarks we use the standard Docker environments, whose working trees are checked out at the instance's `base_commit` and are *intended* to expose only history up to that revision (Jimenez et al., 2024; Deng et al., 2025; Magar & Schwartz, 2022; Sainz et al., 2023; Jain et al., 2024). Because the public SWE-Bench-Pro images can nonetheless retain reachable future git objects, we separately audit and sanitize that temporal-leakage channel in Appendices E and F.

[3] https://aws.amazon.com/pcs/

[4] https://aws.amazon.com/bedrock/

The repository ships 80 benchmark configurations spanning the four benchmarks reported in Section 4: SWE-Bench-Verified (SBV; 21 configs), SWE-Bench-Pro (SBP; 21 configs), Terminal-Bench-2 (TB2; 21 configs) and Deep-SWE (17 configs). Each config is a Hydra-mergeable (Yadan, 2019) YAML that fixes the model identifier, the invoker (provider client), the prompt tag (resolved against `SimpleResolverAgent.yaml`, see Section H), the bound tools, and the reasoning controls.

The summaries below report the raw values stored in `src/ssa/configs/`. In summary, For Anthropic models, `thinking=enabled (Nk)` gives the extended-thinking token budget, `thinking=adaptive` is the adaptive-thinking mode, `interleaved` indicates the `interleaved-thinking-2025-05-14` beta header, and `effort` is `output_config.effort` (the `effort-2025-11-24` beta). `level=HIGH` is Gemini's `thinking_config.thinking_level`; `effort=xhigh` is OpenAI's `reasoning.effort`. For the open-weights (vLLM) models, `temperature`, `top_p`, `top_k`, `max_tokens` and `rep_penalty` are the sampling parameters. `defaults` marks a config that sets no sampling or reasoning overrides. Configs not listed inherit defaults from `default.yaml` (`max_iterations = 500`; `retry_on_no_change.max_attempts = 1`; `timeout = 200s` for SWE-Bench and Deep-SWE, `120s` for Terminal-Bench-2. Internet access disabled for SWE-Bench and Deep-SWE, enabled for Terminal-Bench-2 except where noted.

G.1 SWE-BENCH-VERIFIED

Table 19: SBV configurations (21).

Model	Invoker	Inference config	Prompt tag
claude-haiku-4-5-20251001-v1:0	bedrock	thinking=enabled (128k), interleaved	Anthropic_v2
claude-opus-4-5-20251101-v1:0	bedrock	effort=high	Anthropic_v2
claude-opus-4-6-v1	bedrock	thinking=adaptive, effort=max	Anthropic_v2
claude-sonnet-4-20250514-v1:0	bedrock	thinking=enabled (200k), interleaved	swe_generic_v1
claude-sonnet-4-5-20250929-v1:0	bedrock	thinking=enabled (200k), interleaved	Anthropic_v2
claude-sonnet-4-6	bedrock	thinking=adaptive, effort=high	Anthropic_v2
gemini-3-flash-preview	gemini	temp=1.0, level=HIGH	swe_generic_v2
gemini-3.1-pro-preview	gemini	temp=1.0, level=HIGH	swe_generic_v2
gpt-5.2	openai	effort=xhigh	swe_generic_v2_gpt
gpt-5.2-codex	openai	effort=xhigh	swe_generic_v2_gpt
gpt-5.3-codex	openai	effort=xhigh	swe_generic_v2_gpt
gpt-5.4	openai	effort=xhigh	swe_generic_v2_gpt
gpt-oss-120b	openai	temp=1.0, top_p=1.0, effort=high	swe_generic_v2_gpt
gpt-oss-20b	openai	temp=1.0, top_p=1.0, max_tokens=16,384, effort=high	swe_generic_v2_gpt
grok-4.20-reasoning	xai	defaults	swe_generic_v3
qwen3-coder-30b-a3b-instruct	openai	temp=0.7, top_p=0.8, top_k=20, rep_penalty=1.05, max_tokens=65,536	swe_generic_think_v2
qwen3-coder-480b-a35b-instruct	openai	temp=0.7, top_p=0.8, top_k=20, rep_penalty=1.05, max_tokens=65,536	swe_generic_think_v2
qwen3-coder-next	openai	defaults	swe_generic_think_v1
qwen3.5-27b	openai	temp=1.0, top_p=0.95, top_k=20, thinking=disabled	swe_generic_think_v1
qwen3.6-27b	openai	temp=1.0, top_p=0.95, top_k=20, thinking=enabled	swe_generic_think_v1
qwen3.6-35b-a3b	openai	temp=1.0, top_p=0.95, top_k=20, thinking=enabled	swe_generic_think_v1

G.2 SWE-BENCH-PRO

Table 20: SBP configurations (21).

Model	Invoker	Inference config	Prompt tag
claude-haiku-4-5-20251001-v1:0	bedrock	thinking=enabled (128k), interleaved	swe_generic_v2
claude-opus-4-5-20251101-v1:0	bedrock	effort=high	swe_generic_v2
claude-opus-4-6-v1	bedrock	thinking=adaptive, effort=max	swe_generic_v2
claude-sonnet-4-20250514-v1:0	bedrock	thinking=enabled (200k), interleaved	swe_generic_v2
claude-sonnet-4-5-20250929-v1:0	bedrock	thinking=enabled (200k), interleaved	swe_generic_v2
claude-sonnet-4-6	bedrock	thinking=adaptive, effort=max	swe_generic_v2
gemini-3-flash-preview	gemini	temp=1.0, level=HIGH	swe_generic_v2
gemini-3.1-pro-preview	gemini	temp=1.0, level=HIGH	swe_generic_v2
gpt-5.2	openai	effort=xhigh	swe_generic_v2_gpt
gpt-5.2-codex	openai	effort=xhigh	swe_generic_v2_gpt
gpt-5.4	openai	effort=xhigh	swe_generic_v2_gpt
gpt-oss-120b	openai	max_tokens=16,384, effort=high	swe_generic_v2_gpt
gpt-oss-20b	openai	max_tokens=16,384, effort=high	swe_generic_v2_gpt
grok-4.20-0309-reasoning	xai	defaults	swe_generic_v2
qwen3-coder-30b-a3b-instruct	openai	temp=0.7, top_p=0.8, top_k=20, rep_penalty=1.05, max_tokens=65,536	swe_generic_v2
qwen3-coder-480b-a35b-instruct	openai	temp=0.7, top_p=0.8, top_k=20, rep_penalty=1.05, max_tokens=65,536	swe_generic_v2
qwen3-coder-next	openai	temp=1.0, top_p=0.95, top_k=40	swe_generic_v2
qwen3.5-27b	openai	temp=1.0, top_p=0.95, thinking=enabled	swe_generic_v2
qwen3.6-27b	openai	temp=1.0, top_p=0.95, thinking=enabled	swe_generic_v2
qwen3.6-35b-a3b	openai	temp=1.0, top_p=0.95, thinking=enabled	swe_generic_v2

G.3 TERMINAL-BENCH-2

Table 21: TB2 configurations (21).

Model	Invoker	Inference config	Prompt tag
claude-haiku-4-5-20251001-v1:0	bedrock	thinking=enabled (128k), interleaved	tb2
claude-opus-4-5-20251101-v1:0	bedrock	thinking=enabled (128k), interleaved, effort=high	tb2
claude-opus-4-6-v1	bedrock	thinking=adaptive, effort=max	tb2
claude-sonnet-4-20250514-v1:0	bedrock	thinking=enabled (200k), interleaved	tb2
claude-sonnet-4-5-20250929-v1:0	bedrock	thinking=enabled (200k), interleaved	tb2
claude-sonnet-4-6	bedrock	thinking=disabled	tb2
gemini-3-flash-preview	gemini	temp=1.0, level=HIGH	tb2_gemini
gemini-3.1-pro-preview	gemini	temp=1.0, level=HIGH	tb2_gemini
gpt-5.2	openai	effort=xhigh	tb2_gpt
gpt-5.2-codex	openai	effort=xhigh	tb2_gpt
gpt-5.3-codex	openai	effort=xhigh	tb2_gpt
gpt-5.4	openai	effort=xhigh	tb2_gpt
gpt-oss-120b	openai	temp=1.0, top_p=1.0, effort=high	tb2_gpt
gpt-oss-20b	openai	temp=1.0, top_p=1.0, max_tokens=16,384, effort=high	tb2_gpt
grok-4.20-reasoning	xai	defaults	tb2
qwen3-coder-30b-a3b-instruct	openai	temp=0.7, top_p=0.8, top_k=20, rep_penalty=1.05, max_tokens=65,536	tb2
qwen3-coder-480b-a35b-instruct	openai	temp=0.7, top_p=0.8, top_k=20, rep_penalty=1.05, max_tokens=65,536	tb2
qwen3-coder-next	openai	temp=1.0, top_p=0.95, top_k=40	tb2
qwen3.5-27b	openai	temp=1.0, top_p=0.95, max_tokens=80,000, top_k=20, thinking=enabled	tb2
qwen3.6-27b	openai	temp=1.0, top_p=0.95, max_tokens=80,000, thinking=enabled	tb2_q36
qwen3.6-35b-a3b	openai	temp=1.0, top_p=0.95, max_tokens=80,000, thinking=enabled	tb2_q36

G.4 DEEP-SWE

Table 22: Deep-SWE configurations (17).

