SWE-CONTINUUM: MULTI-STAGE PROFILING OF BEHAVIORAL CONTINUITY IN LONG-HORIZON AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding benchmarks usually score the final repository state, omitting when functionality is gained, lost, or recovered. We introduce **SWE-Continuum**, a benchmark for multi-stage profiling of behavioral continuity, with 16 tasks and 111 requirements from 16 repositories in five languages. Tasks release requirements in a persistent workspace under fixed DAG-based schedules. A verifier checks all released requirements and baseline behavior after each stage. A construction pipeline aligns instructions, tests, and reference patches, with no-op, oracle, dry-run, and human checks. The Area Under the Progress–Expenditure Curve (AUPEC) summarizes net progress over a shared token, time, or cost budget. Across eight models, 17 of 123 valid trials contain 64 pass-to-fail transitions, yet net progress rises at 27 of the 29 checkpoints containing one. Models with nearly equal final net progress differ 4.6-fold in token expenditure; terminal net progress and token AUPEC reverse 14 of 28 model-pair rankings. These results distinguish terminal capability, behavioral continuity, and budgeted net progress.

1 INTRODUCTION

Large language model (LLM) agents increasingly tackle software engineering, system administration, and data analysis (Liu et al., 2024; Merrill et al., 2026; Hong et al., 2025). Beyond isolated requests, these domains involve work continuing in a persistent environment. Agents must satisfy accumulating requirements while preserving previously established functionality (Shen et al., 2026; Deng et al., 2026). We refer to this ability to extend a system without regressing previously passing behavior as *behavioral continuity*, a dimension of agent performance that complements terminal capability.

Model selection depends on correctness, latency, resource expenditure, and reliability under a user's constraints (Liang et al., 2022; Kapoor et al., 2024). Models with similar final capability can incur different costs or follow different development histories. Evaluation should therefore profile both behavior and resource use.

Existing coding evaluations focus on either terminal outcomes or intermediate behavior. Terminal scores capture the final repository state (Jimenez et al., 2024; Deng et al., 2025) but miss when functionality is acquired, whether it regresses, and how much resource is spent. Recent benchmarks expose more of the development trajectory through simulator feedback (Raghavendra et al., 2026), partial-completion scoring (Xu et al., 2026), cumulative tests in persistent development (Shen et al., 2026), and dependency-structured tasks with regression analysis (Deng et al., 2026; Li et al., 2026). However, controlled comparison remains difficult when agent actions change subsequent feedback or requirements, causing runs to receive different information.

SWE-CONTINUUM releases related requirements sequentially in a persistent workspace and verifies progress after each stage (Figure 1). Its *task scheduler* follows a fixed release schedule and cumulative verification scope, compiled from a requirement directed acyclic graph (DAG) under a deterministic policy. Agents receive the same requirement sequence and number of turns, while their workspaces and tool feedback evolve with their actions. Matched message interventions preserve the turn count when testing context sensitivity, motivated by distractor studies (Shi et al., 2023).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

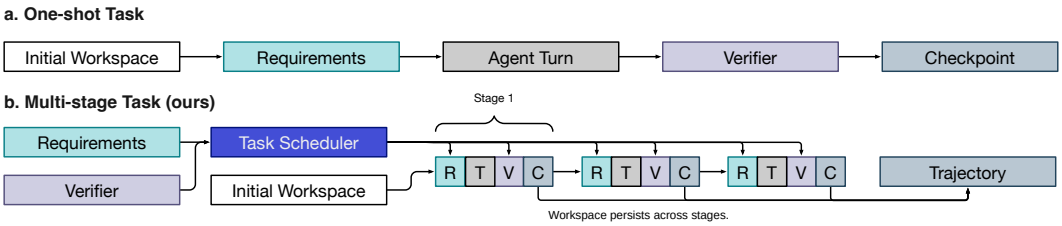


Figure 1: One-shot and multi-stage task protocols. (a) Requirements are given upfront and checked once. (b) The task scheduler fixes requirement release (R) and verification scope (V). Each stage runs one agent turn (T) and records a checkpoint (C). Verification covers all released requirements and baseline behavior. The workspace persists across stages; the checkpoints form one trajectory.

SWE-CONTINUUM aligns instructions, reference patches, and cumulative F2P and P2P tests through a task construction pipeline (§ 3.2), using the test partition of SWE-bench (Jimenez et al., 2024). Before evaluation, we run environment builds, no-op and staged oracle checks, and dry runs. Human review checks the requirement DAG and instruction–test–patch alignment; repeated attempts support task selection for the stable v1 release.

We use these checkpoint histories for *multi-stage profiling*: measuring **terminal capability, behavioral continuity, and budgeted net progress** (Section 3.3). Net progress S, F2P completion minus baseline breakage, measures capability at each checkpoint. Stable test IDs measure continuity: they show when each tested behavior is acquired, lost, and recovered, including losses that S hides when other tests pass at the same checkpoint. The Area Under the Progress–Expenditure Curve (AUPEC), averages S over a shared token, time, or cost budget: earlier progress raises it, while slow starts, late gains, and persistent net losses lower it. It summarizes a progress history in one number when inspecting full curves is impractical. A score–cost Pareto plot retains endpoint trade-offs; AUPEC additionally distinguishes histories with the same final score and expenditure, on the resource axis a user cares about.

Our contributions are:

1. **SWE-CONTINUUM**, 16 multi-stage coding tasks with 111 requirements from 16 repositories in five languages, verified after every stage in a persistent workspace.
2. **A task construction pipeline** that aligns each requirement with an instruction, owned F2P tests, and a reference patch, compiles a dependency DAG into a release schedule, and checks tasks through no-op, staged oracle, dry-run, and human review procedures.
3. **AUPEC and continuity metrics** that measure net progress within a token, time, or cost budget and record the intermediate states a terminal score omits.
4. **A multi-axis evaluation and analysis of eight models.** Staged verification finds 64 pass-to-fail transitions in 17 of 123 valid trials, and S still rises at 27 of the 29 checkpoints that contain them. Terminal S and token AUPEC reverse 14 of 28 model-pair rankings, and the preferred model changes with the resource axis.

We will open-source the interactive visual analysis suite, profiling tools, and a detailed task-building specification. The suite connects model comparisons with trial trajectories, resource curves, and requirement- and test-level histories, so users can inspect where progress is acquired, lost, and recovered (Appendix G).

2 RELATED WORK

Repository and terminal benchmarks. AgentBench evaluates agents across interactive environments (Liu et al., 2024). SWE-bench and Terminal-Bench assess repository or terminal work through executable tests (Jimenez et al., 2024; Merrill et al., 2026). SWE-bench Verified strengthens task validity through expert review (OpenAI, 2024), while SWE-bench Pro extends evaluation to challenging, long-horizon repository tasks (Deng et al., 2025). To limit contamination, LiveCodeBench

Table 1: Selected coding benchmarks. Subscore: partial completion; interim score: intermediate assessment. Features refer to the stated evaluation protocol.

Benchmark	Lang.	Req. delivery	Subscore	Interim score	Perturbation
SWE-bench Verified (OpenAI, 2024)	Python	All at once	×	×	×
SWE-bench Pro (Deng et al., 2025)	Multi	All at once	×	×	×
DeepSWE (Huang et al., 2026)	Multi	All at once	×	×	×
RoadmapBench (Xu et al., 2026)	Multi	All at once	✓ Tests	×	×
SWE-Together (Wu et al., 2026)	Multi	Simulator-driven	✓ Rubrics	×	×
SWE-Interact (Raghavendra et al., 2026)	Multi	Simulator-driven	✓ Rubrics	✓ Rubrics	×
SWE-CONTINUUM	Multi	Precompiled stages	✓ Tests	✓ Tests	✓ Matched turns

collects fresh competition problems (Jain et al., 2025), and SWE-rebench automates the collection of repository tasks (Badertdinov et al., 2026). Resource-aware evaluation examines accuracy–cost trade-offs (Kapoor et al., 2024). Our focus is on how functionality accumulates, regresses, and recovers within a continuing task, rather than only its final outcome and expenditure.

Long and interactive coding tasks. Long-horizon coding benchmarks differ in how requirements and feedback are delivered. *Single-request* settings provide task specifications upfront: RoadmapBench and SWE-EVO report partial-completion scores (Xu et al., 2026; Le et al., 2025), DeepSWE uses functional verifiers (Huang et al., 2026), and LoopsBench evaluates intermediate checkpoints and regressions (Li et al., 2026). These settings measure progress toward known requirements rather than adaptation to newly arriving ones. *Multi-turn* settings introduce requirements or feedback over successive interactions. EvoCode-Bench uses persistent sessions with cumulative tests and a fail-stop rule (Shen et al., 2026), while SWE-Milestone unlocks milestones after prerequisite completion (Deng et al., 2026). User-simulator approaches such as SWE-Together provide reactive requests and feedback (Wu et al., 2026), but action-dependent feedback may expose agents to different subsequent information and complicate matched-input comparisons. SWE-CONTINUUM instead combines staged requirement release with cumulative verification under a fixed, dependency-valid schedule independent of agent behavior.