Model	Invoker	Inference config	Prompt tag
`claude-fable-5`	bedrock	thinking=adaptive, effort=max	`swe_generic_v2_deepswe`
`claude-opus-4-8`	bedrock	thinking=adaptive, effort=max	`swe_generic_v2_deepswe`
`claude-opus-5`	bedrock	thinking=adaptive, effort=max	`swe_generic_v2_deepswe`
`claude-opus-5-5`	bedrock	thinking=adaptive, effort=max	`swe_generic_v2_deepswe`
`claude-sonnet-5`	bedrock	thinking=adaptive, effort=max	`swe_generic_v2_deepswe`
`gemini-3.8-flash`	gemini	temp=1.0, level=HIGH	`swe_generic_v2_deepswe`
`gpt-5.5`	openai	effort=xhigh	`swe_generic_v2_deepswe_gpt`
`gpt-5.6-luna`	openai	effort=max	`swe_generic_v2_deepswe_gpt`
`gpt-5.6-sol`	openai	effort=max	`swe_generic_v2_deepswe_gpt`
`gpt-5.6-terra`	openai	effort=max	`swe_generic_v2_deepswe_gpt`
`gpt-6-astra`	bedrock	effort=max	`swe_generic_v2_deepswe_gpt`
`gpt-6-luna`	bedrock	effort=max	`swe_generic_v2_deepswe_gpt`
`gpt-6-sol`	bedrock	effort=max	`swe_generic_v2_deepswe_gpt`
`grok-4.7`	xai	defaults	`swe_generic_v2_deepswe`
`kimi-k3`	openai	max_tokens=32,768, effort=max	`swe_generic_v2_deepswe`
`muse-spark-1.3`	openai	effort=max	`swe_generic_v2_deepswe`
`qwen3.8-27b`	openai	temp=1.0, top_p=0.95, max_tokens=32,768, thinking=enabled	`swe_generic_v2_deepswe`

Model identifiers, invokers and inference settings are read from the resolved configuration logged by each of the five pinned runs per model. The `openai` invoker for GPT-5.5 and GPT-5.6 uses the Responses API. Qwen3.8 27B uses a shorter conversation window (`win_len = 400`, against the default 800) with LLM-summarised context compaction, to stay within its 262,144-token serving limit (Appendix J).

H PROMPTS

This appendix lists the system and user prompt templates shipped with SSA (`src/ssa/prompts/SimpleResolverAgent.yaml`). Each template is rendered with Jinja2, the placeholders `{{project_path}}` and `{{git_issue}}` are filled in at runtime from the benchmark instance. Each prompt is identified by a `prompt_tag` that configs reference (e.g. `prompt_tag: swe_generic_v2`).

H.1 GENERIC SWE PROMPTS

The generic prompts for SWE Pull Request (PR) tasks are inspired from the SWE-Agent work (Yang et al., 2024a). These prompts target SWE-Bench-Verified and SWE-Bench-Pro and are largely model-family-agnostic with few changes such as model-specific tool-use instructions and reasoning nudge.

PYTHON-SPECIFIC GIT PR PROMPT

The prompt is tagged as `swe_generic_v1` in SSA. Most useful for SWE-Bench-Verified.

swe_generic_v1

```
System:

You are a helpful assistant that can interact with a computer to solve tasks.

Your task is to make the minimal changes to non-tests files in the {{project_path}}
directory to ensure the <pr_description> is satisfied.

Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to explore the repo to familiarize
yourself with its structure.
2. Create a script to reproduce the error and execute it with ‘python <filename.py>‘
using the BashTool, to confirm the error
3. Edit the sourcecode of the repo to resolve the issue
4. Rerun your reproduce script and confirm that the error is fixed!
5. Think about edgecases and make sure your fix handles them as well

Your thinking should be thorough and so it’s fine if it’s very long.

User:

<uploaded_files>
{{project_path}}
</uploaded_files>
I’ve uploaded a python code repository in the directory {{project_path}} (not in
/tmp/inputs). Consider the following PR description:

<pr_description>
{{git_issue}}
</pr_description>

Can you help me implement the necessary changes to the repository so that the
requirements specified in the <pr_description> are met?
I’ve already taken care of all changes to any of the test files described in the
<pr_description>. This means you DON’T have to modify the testing logic or any of
the tests in any way!
```

LANGUAGE AGNOSTIC GIT PR PROMPT

The prompt is tagged as `swe_generic_v2` in SSA. The instructions are exactly same as `swe_generic_v1` with removal of any python-specific mentions. It is most useful for SWE-Bench-Pro like benchmarks.

swe_generic_v2

```
System:

You are a helpful assistant that can interact with a computer to solve tasks.

Your task is to make the minimal changes to non-tests files in the {{project_path}}
directory to ensure the <pr_description> is satisfied.

Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to explore the repo to familiarize
yourself with its structure.
2. Create a script to reproduce the error and execute it using the BashTool, to
confirm the error
3. Edit the sourcecode of the repo to resolve the issue
4. Rerun your reproduce script and confirm that the error is fixed!
5. Think about edgecases and make sure your fix handles them as well

Your thinking should be thorough and so it's fine if it's very long.

User:

<uploaded_files>
{{project_path}}
</uploaded_files>
I've uploaded a code repository in the directory {{project_path}} (not in
/tmp/inputs). Consider the following PR description:

<pr_description>
{{git_issue}}
</pr_description>

Can you help me implement the necessary changes to the repository so that the
requirements specified in the <pr_description> are met?
I've already taken care of all changes to any of the test files described in the
<pr_description>. This means you DON'T have to modify the testing logic or any of
the tests in any way!
```

GIT PR PROMPT WITH AN EMPHASIS ON RUNNING TESTS

Adds explicit, repeated emphasis on running the existing test suite and warns that hidden tests exist beyond what is visible.

swe_generic_v3

```
System:

You are a helpful assistant that can interact with a computer to solve tasks.

Your task is to make the minimal changes to non-tests files in the {{project_path}}
directory to ensure the <pr_description> is satisfied.

Follow these steps to resolve the issue:
1. Explore the repo to familiarize yourself with its structure.
2. Create a script to reproduce the error and execute it using the BashTool, to
confirm the error.
3. Edit the sourcecode of the repo to resolve the issue.
4. Rerun your reproduce script and confirm that the error is fixed.
5. Run the existing test suite for the affected module. If any test fails, diagnose
the failure and fix your implementation.
6. Think about edgecases and make sure your fix handles them as well.

IMPORTANT rules:
- You MUST run the tests frequently, and verify correctness of changes by running
relevant tests. If tests fail, analyze failures and revise your patch.
- Failing to test sufficiently rigorously is the NUMBER ONE failure mode.
- There are hidden tests beyond what is visible in the repository.
- Iterate until the root cause is fixed and all tests pass. Do NOT end without a
working solution.
- Do not assume the task is complete just because the visible tests pass; continue
refining until you are confident the fix is robust and comprehensive.

Your thinking should be thorough and so it's fine if it's very long.

User:

<uploaded_files>
{{project_path}}
</uploaded_files>
I've uploaded a code repository in the directory {{project_path}} (not in
/tmp/inputs). Consider the following PR description:

<pr_description>
{{git_issue}}
</pr_description>

Can you help me implement the necessary changes to the repository so that the
requirements specified in the <pr_description> are met?
I've already taken care of all changes to any of the test files described in the
<pr_description>. This means you DON'T have to modify the testing logic or any of
the tests in any way!

You should use tools as much as possible. You should also implement your own tests
first before attempting the problem.
```

GIT PR PROMPT WITH A think TOOL (HELPFUL FOR MODELS WITHOUT INTERLEAVED THINKING)

A variant that explicitly invokes a separate think tool at specific points in the workflow (problem restatement, fix selection, edge-case enumeration).

swe_generic_think_v1

```
System:

You are a helpful assistant that can interact with a computer to solve tasks.

Your task is to make the minimal changes to non-tests files in the {{project_path}}
directory to ensure the <pr_description> is satisfied.

Follow these steps to resolve the issue:
1. Before exploring anything, use the ‘think‘ tool to write up:
   - the task restated in your own words
   - 3-5 hypotheses for the root cause, ranked by likelihood
2. Explore the repo to familiarize yourself with its structure.
3. Create a script to reproduce the error and execute it with ‘python <filename.py>‘
using the BashTool, to confirm the error.
4. Use the ‘think‘ tool to list 2-3 candidate fixes in 1-2 lines each, then pick the
simplest one.
5. Edit the sourcecode of the repo to resolve the issue.
6. Rerun your reproduce script and confirm that the error is fixed.
7. Use the ‘think‘ tool to enumerate 3-5 edge cases for the changed code, then
exercise each via the reproduction script or shell.
8. Find and run the repository’s own existing tests for the files and functions you
modified (e.g., ‘pytest path/to/test_file.py‘, the project’s tests, etc.).
    - If any test fails, diagnose the failure, revise your fix, and rerun until they
all pass.
    - Do not finish until the relevant repo tests pass.

Your thinking should be thorough and so it’s fine if it’s very long.

User:

<uploaded_files>
{{project_path}}
</uploaded_files>
I’ve uploaded a python code repository in the directory {{project_path}} (not in
/tmp/inputs). Consider the following PR description:

<pr_description>
{{git_issue}}
</pr_description>

Can you help me implement the necessary changes to the repository so that the
requirements specified in the <pr_description> are met?
I’ve already taken care of all changes to any of the test files described in the
<pr_description>. This means you DON’T have to modify the testing logic or any of
the tests in any way!
```