Coding agents. OpenHands supports generalist agents with code, shell, and browser interaction (Wang et al., 2025); SWE-agent develops specialized agent–computer interfaces (Yang et al., 2024); and Terminus-2 uses a terminal-centric scaffold for autonomous sandboxed execution (Merrill et al., 2026). Agentless fixes the workflow to localization, repair, and patch validation (Xia et al., 2024). For work that spans context windows, Pi persists conversation state in sessions that can be resumed or compacted (Earendil, 2026).

3 SWE-CONTINUUM

3.1 MULTI-STAGE TASK DESIGN

A task fixes an initial workspace W_0, a finite set of requirements $\mathcal{R}$, and a test suite $\mathcal{V}$:

$$\mathcal{T} = (W_0, \mathcal{R}, \mathcal{V}). \tag{1}$$

Each requirement $r \in \mathcal{R}$ specifies an observable behavior and comes with a reference patch; applying all reference patches to W_0 gives the reference implementation. Requirement r owns a nonempty set $\mathcal{F}_r$ of fail-to-pass (F2P) tests, which fail in W_0 and pass in the reference implementation. The sets $\mathcal{F}_r$ are disjoint, so every F2P test has one owner. Pass-to-pass (P2P) tests $\mathcal{P}$ pass in W_0 and guard existing behavior. The full suite is $\mathcal{V} = \mathcal{F} \cup \mathcal{P}$, where $\mathcal{F} = \bigcup_{r \in \mathcal{R}} \mathcal{F}_r$.

Stages. A task runs as a sequence of stages $t = 1, \ldots, T$. Stage t releases a set of new requirements $\Delta\mathcal{R}_t$ and then gives the agent one turn, which may contain many model and tool calls. Requirement-bearing releases partition $\mathcal{R}$; with $\mathcal{R}_0 = \varnothing$, they accumulate as

$$\mathcal{R}_t = \bigcup_{j=1}^{t} \Delta\mathcal{R}_j, \qquad \mathcal{R}_T = \mathcal{R}. \tag{2}$$

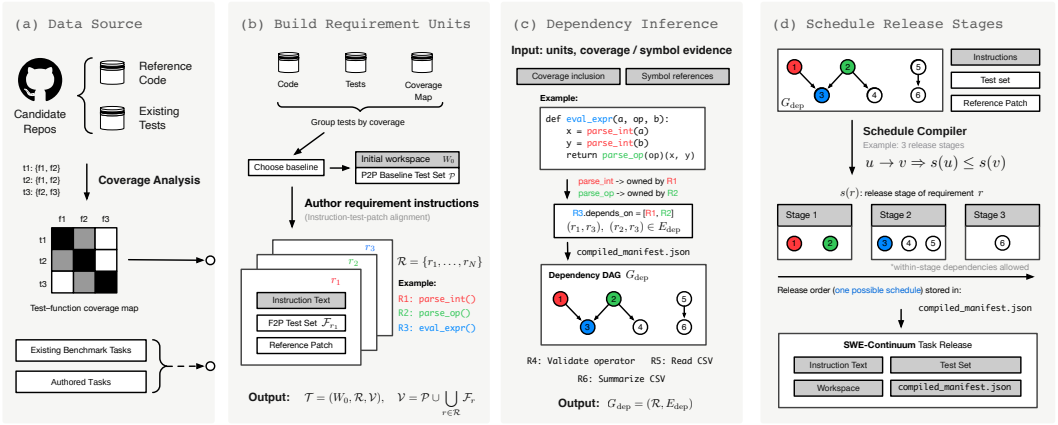


Figure 2: Task construction. (a) Repository code and tests, existing benchmark tasks, or authored tasks provide source evidence. (b) Test coverage groups code into requirements, each with an instruction, owned F2P tests, and a reference patch; P2P tests are shared. (c) Code, symbol, and test evidence yield a dependency graph $G_{\text{dep}} = (\mathcal{R}, E_{\text{dep}})$. (d) The schedule compiler assigns each requirement r a release stage $s(r)$ under a deterministic policy.

Earlier requirements stay in force, and code changes and unresolved errors carry over in the workspace. A failed test does not stop later stages.

Execution profiles. An execution profile fixes the messages the agent receives and its context policy, that is, whether the conversation resumes or restarts at each stage. The clean profile (C) sends only requirement instructions. A profile may also insert message-only stages, which release no requirement ($\Delta\mathcal{R}_t = \varnothing$) but still run one agent turn; T counts these stages. Two matched profiles use them: review (E) sends a task-agnostic request to check the preceding change, and semantic interference (I) sends relevant but incorrect advice about the preceding agent-authored change. E and I share insertion points, turn limits, and context policy, so their comparison controls the number of extra turns (Appendix A.3).