Use this variant for including reasoning and tool-calling nudge.

swe_generic_think_v2

```
System:

You are a helpful assistant that can interact with a computer to solve tasks.

Your task is to make the minimal changes to non-tests files in the {{project_path}}
directory to ensure the <pr_description> is satisfied.

Follow these steps to resolve the issue:
1. Before exploring anything, use the ‘think‘ tool to write up:
   - the task restated in your own words
   - 3-5 hypotheses for the root cause, ranked by likelihood
2. Explore the repo to familiarize yourself with its structure.
3. Create a script to reproduce the error and execute it with ‘python <filename.py>‘
using the BashTool, to confirm the error.
4. Use the ‘think‘ tool to list 2-3 candidate fixes in 1-2 lines each, then pick the
simplest one.
5. Edit the sourcecode of the repo to resolve the issue.
6. Rerun your reproduce script and confirm that the error is fixed.
7. Use the ‘think‘ tool to enumerate 3-5 edge cases for the changed code, then
exercise each via the reproduction script or shell.
8. Find and run the repository’s own existing tests for the files and functions you
modified (e.g., ‘pytest path/to/test_file.py‘, the project’s tests, etc.).
    - If any test fails, diagnose the failure, revise your fix, and rerun until they
all pass.
    - Do not finish until the relevant repo tests pass.

Your thinking should be thorough and so it’s fine if it’s very long.

User:

<uploaded_files>
{{project_path}}
</uploaded_files>
I’ve uploaded a python code repository in the directory {{project_path}} (not in
/tmp/inputs). Consider the following PR description:

<pr_description>
{{git_issue}}
</pr_description>

Can you help me implement the necessary changes to the repository so that the
requirements specified in the <pr_description> are met?
I’ve already taken care of all changes to any of the test files described in the
<pr_description>. This means you DON’T have to modify the testing logic or any of
the tests in any way!

Always verify your changes extremely thoroughly. You can make as many tool calls as
you like - the user is very patient and prioritizes correctness above all else. Make
sure you are 100% certain of the correctness of your solution before ending.
```

H.2 GIT PR PROMPT WITH GPT-SPECIFIC INSTRUCTIONS

A modification to `swe_generic_v2` with additions of GPT models specifics such as `apply_patch` usage instructions. The prompt is tagged as `swe_generic_v2_gpt` in SSA.

GPT-flavored variant of `swe_generic_v2` that documents the `apply_patch` command (a GPT-preferred patch format) inline in the system prompt and switches `<pr_description>` to `<task_description>` in the user template.

swe_generic_v2_gpt

```
System:

You are a helpful assistant that can interact with a computer to solve tasks.

Your task is to make the minimal changes to non-tests files in the {{project_path}}
directory to ensure the <pr_description> is satisfied.

# High-Level Problem Solving Strategy
1. As a first step, it might be a good idea to explore the repo to familiarize
yourself with its structure.
2. Create a script to reproduce the error and execute it with ‘python <filename.py>‘
using the BashTool, to confirm the error
3. Edit the sourcecode of the repo to resolve the issue
4. Rerun your reproduce script and confirm that the error is fixed!
5. Think about edgecases and make sure your fix handles them as well

Your thinking should be thorough and so it’s fine if it’s very long.

In this environment, you can run ‘<apply_patch_command>‘ to execute a diff/patch
against a file, where <apply_patch_command> is a specially formatted apply patch
command representing the diff you wish to execute.
A valid <apply_patch_command> looks like:

apply_patch << ’PATCH’
*** Begin Patch
[YOUR_PATCH]
*** End Patch
PATCH

Where [YOUR_PATCH] is the actual content of your patch.

## Making Code Changes
- Before editing, always read the relevant file contents or section to ensure
complete context.
- If a patch is not applied correctly, attempt to reapply it.
- Make small, testable, incremental changes that logically follow from your
investigation and plan.

Always verify your changes extremely thoroughly. You can make as many tool calls as
you like - the user is very patient and prioritizes correctness above all else. Make
sure you are 100% certain of the correctness of your solution before ending.
IMPORTANT: not all tests are visible to you in the repository, so even on problems
you think are relatively straightforward, you must double and triple check your
solutions to ensure they pass any edge cases that are covered in the hidden tests,
not just the visible ones.

User:

<uploaded_files>
{{project_path}}
</uploaded_files>
I’ve uploaded a python code repository in the directory {{project_path}} (not in
/tmp/inputs). Consider the following PR description:

<task_description>
{{git_issue}}
</task_description>

Can you help me implement the necessary changes to the repository so that the
requirements specified in the <task_description> are met?
I’ve already taken care of all changes to any of the test files described in the
<pr_description>. This means you DON’T have to modify the testing logic or any of
the tests in any way
```

H.3 CLAUDE SPECIFIC PROMPT WITH REASONING/TOOL-CALL NUDGE

These prompts include the quantitative tool-call nudge ("ideally more than 100 times") discussed in Section 2. The prompt is tagged as `Anthropic_v2` in SSA.

Anthropic_v2

```
System:

You are a helpful assistant that can interact with a computer to solve tasks.

Your task is to make the minimal changes to non-tests files in the {{project_path}}
directory to ensure the <pr_description> is satisfied.

Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to explore the repo to familiarize
yourself with its structure.
2. Create a script to reproduce the error and execute it with ‘python <filename.py>‘
using the BashTool, to confirm the error
3. Edit the sourcecode of the repo to resolve the issue
4. Rerun your reproduce script and confirm that the error is fixed!
5. Think about edgecases and make sure your fix handles them as well

Your thinking should be thorough and so it’s fine if it’s very long.

User:

<uploaded_files>
{{project_path}}
</uploaded_files>
I’ve uploaded a python code repository in the directory {{project_path}} (not in
/tmp/inputs). Consider the following PR description:

<pr_description>
{{git_issue}}
</pr_description>

Can you help me implement the necessary changes to the repository so that the
requirements specified in the <pr_description> are met?
I’ve already taken care of all changes to any of the test files described in the
<pr_description>. This means you DON’T have to modify the testing logic or any of
the tests in any way!

You should use tools as much as possible, ideally more than 100 times. You should
also implement your own tests first before attempting the problem.
```

H.4 TERMINAL-BENCH-2 PROMPTS

Terminal-Bench-2 (TB2) prompts add explicit cleanup instructions and (in later variants) tool-usage guidelines tuned for the constrained environment described in Section 4.

GENERIC TB2 PROMPT

Starter prompt for TB2 tasks. Also used for Claude models.

tb2

```
System:

You are a helpful assistant that can interact with a computer to solve tasks.

Your task is to make minimal changes in the {{project_path}} directory to ensure the
<task_description> is satisfied.

Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to explore the repo to familiarize
yourself with its structure.
2. Write/edit code to finish the task.
3. Create test scripts for task validation and execute it using the BashTool, to
confirm task is finished.
4. Think about edgecases and make sure your changes handle them as well.
5. Clean up scratch artifacts ONLY - test scripts, experiment files, and temporary
files you created. Do NOT remove or stop anything the task's end-state depends on:
environment setup files, generated artifacts, or background services/daemons the
deliverable requires (e.g. an HTTP server, database, port listener). If the task
requires a service to be running at completion, launch it so it survives your shell
exiting.

Treat <task_description> as a sincere request for a real solution - not a problem to
be gamed. A solution that is suspiciously short for the stated problem, or that only
works because of a convenient resource, is almost certainly not what the user wanted.

Your thinking should be thorough and so it's fine if it's very long.


## Tool Usage Guidelines
- When using the shell tool, do not set a timeout unless the command is expected to
take longer than 1 minutes. Simple commands like ls, cat, grep, find, and file reads
need no timeout - the default is sufficient.
- When a timeout is needed, choose the smallest reasonable value. For example: 30s
for a test run, 60s for a build, 120s for a large install. Do not default to
large round numbers like 1000.
- If the shell tool accepts multiple inputs in one call (e.g. `batch_shell`), pack
independent commands together whenever the next command does not depend on the
previous one's output. Typical batches: layout discovery (`ls`, `cat README`, `find
...`), reading several files at once, or running unrelated tests/checks. Use a
single-input call only when you genuinely need to see the previous output before
choosing the next command.
- The container has internet access. If a tool or library you need isn't available,
do a minimal setup to install it rather than working around its absence.

User:

<uploaded_files>
{{project_path}}
</uploaded_files>
I've uploaded a code repository in the directory {{project_path}} (not in
/tmp/inputs). Consider the following task description:

<task_description>
{{git_issue}}
</task_description>

Can you help me implement the necessary changes to the repository so that the
requirements specified in the <task_description> are met?

Always verify your changes extremely thoroughly. You can make as many tool calls as
you like - correctness is prioritized above all else. Make sure you are 100\%
certain of the correctness of your solution before ending. Not all tests are visible
to you, so even on problems you think are straightforward, double and triple check
your solutions.
```

TB2 PROMPT FOR GEMINI MODELS

Gemini-flavored Terminal-Bench-2 variant; the user template omits the trailing instruction block from tb2.

tb2_gemini

```
System:

You are a helpful assistant that can interact with a computer to solve tasks.

Your task is to implement changes to ensure the <task_description> is satisfied.

Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to explore the repo to familiarize
yourself with its structure.
2. Write/edit code to finish the task.
3. Create test scripts for task validation and execute it using the BashTool, to
confirm task is finished.
4. Think about edgecases and make sure your changes handles them as well.
5. Clean up any test scripts, experiment files, or temporary files you created
during the process.

Your thinking should be thorough and so it's fine if it's very long.

User:

<uploaded_files>
{{project_path}}
</uploaded_files>
I've uploaded a code repository in the directory {{project_path}}. Consider the
following task description:

<task_description>
{{git_issue}}
</task_description>
```

TB2 PROMPT FOR GPT MODELS

GPT-flavored Terminal-Bench-2 variant; extends `tb2` with the `apply_patch` command documentation inline.

tb2_gpt

```
System:

You are a helpful assistant that can interact with a computer to solve tasks.

Your task is to make minimal changes in the {{project_path}} directory to ensure the
<task_description> is satisfied.

Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to explore the repo to familiarize
yourself with its structure.
2. Write/edit code to finish the task.
3. Create test scripts for task validation and execute it using the BashTool, to
confirm task is finished.
4. Think about edgecases and make sure your changes handle them as well.
5. Clean up scratch artifacts ONLY - test scripts, experiment files, and temporary
files you created. Do NOT remove or stop anything the task's end-state depends on:
environment setup files, generated artifacts, or background services/daemons the
deliverable requires (e.g. an HTTP server, database, port listener). If the task
requires a service to be running at completion, launch it so it survives your shell
exiting.

Treat <task_description> as a sincere request for a real solution - not a problem to
be gamed. A solution that is suspiciously short for the stated problem, or that only
works because of a convenient resource, is almost certainly not what the user wanted.

Your thinking should be thorough and so it's fine if it's very long.


## Tool Usage Guidelines
- When using the shell tool, do not set a timeout unless the command is expected to
take longer than 1 minutes. Simple commands like ls, cat, grep, find, and file
reads need no timeout - the default is sufficient.
- When a timeout is needed, choose the smallest reasonable value. For example: 30s
for a test run, 60s for a build, 120s for a large install. Do not default to
large round numbers like 1000.
- If the shell tool accepts multiple inputs in one call (e.g. `batch_shell`), pack
independent commands together whenever the next command does not depend on the
previous one's output. Typical batches: layout discovery (`ls`, `cat README`, `find
...`), reading several files at once, or running unrelated tests/checks. Use a
single-input call only when you genuinely need to see the previous output before
choosing the next command.
- The container has internet access. If a tool or library you need isn't available,
do a minimal setup to install it rather than working around its absence.
- In this environment, you can run `<apply_patch_command>` to execute a diff/patch
against a file, where <apply_patch_command> is a specially formatted apply patch
command representing the diff you wish to execute.
  A valid <apply_patch_command> looks like:

  apply_patch << 'PATCH'
  *** Begin Patch
  [YOUR_PATCH]
  *** End Patch
  PATCH

  Where [YOUR_PATCH] is the actual content of your patch.

User:
```

```
<uploaded_files>
{{project_path}}
</uploaded_files>
I've uploaded a code repository in the directory {{project_path}} (not in
/tmp/inputs). Consider the following task description:

<task_description>
{{git_issue}}
</task_description>

Can you help me implement the necessary changes to the repository so that the
requirements specified in the <task_description> are met?

Always verify your changes extremely thoroughly. You can make as many tool calls as
you like - correctness is prioritized above all else. Make sure you are 100% certain
of the correctness of your solution before ending. Not all tests are visible to you,
so even on problems you think are straightforward, double and triple check your
solutions.
```

TB2 PROMPT FOR QWEN3.6 MODELS

Minor modification to Terminal-Bench-2; extends tb2 with the re-reading and final verification to avoid early exit.

tb2_q36

```
System:

You are a helpful assistant that can interact with a computer to solve tasks.

Your task is to make minimal changes in the {{project_path}} directory to ensure the
<task_description> is satisfied.

Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to explore the repo to familiarize
yourself with its structure.
2. Write/edit code to finish the task.
3. Create test scripts for task validation and execute it using the BashTool, to
confirm task is finished.
4. Think about edgecases and make sure your changes handle them as well.
5. Clean up scratch artifacts ONLY -- test scripts, experiment files, and temporary
files you created. Do NOT remove or stop anything the task's end-state depends on:
generated output files, environment setup, or background services/daemons the
deliverable requires (e.g. an HTTP server, database, port listener). If the task
requires a service to be running at completion, launch it so it survives your shell
exiting.

Before you conclude the task is finished, do a final verification pass:
- Re-read the <task_description> and list EVERY explicit, checkable requirement it
states: exact output file paths and formats, numeric thresholds and limits (sizes,
counts, ranges, tolerances), required substrings/fields, and any "must/exactly/at
least/no more than" conditions.
- For each one, run a concrete command that checks the ACTUAL produced artifact
satisfies it, and read the output. Do not assume, verify against what the task
literally asked for, using the exact path/invocation it specifies.
- If any check fails, fix the real cause and re-verify. Only finish once every
checkable requirement has demonstrably passed.

Solve the task for real. Do not fabricate, hard-code, or guess an output to make a
check pass; if a value must be extracted/computed, extract/compute it properly.
Never modify the task's own test or grading files.

Your thinking should be thorough and so it's fine if it's very long.

User:

<uploaded_files>
{{project_path}}
</uploaded_files>
I've uploaded a code repository in the directory {{project_path}} (not in
/tmp/inputs). Consider the following task description:

<task_description>
{{git_issue}}
</task_description>

Can you help me implement the necessary changes to the repository so that the
requirements specified in the <task_description> are met?

You should use tools as much as possible, ideally more than 100 times. You should
also implement your own tests first before attempting the problem.
```

H.5 DEEP-SWE PROMPTS

Deep-SWE prompts extend `swe_generic_v2` with a regression baseline: the agent runs the repository's existing test suite before editing, and must finish with no more failing tests than that baseline. This targets Deep-SWE grading, where a submission that implements the task but breaks a previously passing test scores zero. The user template drops the note that test files have already been updated and instead asks for extremely thorough verification.

GENERIC DEEP-SWE PROMPT

Used for all non-GPT models on Deep-SWE (Claude, Fable, Gemini, Grok, Kimi, Muse Spark and Qwen).

swe_generic_v2_deepswe

```
System:

You are a helpful assistant that can interact with a computer to solve tasks.

Your task is to make the minimal changes to non-tests files in the {{project_path}}
directory to ensure the <pr_description> is satisfied.

Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to explore the repo to familiarize
yourself with its structure.
2. Run the repository's existing test suite BEFORE making any change and record
exactly which tests fail. This is your baseline: any test that passes now must still
pass when you are done.
3. Create a script to reproduce the error and execute it using the BashTool, to
confirm the error
4. Edit the sourcecode of the repo to resolve the issue
5. Rerun your reproduce script and confirm that the error is fixed!
6. Think about edgecases and make sure your fix handles them as well
7. Before you finish, rerun the full existing test suite and diff the failures
against your baseline from step 2. The set of failing tests must be no larger than
the baseline.

Your thinking should be thorough and so it's fine if it's very long.

User:

<uploaded_files>
{{project_path}}
</uploaded_files>
I've uploaded a code repository in the directory {{project_path}} (not in
/tmp/inputs). Consider the following PR description:

<pr_description>
{{git_issue}}
</pr_description>

Can you help me implement the necessary changes to the repository so that the
requirements specified in the <pr_description> are met?
Always verify your changes extremely thoroughly. You can make as many tool calls as
you like - the user is very patient and prioritizes correctness above all else. Make
sure you are 100% certain of the correctness of your solution before ending.
```

DEEP-SWE PROMPT WITH GPT-SPECIFIC INSTRUCTIONS

Used for the GPT-5.5, GPT-5.6 and GPT-6 models. It documents `apply_patch` inline.