Checkpoints. After each agent turn, a verifier outside the agent environment runs the F2P tests of every released requirement and all P2P tests:

$$\mathcal{V}_t = \mathcal{F}_{\leq t} \cup \mathcal{P}, \qquad \mathcal{F}_{\leq t} = \bigcup_{r \in \mathcal{R}_t} \mathcal{F}_r. \tag{3}$$

Hidden verifier tests and future instructions stay outside the agent workspace; F2P tests of unreleased requirements are not run. The resulting checkpoint t records the test verdicts, the workspace, and cumulative resource use; checkpoint 0 records W_0. Stable test IDs link the checkpoints, so each test's history shows when it first passes, breaks, and passes again. A trial is one execution of a task by one agent under one execution profile; its checkpoint sequence is a trajectory. A trial is complete if it reaches the last planned checkpoint and valid if no infrastructure or verifier error invalidates its records (Appendix A).

3.2 Task Construction

Sources. Tasks come from three sources: repositories at a fixed commit, filtered for license, size, reproducible builds, and runnable tests; existing benchmark tasks such as DeepSWE (Huang et al., 2026); and authored tasks with instructions, tests, and reference patches. A verified pull request or release can define the scope of a repository task (Appendix F).

Requirements and test ownership. For a repository, we record which functions and methods, called source units, each test executes. Tests that execute the same units form a group, and reviewing the group's code and tests identifies the behavior it checks. We remove or stub the units of selected groups to obtain W_0 and restore them as a sequence of reference patches, merging a patch that

adds only helper code into the next one. Tests that fail on W_0 are F2P and tests that pass are P2P. Each F2P test belongs to the first patch that makes it pass, and each patch, its tests, and a written instruction form one requirement. We review this alignment before freezing test ownership. Existing and authored tasks go through the same alignment without code removal (Appendix F.2).

Dependencies. Some requirements use interfaces that others add, and some revise code that others introduce. The DAG $G_{\text{dep}} = (\mathcal{R}, E_{\text{dep}})$ records the first relation: $(u, v) \in E_{\text{dep}}$ means that implementing v relies on behavior specified by u. The edge set E_{rev} records the second: $(u, v) \in E_{\text{rev}}$ means that v revises code or an interface that u introduces. Candidate edges come from symbol references in the reference patches and from the source units that tests cover; shared coverage alone does not establish a dependency. Reference-patch ablations and human review provide checks for candidate edges (Appendix F.3).

Schedule compilation. Let $s(r)$ be the release stage of r. A valid schedule satisfies

$$s(u) \le s(v) \quad ((u, v) \in E_{\text{dep}}), \qquad s(u) < s(v) \quad ((u, v) \in E_{\text{rev}}). \tag{4}$$

Ordinary dependencies may share a stage; revisions follow an earlier release. The union of the two edge sets must be acyclic, and it usually admits several valid orders. A saved deterministic policy π therefore chooses among the ready requirements, those whose predecessors have all been released, by stable requirement ID or follows a supplied schedule satisfying Equation 4. Compilation freezes each stage's instruction and test scope; the task scheduler follows this plan regardless of agent success. Appendix F.3 gives the algorithm, and Figure 12 shows a valid and an invalid schedule.

Qualification. Before freezing a task, we check the environment build and run no-op, oracle, and dry-run executions. The no-op leaves W_0 unchanged: every F2P test must fail and every P2P test must pass. The staged oracle applies reference patches and checks each cumulative scope. We inspect the DAG and manually audit instruction–test–patch alignment, release order, and information visibility. Multiple attempts inform selection of the stable v1 tasks; failed checks lead to repair and rechecking (Appendix F.4).

Corpus. The v1 corpus retains 16 tasks from a larger candidate pool, spanning five languages, 111 requirements, and 110 stages; one stage releases two requirements. Each task combines successive feature requests in one persistent workspace, so later work must preserve earlier behavior. Task IDs use a language and serial number. Table 7 summarizes test counts and dependency depth; Appendix H provides all stage instructions and DAGs.

3.3 Evaluation Metrics

Each trial yields checkpoints $t = 0, \ldots, T$ with test verdicts and cumulative resource use. These records support multi-stage profiling through checkpoint states, test events, and budgeted net progress. Scores are defined as fractions and displayed as percentages.