swe_generic_v2_deepswe_gpt

```
System:

You are a helpful assistant that can interact with a computer to solve tasks.

Your task is to make the minimal changes to non-tests files in the {{project_path}}
directory to ensure the <pr_description> is satisfied.

# High-Level Problem Solving Strategy
1. As a first step, it might be a good idea to explore the repo to familiarize
yourself with its structure.
2. Run the repository's existing test suite BEFORE making any change and record
exactly which tests fail. This is your baseline: any test that passes now must still
pass when you are done.
3. Create a script to reproduce the error and execute it using the shell tool, to
confirm the error
4. Edit the sourcecode of the repo to resolve the issue
5. Rerun your reproduce script and confirm that the error is fixed!
6. Think about edgecases and make sure your fix handles them as well
7. Before you finish, rerun the full existing test suite and diff the failures
against your baseline from step 2. The set of failing tests must be no larger than
the baseline.

Your thinking should be thorough and so it's fine if it's very long.

In this environment, you can run `<apply_patch_command>` to execute a diff/patch
against a file, where <apply_patch_command> is a specially formatted apply patch
command representing the diff you wish to execute.
A valid <apply_patch_command> looks like:

apply_patch << 'PATCH'
*** Begin Patch
[YOUR_PATCH]
*** End Patch
PATCH

Where [YOUR_PATCH] is the actual content of your patch.

## Making Code Changes
- Before editing, always read the relevant file contents or section to ensure
complete context.
- If a patch is not applied correctly, attempt to reapply it.
- Make small, testable, incremental changes that logically follow from your
investigation and plan.

User:

<uploaded_files>
{{project_path}}
</uploaded_files>
I've uploaded a code repository in the directory {{project_path}} (not in
/tmp/inputs). Consider the following PR description:

<pr_description>
{{git_issue}}
</pr_description>

Can you help me implement the necessary changes to the repository so that the
requirements specified in the <pr_description> are met?
Always verify your changes extremely thoroughly. You can make as many tool calls as
you like - the user is very patient and prioritizes correctness above all else. Make
sure you are 100% certain of the correctness of your solution before ending.
```

```
IMPORTANT: not all tests are visible to you in the repository, so even on problems
you think are relatively straightforward, you must double and triple check your
solutions to ensure they pass any edge cases that are covered in the hidden tests,
not just the visible ones.
```

I TOOL DEFINITIONS

The harness ships a total of fifteen tool implementations across all model-families. Typically, a model only uses two tools — a `bash` tool and an `editing` tool.

Each tool exposes a `TOOL_SPEC` dict (name, description, `inputSchema`). Configurations bind a small subset of these tools to the agent (Section G). For example, Anthropic SWE-Bench configs bind `bash` and `str_replace_editor`, while the Terminal-Bench-2 GPT configs bind only `openai.batch_shell`.

Table 23: Tool inventory shipped in `ssa.tools`.

Tool	Summary	Used by
`bash`	issues a single bash command	Anthropic / Qwen / xAI / Gemini
`bash_timed`	issues a single bash command with timeout	Anthropic / Qwen / xAI for Terminal-Bench-2
`batch_bash`	can issue multiple bash commands	Gemini for Terminal-Bench-2
`str_replace_editor`	Monolith tool with `view`, `create`, `str_replace` and `undo_edit` modes	preferred by Anthropic / Qwen
`submit`	proxy to help with agent termination	Gemini
`think`	dedicated think tool for older models without interleaved thinking	Qwen3-Coder-30B / Qwen3-Coder-480B
`shell`	Execute a single command specified as string	GPT (`input`-style schema)
`batch_shell`	Execute multiple commands (specified as strings) sequentially	GPT models in Terminal-Bench-2
`apply_patch`	OpenAI-specific file editor using apply-patch mechanism	preferred by GPT models
`v1.shell`	Same as shell with input commands specified in array format	`cmd`-array (argv) shell variant for `gpt-oss` / Codex
`file_read`	reads file, view line-ranges for provided path	preferred by xAI Grok
`file_edit`	edit a file via string-replace mechanism	preferred by xAI Grok
`file_write`	write to a file with provided strings content	preferred by xAI Grok
`search`	rigprep-based string search	useful for `gpt-oss` models

I.1 GENERIC SHELL AND EDITOR TOOLS

bash

The default shell tool. Two required inputs (`description`, `command`); the timeout is read from the per-tool YAML config rather than from the model.

TOOL: bash

DESCRIPTION

```
Run commands in a bash shell
* When invoking this tool, the contents of the "command" parameter does NOT need
to be XML-escaped.
* You don't have access to the internet via this tool.
* You do have access to a mirror of common linux and python packages via apt and
pip.
* State is persistent across command calls and discussions with the user.
* Please avoid commands that may produce a very large amount of output.
* Please run long lived commands in the background, e.g. 'sleep 10 &' or start a
server in the background.
Note: Don't pipe long-running commands through tail/grep; redirect to a file (>
/tmp/out 2>&1 &) and inspect afterward
```

PARAMETERS

1. **description**: Why I am running this bash command.
2. **command**: The bash command to run.

bash_timed

Adds a model-supplied `timeout` (seconds) to the bash schema. Used in Terminal-Bench-2 configs where commands span the full range from instant `ls` to multi-minute test suites.

TOOL: bash_timed

DESCRIPTION

```
Run commands in a bash shell
* When invoking this tool, the contents of the "command" parameter does NOT need
to be XML-escaped.
* You don't have access to the internet via this tool.
* You do have access to a mirror of common linux and python packages via apt and
pip.
* State is persistent across command calls and discussions with the user.
* To inspect a particular line range of a file, e.g. lines 10-25, try 'sed -n
10,25p /path/to/the/file'.
* Please avoid commands that may produce a very large amount of output.
* Please run long lived commands in the background, e.g. 'sleep 10 &' or start a
server in the background.
```

PARAMETERS

1. **description**: Why I am running this bash command.
2. **command**: The bash command to run.
3. **timeout**: Seconds. Guidance: ls/cat/echo/grep → 5; find/git diff/git log → 15; pip install/git clone → 60; test suites, builds, large downloads → 200+; default 120 if unsure.

batch_bash

Multi-command variant. Accepts an array of {command, description, timeout?, ignore_errors?} entries that share working directory and environment variables.

TOOL: batch_bash

DESCRIPTION

```
* Run multiple bash commands in a single tool call.
* Use this tool to execute several independent commands at once instead of
calling bash multiple times.
* Commands are executed sequentially in the order provided.
* Each command runs in the same shell environment and shares state (working
directory, environment variables).
 - If you need strict sequential execution where a failure stops remaining
commands, set ignore_errors=false.
* When invoking this tool, the contents of each command does NOT need to be
XML-escaped.
* You don't have access to the internet via this tool.
* You do have access to a mirror of common linux and python packages via apt and
pip.
* State is persistent across command calls and discussions with the user.
* Please avoid commands that may produce a very large amount of output.
* Please run long lived commands in the background, e.g. 'sleep 10 &' or start a
server in the background.
```

PARAMETERS

1. **commands**: List of bash command objects, each containing:
   - **command**: The bash command to run.
   - **description**: Why this command is being run.
   - **timeout**: Seconds; default 120.
   - **ignore_errors**: Set false to stop on failure; default true.

`str_replace_editor`

A four-command (`view`, `create`, `str_replace`, `undo_edit`) file tool with whitespace-tolerant matching. Behind the scenes the harness applies the fuzzy-match logic described in Section 2 (multi-match disambiguation, partial-line refusal, diff-style success feedback).

TOOL: `str_replace_editor`

DESCRIPTION

```
Custom editing tool for viewing, creating, and editing files.
* State is persistent across command calls and discussions with the user
Supported features:
* If ‘path‘ is a file, ‘view‘ displays the result of applying ‘cat -n‘.
  If ‘path‘ is a directory, ‘view‘ lists non-hidden files and directories up to 2
levels deep
* The ‘create‘ command cannot be used if the specified ‘path‘ already exists as a
file
* If a ‘command‘ generates a long output, it will be truncated and marked with
‘<response clipped>‘
* The ‘undo_edit‘ command will revert the last edit made to the file at ‘path‘

Notes for using the ‘str_replace‘ command:
* The ‘old_str‘ parameter should match EXACTLY one or more consecutive lines from
the original file. Be mindful of whitespaces!
* If the ‘old_str‘ parameter is not unique in the file, the replacement will not
be performed. Make sure to include enough context in ‘old_str‘ to make it unique
* The ‘new_str‘ parameter should contain the edited lines that should replace the
‘old_str‘
```

PARAMETERS

1. **command**: view | create | str_replace | undo_edit.
2. **description**: Why I’m making this edit.
3. **path**: Absolute path to file or directory.
4. **file_text**: Required for create: content of new file.
5. **old_str**: Required for str_replace: exact text to replace.
6. **new_str**: Required for str_replace: replacement text.
7. **view_range**: Optional for view: [start, end] line range.

`submit`

Explicit termination signal. Setting `request_state["stop_event_loop"] = True` is what tells the harness loop the agent is done; the `paths` argument is what gets evaluated against the benchmark's hidden tests.

TOOL: submit

DESCRIPTION

```
Submit the final result and conclude the task.
Call this tool ONLY when all assigned tasks have been completed successfully.
This signals that the agent has finished its work and is ready to return the
final output.
Do not call this tool if there are remaining steps, unresolved errors, or
incomplete objectives.
```

PARAMETERS

1. **summary**: Brief summary of what was accomplished.
2. **status**: success | partial_success | failure.
3. **paths**: Files created/modified to submit. Include only files strictly necessary; broader changes may break hidden tests.

`think`

A no-op tool whose only effect is to log the agent's thought. Used as a deliberation step — the model is asked to enumerate hypotheses or candidate fixes before committing to a multi-step action.

TOOL: think

DESCRIPTION

```
Structure your reasoning before committing to a multi-step action. This tool does
not read files, run code, or change the repository - it forces a deliberation
step so you act on a considered plan rather than the first idea.

Call it in these situations:
1. After localizing a bug, before writing the patch - state the root cause in one
sentence, then list 2-4 candidate fixes. For each, note whether it addresses the
root cause or just the observed symptom, and pick the minimal change that fixes
the cause.
2. After a failed test or reproduction - list 2-4 hypotheses for the failure
ranked by likelihood, and identify the cheapest next action that would
discriminate between them.
3. Before a refactor or multi-file change - outline the approach, the files
touched, and one concrete risk (e.g., a caller you might break).

Skip for trivial actions: one-line fixes, obvious next commands, simple file
lookups, or steps where the next action is already determined. Do not use it to
narrate what you just did.

Format: 3-5 concise bullets. No prose paragraphs. No restating the problem.
```

PARAMETERS

1. **thought**: Your structured reasoning (3-5 bullets, no prose paragraphs).

I.2 OpenAI-specific tools

`shell`

Input field is named `input` (not `command`) and exposes a model-controlled `timeout`. Used for the GPT model-family.

TOOL: shell

DESCRIPTION

```
Run commands in a bash shell
* When invoking this tool, the contents of the "command" parameter does NOT need
to be XML-escaped.
* You don't have access to the internet via this tool.
* State is persistent across command calls and discussions with the user.
* To inspect a particular line range of a file, e.g. lines 10-25, try 'sed -n
10,25p /path/to/the/file'.
* Please avoid commands that may produce a very large amount of output.
* Please run long lived commands in the background, e.g. 'sleep 10 &' or start a
server in the background.
```

PARAMETERS

1. **description**: Why I am running this bash command.
2. **input**: The shell command to run.
3. **timeout**: Seconds; set conservatively to exceed expected runtime: 5–10s instant; 30–60s installs; 120–180s test suites/builds; 300–600s extreme.

`batch_shell`

Batched variant of `shell` for Terminal-Bench-2 configs. The description explicitly tells the model when packing inputs is preferred (layout discovery, parallel test runs, multi-file reads) and when a single input is correct (the next command depends on the previous output).

TOOL: batch_shell

DESCRIPTION

```
Run one or more bash commands in a single tool call.

When to pack multiple inputs in one call (PREFERRED whenever possible):
* Independent exploration steps whose next command does not depend on the
previous output, e.g. layout discovery: `ls -la`, `cat README.md`, `find . -name
pyproject.toml`, `git log --oneline -20`.
* Running several unrelated test files or check commands together, e.g. `pytest
tests/test_a.py`, `pytest tests/test_b.py`, `ruff check src/`.
* Reading multiple files or inspecting multiple symbols at once, e.g. `sed -n
'1,80p' a.py`, `sed -n '1,80p' b.py`, `grep -n foo src/`.

When to use a SINGLE input:
* The next command genuinely depends on the previous output (e.g. you need to see
a file before deciding what to edit).
* You are running a single long command or a script.
```

PARAMETERS

1. **inputs**: List of shell command objects, each containing:
   - **input**: The shell command to run.
   - **description**: Why this command is being run.
   - **timeout**: Seconds; default 10, max 600.
   - **ignore_errors**: Set false to stop on failure; default true.

apply_patch

A standalone diff-applying tool that wraps OpenAI's `*** Begin Patch / *** End Patch` format. Available as a tool call for configs that prefer it over heredoc patching inside `shell`.

TOOL: `apply_patch`

DESCRIPTION

```
Apply a patch to one or more files using the apply_patch format. The ‘patch‘
value MUST start with ‘*** Begin Patch‘ and end with ‘*** End Patch‘. Example
‘patch‘ value:
*** Begin Patch
*** Update File: path/to/file.py
@@
- old line
+ new line
*** End Patch
```

PARAMETERS

1. **patch**: apply_patch payload bounded by *** Begin Patch / *** End Patch.
2. **description**: Why I'm making this edit.

v1.shell

A second OpenAI shell variant that takes a `cmd` argv array rather than a single `input` string, matching the native shape that `gpt-oss-120b` and Codex models emit. The harness collapses `["bash", "-lc", "<body>"]` back into the body string before execution.

TOOL: `v1.shell`

DESCRIPTION

```
Run commands in a bash shell.
* The "cmd" field is an argv array. The conventional invocation is ["bash",
"-lc", "<your command>"].
* When invoking this tool, the contents of the command do NOT need to be
XML-escaped.
* You don't have access to the internet via this tool.
* State is persistent across command calls.
* To inspect a particular line range of a file, e.g. lines 10-25, try 'sed -n
10,25p /path/to/the/file'.
* Please avoid commands that may produce a very large amount of output.
* Please run long lived commands in the background, e.g. 'sleep 10 &' or start a
server in the background.
```

PARAMETERS

1. **cmd**: Argv array; typically ["bash", "-lc", "<cmd>"].

I.3 xAI-SPECIFIC TOOLS

`file_read`

A read-only file viewer split out of `str_replace_editor`. Grok-4.20 emits a variety of line-range parameter names (`lineStart`, `startLine`, `from_line`, etc.); the implementation contains a `_infer_line_range` fallback that scans `tool_input` for any key matching a line-endpoint pattern — an example of harness-side tolerance for model preferrence.

TOOL: `file_read`

DESCRIPTION

```
Tool for viewing files and directories.
* If ‘path‘ is a file, ‘view‘ displays the result of applying ‘cat -n‘.
  If ‘path‘ is a directory, ‘view‘ lists non-hidden files and directories up to 2
levels deep.
* Use ‘view_range‘ to display specific line ranges of a file, e.g. [11, 20] shows
lines 11-20.
* If output is too long, it will be truncated and marked with ‘<response
clipped>‘.
```

PARAMETERS

1. **path**: Absolute path to file or directory.
2. **view_range**: [start, end] line range; use [start, -1] for “from start to EOF.”

`file_edit`

The xAI counterpart of `str_replace_editor`’s editing path. Two commands (`str_replace`, `undo_edit`); creation is split out into `xai.file_write` so the model has to commit to one mode of modification per call.

TOOL: `file_edit`

DESCRIPTION

```
Tool for editing existing files.
* The ‘str_replace‘ command replaces an exact string in a file with a new string.
* The ‘undo_edit‘ command reverts the last edit made to the file.

Notes for using the ‘str_replace‘ command:
* The ‘old_str‘ parameter should match EXACTLY one or more consecutive lines from
the original file. Be mindful of whitespaces!
* If the ‘old_str‘ parameter is not unique in the file, the replacement will not
be performed. Make sure to include enough context in ‘old_str‘ to make it unique.
* The ‘new_str‘ parameter should contain the edited lines that should replace the
‘old_str‘.
* To create new files, use the ‘file_write‘ tool instead.
```

PARAMETERS

1. **command**: str_replace | undo_edit.
2. **description**: Why I’m making this edit.
3. **path**: Absolute path to file.
4. **old_str**: Required for str_replace: exact text to replace.
5. **new_str**: Required for str_replace: replacement text.

`file_write`

File-creation tool. Fails if the target already exists.

TOOL: file_write

DESCRIPTION

```
Tool for creating new files.
* Creates a new file at the given path with the provided content.
* Fails if the file already exists - use `file_edit` to modify existing files.
* The parent directory must already exist.
```

PARAMETERS

1. **path**: Absolute path for the new file.
2. **file_text**: Full content of the file to be created.
3. **description**: Why I'm creating this file.

`search`

Ripgrep-backed search tool. The description includes a hard rule ("ALWAYS use this tool for search tasks. NEVER invoke grep or rg as a Bash command") because Grok otherwise tends to issue `grep` via the bash tool and loses the structured output.

TOOL: search

DESCRIPTION

```
A powerful search tool built on ripgrep.
* ALWAYS use this tool for search tasks. NEVER invoke grep or rg as a Bash
command.
* `query` supports full regex syntax (ripgrep flavor), e.g. "log.*Error",
"function\s+\w+".
* `path` restricts the search to a specific file or directory; defaults to the
working directory.
* Returns matching lines with file path and line number, capped at `max_results`
entries.
```

PARAMETERS

1. **query**: Regex pattern (ripgrep syntax).
2. **path**: File or directory to search in; defaults to cwd.
3. **max_results**: Max matching lines to return.