Net progress. Let $y_{i,t} \in \{0, 1\}$ be the verdict of test i at checkpoint t, with 1 for a pass. F2P completion $C_{F,t}$ and P2P preservation $B_{P,t}$ are

$$C_{F,t} = \frac{\sum_{f \in \mathcal{F}_{\le t}} y_{f,t}}{|\mathcal{F}|}, \qquad B_{P,t} = \frac{\sum_{p \in \mathcal{P}} y_{p,t}}{|\mathcal{P}|}, \qquad S_t = C_{F,t} + B_{P,t} - 1, \tag{5}$$

and net progress S_t is the fraction of F2P tests passed minus the fraction of P2P tests broken. It lies in $[-1, 1]$, with $S_0 = 0$ and $S_T = 1$ when every test in $\mathcal{V}$ passes. Because $C_{F,t}$ divides by all F2P tests rather than the released ones, S_t is comparable across stages; unreleased tests earn no credit. If $\mathcal{P}$ is empty, $B_{P,t} = 1$. Requirement completion $C_{R,t}$ is the fraction of requirements whose owned F2P tests all pass (Appendix B.1). Terminal scores use $t = T$.

Test events. Comparing consecutive checkpoints classifies each test change. An F2P test's first pass after release is an acquisition. A pass-to-fail change is a regression. A later fail-to-pass change is a recovery if the test had passed before; otherwise it is a delayed acquisition. P2P tests start passing, so their failures are regressions and their returns are recoveries. A regression can leave S_t unchanged or rising when other tests gain at the same checkpoint, so we report event counts alongside S (Appendix B.2).

AUPEC. Terminal S and total expenditure do not show when progress was made or lost. Let $E_{t,d}$ be the cumulative expenditure at checkpoint t on resource axis d (tokens, agent steps, agent time, or cost), with $E_{0,d} = 0$. The Progress–Expenditure Curve (PEC) $\phi_d(e)$ is the net progress of the last checkpoint reached within expenditure e. For a budget $B > 0$, the Area Under the PEC is

$$\phi_d(e) = S_{\max\{t:E_{t,d}\le e\}}, \qquad \mathrm{AUPEC}_{S,d}(B) = \frac{1}{B}\int_0^B \phi_d(e)\,\mathrm{d}e. \tag{6}$$

AUPEC is the average net progress held over the budget. It lies in $[-1, 1]$ like S, rewards earlier progress, and penalizes delayed gains and net losses, including broken P2P behavior, for the expenditure over which they persist (Figure 3). It measures budgeted net progress; losses offset by concurrent gains still require test-level event records.

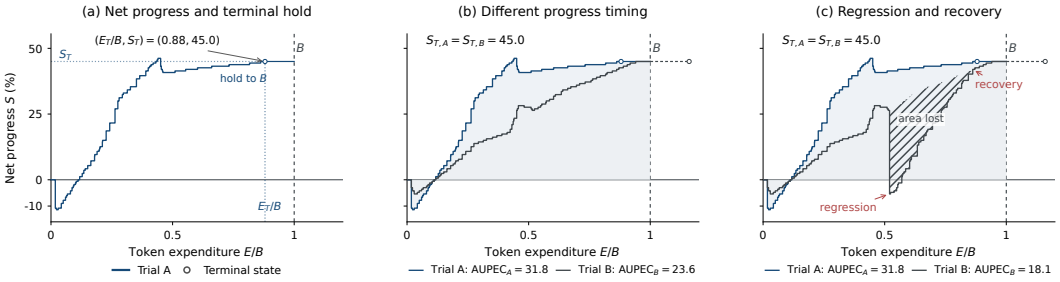


Figure 3: AUPEC on synthetic trials. The curves plot S; C_F, B_P, and C_R are not shown. (a) Terminal state; a complete trial ending at $E_T < B$ holds its terminal score to B. (b) Two trials with equal terminal S but different AUPEC, the shaded area over $[0, B]$ divided by B; progress after B is dashed and excluded. (c) Regression and recovery lower AUPEC by the hatched area; area with negative S is subtracted.

Budget. A complete trial that ends before B holds its terminal S until B. For a trial that ends after B, only the part up to B counts. Incomplete trials are excluded. Each comparison uses one B per axis for all trials: the 90th percentile (P90) of terminal expenditure over valid complete trials. P90 limits the influence of long expenditure tails and concentrates comparison on a shared practical budget. We freeze its numerical value for the reported comparison; additional models can be evaluated at that same budget. We write $S(B) = \phi_d(B)$ for the net progress held at the budget. Appendix B.3 gives the discrete form and resource accounting.