J vLLM Serving Configurations

The open-weights configurations in Section G (Qwen-3.5/3.6/3.8 variants (Yang et al., 2025), gpt-oss-120b/20b (OpenAI, 2025a)) are hosted on a private vLLM (Kwon et al., 2023) endpoint stood up on an internal SLURM cluster with Amazon EC2 `p4.24xlarge` instances, with a LiteLLM (BerriAI, 2023) proxy. The `vllm serve` invocations used to bring those endpoints up are listed below.

J.1 PER-MODEL SERVE COMMANDS

qwen3.5-27b

```
vllm serve Qwen/Qwen3.5-27B \
  --host 0.0.0.0 --port 8000 \
  --tensor-parallel-size 8 \
  --distributed-executor-backend ray \
  --max-model-len 200000 \
  --gpu-memory-utilization 0.92 \
  --enable-prefix-caching \
  --enable-prompt-tokens-details \
  --reasoning-parser qwen3 \
  --enable-auto-tool-choice \
  --tool-call-parser qwen3_xml \
  --served-model-name qwen3.5-27b \
  --language-model-only
```

qwen3.6-35b-a3b (SWE tasks)

```
vllm serve Qwen/Qwen3.6-35B-A3B \
  --host 0.0.0.0 --port 8100 \
  --tensor-parallel-size 8 \
  --distributed-executor-backend ray \
  --max-model-len 200000 \
  --gpu-memory-utilization 0.92 \
  --enable-prefix-caching \
  --enable-prompt-tokens-details \
  --reasoning-parser qwen3 \
  --enable-auto-tool-choice \
  --tool-call-parser qwen3_xml \
  --served-model-name qwen3.6-35b-a3b \
  --language-model-only
```

qwen3.6-35b-a3b (Terminal-Bench-2)

```
vllm serve Qwen/Qwen3.6-35B-A3B \
  --host 0.0.0.0 --port 8100 \
  --tensor-parallel-size 8 \
  --distributed-executor-backend ray \
  --max-model-len 262144 \
  --gpu-memory-utilization 0.92 \
  --enable-prefix-caching \
  --enable-prompt-tokens-details \
  --reasoning-parser qwen3 \
  --enable-auto-tool-choice \
  --tool-call-parser qwen3_xml \
  --served-model-name qwen3.6-35b-a3b \
  --language-model-only
```

qwen3.6-27b (SWE tasks)

```
vllm serve Qwen/Qwen3.6-27B \
  --host 0.0.0.0 --port 8200 \
  --tensor-parallel-size 8 \
  --distributed-executor-backend ray \
  --max-model-len 200000 \
  --gpu-memory-utilization 0.92 \
  --enable-prefix-caching \
  --enable-prompt-tokens-details \
  --reasoning-parser qwen3 \
  --enable-auto-tool-choice \
  --tool-call-parser qwen3_xml \
  --served-model-name qwen3.6-27b \
  --language-model-only
```

qwen3.6-27b (Terminal-Bench-2)

```
vllm serve Qwen/Qwen3.6-27B \
  --host 0.0.0.0 --port 8200 \
  --tensor-parallel-size 8 \
  --distributed-executor-backend ray \
  --max-model-len 262144 \
  --gpu-memory-utilization 0.92 \
  --enable-prefix-caching \
  --enable-prompt-tokens-details \
  --reasoning-parser qwen3 \
  --enable-auto-tool-choice \
  --tool-call-parser qwen3_xml \
  --served-model-name qwen3.6-27b \
  --language-model-only
```

qwen3.8-27b (Deep-SWE)

```
vllm serve Qwen/Qwen3.8-27B \
  --host 0.0.0.0 --port 8000 \
  --tensor-parallel-size 8 \
  --max-model-len 262144 \
  --gpu-memory-utilization 0.90 \
  --enable-prefix-caching \
  --enable-prompt-tokens-details \
  --reasoning-parser qwen3 \
  --enable-auto-tool-choice \
  --tool-call-parser qwen3_xml \
  --served-model-name qwen3.8-27b \
  --language-model-only
```

Served with vLLM 0.28.0 using its default multiprocess executor (no Ray), one TP=8 replica per node.

gpt-oss-120b

```
vllm serve openai/gpt-oss-120b \
  --host 0.0.0.0 --port 8300 \
  --tensor-parallel-size 8 \
  --max-model-len 131072 \
  --gpu-memory-utilization 0.90 \
  --enable-prefix-caching \
  --enable-prompt-tokens-details \
  --reasoning-parser openai_gptoss \
  --enable-auto-tool-choice \
  --tool-call-parser openai \
  --served-model-name gpt-oss-120b
```

MXFP4 datatype is auto-detected from the model config.

gpt-oss-20b

```
vllm serve openai/gpt-oss-20b \
  --host 0.0.0.0 --port 8400 \
  --tensor-parallel-size 2 \
  --max-model-len 131072 \
  --gpu-memory-utilization 0.90 \
  --enable-prefix-caching \
  --enable-prompt-tokens-details \
  --reasoning-parser openai_gptoss \
  --enable-auto-tool-choice \
  --tool-call-parser openai \
  --served-model-name gpt-oss-20b
```

J.2 A NOTE ON VLLM FLAGS

- **Compile mode is on for every model.** `ENFORCE_EAGER=0` in every sbatch means no `-enforce-eager` and no dynamo disable: $\sim$2–3$\times$ decode speedup over eager.
- **No `-quantization` flag anywhere.** BF16 models load BF16 weights; gpt-oss auto-detects `mxfp4` from its checkpoint's `config.json`.

K HARMONY-FORMAT PARSING FAILURES IN THE HARNESS

This section describes the initial gap we observed for the open-weights gpt-oss evaluations (`gpt-oss-120b` and `gpt-oss-20b`) in the officially reported pass@1. For `gpt-oss-120b`, on SWE-Bench-Verified, we observed 42% pass@1 while it is reported at 62% in the release card. The root-cause was a failure mode observed in the open-weights `gpt-oss` evaluations when sampling was done at high temperatures, such as 1.0. The affected outputs had a mix of non-existent tool-calls and empty tool-use payloads. After fixing the issue, the pass@1 of SSA increased to 64.64% as reported in Appendix C.

This is an interface failure with disproportionate trajectory-level consequences. In an agent loop, one corrupted tool turn is fed back as an environmental observation. The model then reasons from a false account of its own action and spends subsequent turns repairing a call that was never malformed. The remainder of this appendix documents the symptom and the cache diagnostic that exposed the affected batch (Section K.1), the stateless re-tokenization boundary that makes the corruption persistent once it reaches the transcript (Section K.2), token-level evidence (Section K.3), and the retry-based mitigation (Section K.4).

K.1 THE SYMPTOM: EMPTY AND MALFORMED TOOL CALLS

We identified the issue after noticing an abnormal cache hit-rate of zero for `gpt-oss` experiments. In a healthy multi-turn agent trajectory the cache hit rate should rise towards 1.0 as each request resubmits a long prefix that the previous request has already processed and cached. Intermittently, we observed that the input cache was dropped in the middle of the trajectory (cache hit-rate of zero) followed by a cache-write. It indicated that prefix matching before cache-read failed and hence a mismatch existed between agent-sent input context vs what existed in the server. Since, agent context and server cached context differs only at the last assistant message, these events provided a first-level diagnostic of tampered model generated output sequence.

In the logs, we found that `gpt-oss` turns intermittently decode into a tool-call whose argument body is empty, or into a tool-call whose name contains literal Harmony control-token text. Both cases are problematic because they are presented to the agent as an ordinary tool outcomes. The model is then shown a validation error for an action it did not actually emit in that form, and its next turn is pulled away from the original task.

The failure location is the Harmony-to-OpenAI (chat-completion format) response boundary. `gpt-oss` encodes assistant output in the *Harmony* format (OpenAI, 2025b), which multiplexes internal reasoning, tool-use communication, and final answers onto one token stream. The `analysis` channel carries the model's reasoning, the `commentary` channel carries tool calls and tool-use planning, and the `final` channel carries the user-visible response. Reserved control tokens delimit channels and message bodies (Table 24). A serving stack such as vLLM (Kwon et al., 2023) must demultiplex this stream and re-emit a standard OpenAI-compatible response (for example, a chat-completion api). Reasoning stripped or separated, ordinary text for final answers, and a structured `{name, arguments}` object for each parsed tool call. The observed corruption enters at this demultiplexing stage, where an ill-formed Harmony boundary can still be returned to the client as a nominally valid tool call.

Table 24: Harmony control tokens that delimit channels and messages. The two assistant stop tokens, `<|return|>` and `<|call|>`, both halt generation but signal different terminal states; the distinction is used by the retry check in Section K.4.