4 Results

4.1 Experimental Setup

We evaluate eight models, with the short names used throughout in parentheses: GPT-6 Astra (Astra), GPT-5.6 Luna (Luna) (OpenAI, 2026a;b), DeepSeek-V4-Flash-0731 (DS Flash V4-0731), DeepSeek-V4.1-Flash (DS Flash V4.1) (DeepSeek, 2026), MiMo V2.5 (MiMo) (Xiaomi, 2026), GLM 5.3 Flash (GLM) (AutoClaw Team, 2026), Qwen 3.8 Flash (Qwen) (Alibaba Cloud, 2026), and Hy4 Preview (Hy4) (Tencent, 2026). All runs request High effort through the adapter.

For convenient development and affordable small-scale evaluation, we designate nine of the 16 tasks as a primary subset, selected from pilot runtime and trajectory checks. The model comparison uses this subset; Qwen has eight valid tasks, MiMo seven, and the others nine. The calibration also evaluates all 16 tasks (Appendix D). All model-comparison runs use Harbor 0.20.0 (Harbor, 2026b), Pi 0.84.2, the clean profile, and a 128K-token context limit. Astra has one planned attempt per task and the other models two, giving 135 scheduled trials. Pi resumes the conversation and workspace after each verified stage and controls context compaction, which summarizes earlier turns to remain within the context limit. All reported trials use resumed sessions; execution details appear in Appendix A.

We average valid trials within each task and then weight tasks equally (task-macro averaging). For model comparison, terminal C_F, C_R, B_P, and S describe delivered function; test-level transitions

expose continuity; AUPEC measures net progress within a shared $B = 63.15$ million-token budget. Tokens sum inclusive input and output, counting cached input once (Appendix B.3). Agent steps count trace records, distinct from release stages. Compactions, wall time, and historical USD cost provide further resource measures. Wall time includes verification; historical cost combines recorded charges and estimates.

RQ1. To what extent does staged verification reveal regressions obscured by aggregate and terminal scores under clean and perturbed execution?
RQ2. How do models differ in functional performance and resource expenditure?
RQ3. What information do progress trajectories add to terminal evaluation across resource budgets?

4.2 Regression Observability under Clean and Perturbed Execution

To address RQ1, we quantify regressions that aggregate and terminal scores cannot identify. Final target-level scores, as in RoadmapBench (Xu et al., 2026), measure partial completion; cumulative test histories additionally resolve losses offset by concurrent gains or followed by recovery.

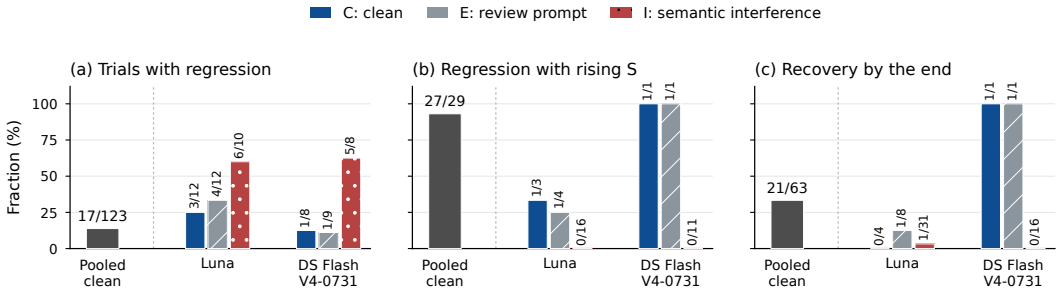


Figure 4: Regression observability. (a) Trials with regression / valid trials. (b) Regression checkpoints with rising S / all regression checkpoints. (c) Regressed trial–test pairs passing at the end / all regressed pairs. Gray bars pool the clean model-comparison trials. C/E/I denote clean execution, a review prompt, and semantic interference in a separate batch (59/72 valid trials). Luna covers four tasks; DS Flash V4-0731 covers three. Counts are pooled within each model and profile; C has fewer checkpoints.

Clean execution. Of 123 valid trials, 17 across five tasks contain a regression, yielding 64 pass-to-fail transitions. Net progress still rises at 27 of the 29 regression checkpoints, and 21 of 63 distinct regressed trial–test pairs pass again at the end (Figure 4). Both aggregate progress and terminal verdicts therefore conceal intervening functional losses: gains on newly released tests can outweigh losses on earlier tests, and later repairs can erase those losses from the final state. Table 4 gives task-level counts; Figure 8 illustrates both forms of hidden regression.

Semantic interference. A separate four-task batch yields 59 valid trials from 72 planned runs of Luna and DS Flash V4-0731. Clean (C) adds no turns; a task-agnostic review prompt (E) and semantic interference (I) each add one turn after every requirement-bearing stage, matching execution opportunities. Under I, regressions occur in 6/10 Luna and 5/8 DS Flash V4-0731 trials, compared with 4/12 and 1/9 under E (Figure 4). None of I's 27 regression checkpoints has rising S, and only 1/47 regressed trial–test pairs recovers by the end. Thus, losses under I are predominantly persistent and visible in the score; checkpoint histories locate their onset and retain the one recovered loss. At inserted message checkpoints, the fraction of previously passing tests lost is higher under I by 1.02 percentage points for Luna and 0.88 for DS Flash V4-0731, after averaging within four and three matched tasks, respectively. Appendix A.3 reports exclusions, uncertainty intervals, and terminal and resource outcomes.

4.3 Functional Performance and Resource Expenditure

RQ2 examines whether comparable functional performance entails comparable resource expenditure. Figure 5 reports both dimensions; Table 5 retains the numerical results, including baseline preservation.

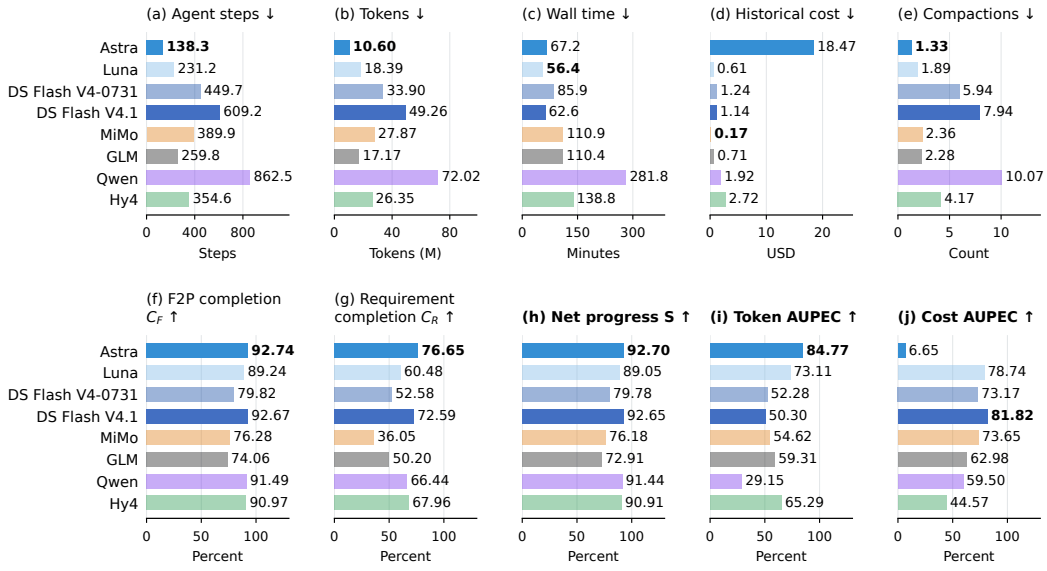


Figure 5: Eight-model comparison (Pi, High effort, clean). Colors match Figure 6; opaque bars highlight Astra and DS Flash V4.1. Bold numbers mark the best displayed values; arrows indicate the preferred direction. Means use the aggregation and coverage in Section 4.1. The AUPEC budgets are 63.15M tokens and $3.32. Cost AUPEC revalues checkpoint tokens at the public prices in Appendix E.1; panel (d) retains historical costs. Wall time includes verification.

Of 135 scheduled trials, 123 are valid. Astra has the highest displayed terminal S (92.70) and token AUPEC (84.77). DS Flash V4.1 reaches similar S (92.65) but uses 49.26 versus 10.60 million tokens and has AUPEC 50.30 versus 84.77. Its traces also contain more agent steps (609.2 versus 138.3) and compactions (7.94 versus 1.33), consistent with more repeated interaction and context processing. Wall time remains comparable (62.6 versus 67.2 minutes), so token volume does not directly translate into elapsed time. The panels distinguish eventual progress from the resources required to obtain it. Figure 6 plots the token-budget progress of all eight models, with the same aggregation as Figure 5.