Token	Role in the stream
`<|start|>`	Begin a new message; followed by the role (and optional `to=functions.<name>` recipient)
`<|channel|>`	Followed by the channel name (`analysis` / `commentary` / `final`)
`<|constrain|>`	Begin a constrained span (e.g. declaring the tool-argument body is `json`)
`<|message|>`	Begin the message body (free-form text or the tool-argument JSON)
`<|end|>`	End of a non-terminal message (more messages follow in the turn)
`<|return|>`	End of the assistant turn with a final answer (normal end-of-stream)
`<|call|>`	End of an assistant turn that emitted a tool call

K.2 THE SERVING BOUNDARY: STATELESS RE-TOKENIZATION

The main issue is the misplacement of Harmony control tokens in Table 24 in the parsed JSON output. Due to high sampling temperature of 1.0, the model has a higher chance of generating delimiter tokens (such as <|channel|>, <|call|>) at improper locations. This ill-generated token-sequence passes through Chat Completions API which converts it to structured JSON. See Figure 59, where one incorrect generated token confused the system to parse the string into a tool-name that didn't exist.

Once a corrupted turn is admitted into the transcript, the serving path has no mechanism for restoring the original token sequence. The Chat Completions API is stateless at the token-level. At every turn, the harness resubmits the conversation as a structured JSON and the server reconstructs the prompt rather than reusing the token-ids generated in the previous turn. In the `gpt-oss` path, vLLM rebuilds Harmony `Message` objects from fields such as `content`, `reasoning`, and each `tool_call`'s `name` and `arguments`. It then re-encodes those reconstructed messages for the next prefill. Each assistant turn is therefore decoded, parsed into JSON, returned to the client, and re-tokenized from JSON when the conversation continues.

This decode–re-encode operation is not an identity map, which is the token fidelity issue discussed by Gallouédec & Rasul (2026). Tokenization is not generally invertible as re-encoding decoded text need not recover the original token-ids. Ordinary text usually survives this round trip, which is why prompt caching remains effective, but the reconstructed assistant turn is not byte-identical to the model's original output. Prior `analysis` content is intentionally dropped during reconstruction, and tool-call bodies are re-rendered as ordinary `json` rather than as the original <|constrain|>json span. These deterministic differences affect only the final assistant segment of the prompt, while the longer shared prefix still re-tokenizes identically and can be served from cache.

Note, a token-in/token-out (TITO) interface would avoid this failure by feeding the original generated token-ids back into the model, rather than reconstructing them from text. The JSON-based Chat Completions path cannot do that. It must rebuild the prompt from structured fields, and therefore cannot distinguish a reserved Harmony control token from a literal string that happens to spell the same text. For example, once <|channel|> has leaked into decoded text, it re-encodes as ordinary text tokens, not as the single reserved control-token id. Consequently, a malformed Harmony turn that reaches the transcript remains malformed on subsequent requests.

K.3 TOKEN-LEVEL EVIDENCE

In healthy turns, Harmony control tokens separate the reasoning message, the tool-call header, and the argument body. In failing turns, those boundaries collapse: an <|end|> that should close the analysis message, or a fresh <|start|>...<|channel|>...<|constrain|> header that should open the tool call, is missing or fused with adjacent text. As a result, channel names, recipients, and message bodies are no longer separated in the form expected by the parser. The figures below show the relevant decoded `gpt-oss-120b` streams, together with the client-visible error and, the clean retry that follows.

When no clean argument span can be recovered, this boundary collapse yields the empty-argument tool call described in Section K.1. When a control token is instead incorporated into the recipient, the same family of failures surfaces as a *corrupt tool name*. One observed server response named the tool `shell<|channel|>commentary`, and SSA rejected it because no registered tool has that name.

A control token folded into the tool name, rejected at the harness

```
Intended tool_name=shell
Decoded tool_name=<shell<|channel|>commentary>
SSA replies -> 'invalid tool name'
```

A corrupt tool name changes the trajectory. In Figure 58, the model attempts to read a specific function, but a stray <|channel|> becomes part of the tool recipient. The harness rejects the resulting name with `invalid tool name pattern`. The model cannot observe the injected token, interprets the failure as its own usage error, retries unsuccessfully, and then abandons the targeted read in

favor of a search. Thus the corrupted parse does not merely consume a turn, it removes an intended observation from the trajectory. Figure 59 shows the corresponding raw streams.

TURN k — a stray token corrupts the tool name

INTENT (model's reasoning)

The model wants to read a specific function: *"Inspect* `sympy/core/add.py` *for its* `_eval_power`*."*

EXECUTION (harness closed-loop)

It issues `file_read` tool call, but a stray `<|channel|>` folds into the tool-name header, so the name parses as `file_read<|channel|>commentary`.
The harness answers `tool_name=<file_read<|channel|>commentary> | invalid tool name pattern`.

TURN $k{+}1$ — attributes the failure to its own usage

INTENT (model's reasoning)

Reading the rejection as its own mistake, model thinks: *"Actually the tool name is* `file_read`*. Use that."*

EXECUTION (harness closed-loop)

It re-issues the same `file_read`; and folds in the stray token, so the same `invalid tool name pattern` returns. The model never gets to see the problematic token in the tool error message, so the retry does not help.

TURN $k{+}2$ — the targeted read is abandoned

INTENT (model's reasoning)

Unable to open the file, the model drops the inspection and switches to a different `search` action.

EXECUTION (harness closed-loop)

`search(. . . query="_eval_power". . . )` runs, but the targeted view of `add.py` lines 340–380 is *dropped here*: the model proceeds on what it can infer for now, having lost the read it set out to make. ×

Figure 58: **A corrupted tool-name causes the model to abandon an intended read** (`gpt-oss-120b`, `sympy-13974` (SWE-Bench-Verified). The model sets out to read `add.py` around `_eval_power`. A stray `<|channel|>` is folded into the `file_read` recipient, producing `file_read<|channel|>commentary`. The harness rejects the call with `invalid tool name pattern`. Because the injected token is invisible to the model, it concludes that it used the tool incorrectly. After the retry fails, the model pivots to `search` (Turn $k{+}2$), so the intended inspection of those `add.py` lines is skipped at that point. Complete streams are shown in Figure 59.

Token-by-token streams

Turn k — the corrupted `file_read`: a stray `<|channel|>` is inserted after the recipient.

`<|channel|>` analysis `<|message|>` Inspect sympy/core/add.py for its _eval_power. `<|end|>` `<|start|>` assistant `<|channel|>` commentary to=functions.file_read `<|channel|>` commentary code `<|message|>` `{"path": "/testbed/sympy/core/add.py", "view_range": [340, 380]}` `<|call|>`

⇒ the recipient and the `<|channel|>` after it parse together as the tool name `file_read<|channel|>commentary`, so the harness returns `invalid tool name pattern`.

Turn $k{+}1$ — it attributes the failure to its own usage and retries, but the recipient now sits on the `analysis` channel (a tool call must be on `commentary`):

`<|channel|>` analysis `<|message|>` Actually the tool name is file_read. Use that. `<|end|>` `<|start|>` assistant `<|channel|>` analysis to=functions.file_read `<|constrain|>` json `<|message|>` `{"path": "/testbed/sympy/core/add.py", "view_range": [340, 380]}` `<|call|>`

⇒ on the wrong channel the call is not dispatched, so the `file_read` still does not land.

Turn $k{+}2$ — the targeted read is abandoned and the model pivots to a `search` (no stray token here):

`<|channel|>` analysis `<|message|>` Thus there is independent handling for Pow for base with power. `<|end|>` `<|start|>` assistant `<|channel|>` analysis to=functions.search code `<|message|>` `{"path": "/testbed/sympy", "query": "if hasattr(base, \"_eval_power\")", "max_results": 20}` `<|call|>`

⇒ `search` runs; the inspection of `add.py` lines 340–380 is dropped here — the corrupted parse redirected the trajectory.

Figure 59: **Token-level view of the corrupt-name example in Figure 58.** Each reserved Harmony control token is a grey chip. In Turn k the single defect is the `<|channel|>` the server inserts after `to=functions.file_read`, where `code<|message|>` should follow. The parser reads the recipient and that stray token together as the tool name `file_read<|channel|>commentary`, so the call is rejected. The model cannot see the token, so it attributes the failure to its own usage, its retry (Turn $k{+}1$) puts the recipient on the `analysis` channel and is silently not dispatched, so it pivots to a `search` (Turn $k{+}2$). The read of `add.py` it set out to make is dropped at this point.

K.4 The fix: surface a single retryable error

The mitigation is to make a corrupted sample retryable before it becomes part of the model-visible transcript. The failures arises from rare token sequences at temperature 1.0, and resampling the same prompt typically yields a well-formed turn. The OpenAI-compatible server is therefore hardened so that all observed corruption modes converge to a single explicit signal, `HarmonyParseError` over HTTP 503, which SSA treats as retryable with backoff.

Evaluation after the fix Re-serving the identical agent configuration through a corrected vLLM build drops these parse errors to *zero*, localizing the failure to Harmony handling in the model server rather than to the model weights or SSA's tool registry. The retry hook described above recovers all Harmony parsing errors increasing 'pass@1' of `gpt-oss-120b` from 42% to 64.64% (reported model card number is 62%).