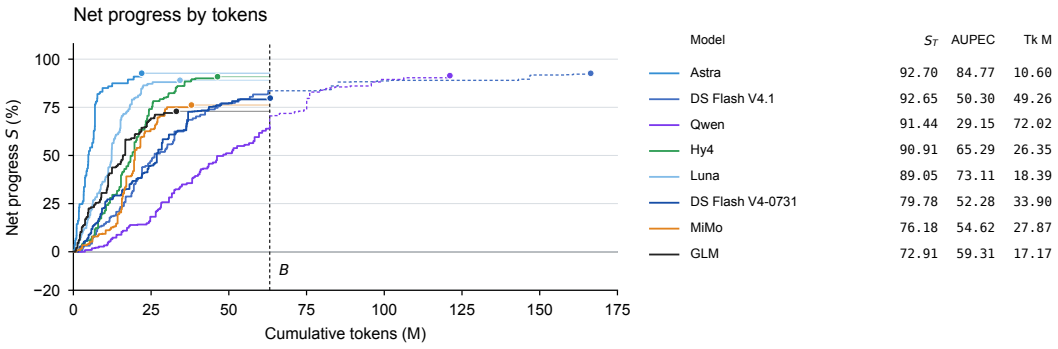


Figure 6: Token PECs, with Figure 5's aggregation and key (terminal S_T, AUPEC, mean terminal tokens). Dots mark aggregate endpoints at the largest observed expenditure; faint lines hold early finishes to $B = 63.15$M, and dashes show post-budget progress.

5 Analysis

For RQ3, 63 trial pairs with identical final test verdicts differ by 17.65 token-AUPEC points on average across tasks (Appendix C.1). Eight pairs also have total token expenditure within 10%; their task-macro gap is 3.55 points. These approximate matches retain variation in progress timing. Terminal S and token AUPEC reverse 14 of 28 model-pair rankings. Figure 7 compares progress and AUPEC across resource axes.

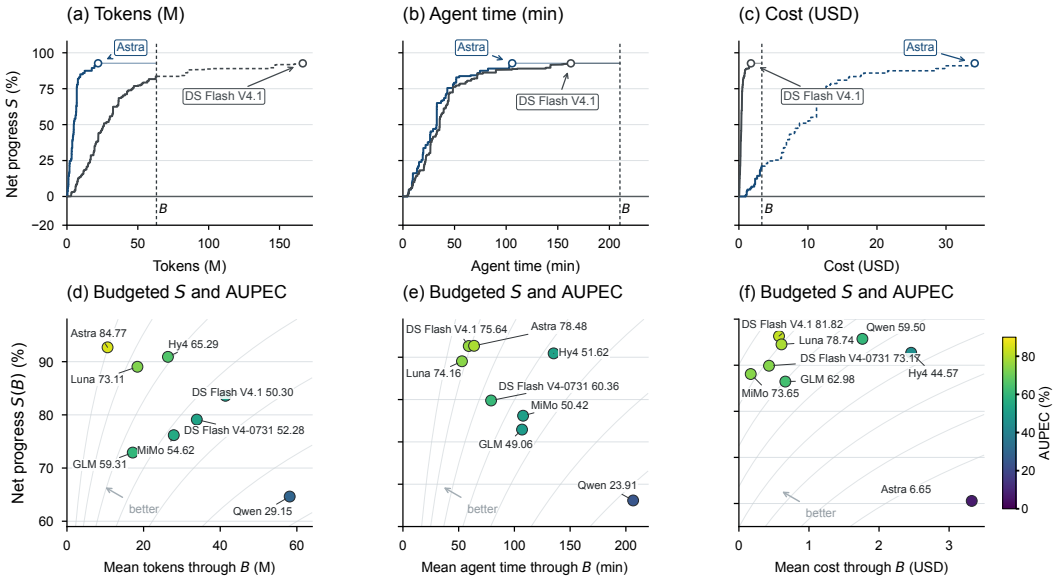


Figure 7: Token, agent-time, and cost comparisons. Top: Astra and DS Flash V4.1 PECs, with terminal dots, early-finish extensions, and post-budget dashes. Bottom: task-macro spend through B versus $S(B)$, colored by AUPEC; contours extrapolate a descriptive eight-point fit, not the AUPEC definition. Pooled P90 budgets: 63.15M tokens, 210.23 minutes, $3.32. Cost uses public-price revaluation; agent time excludes verification.

The resource axis changes the comparison. Astra versus DS Flash V4.1 AUPEC is 84.77 versus 50.30 for tokens, 78.48 versus 75.64 for agent time, and 6.65 versus 81.82 for cost. Cost uses public-price revaluation (Appendix E.1), while Figure 5(d) retains historical costs. The ordering depends on the resource constraint and pricing convention.

6 Conclusion

SWE-Continuum provides a task construction and evaluation paradigm for multi-stage behavioral continuity. Fixed requirement schedules, persistent workspaces, and cumulative verification show when functionality is acquired, lost, and recovered. Terminal capability, test-level continuity, and AUPEC summarize complementary aspects of this history; terminal scores alone cannot reconstruct it.

Scope and limitations. Repeated execution and cumulative verification make trials costly. Within available compute, we prioritize eight strong proprietary and open-weight models rather than every available model. The collection spans 16 repository tasks, five languages, and 111 requirements over 110 stages. These successive requests provide depth within continuing workspaces; broader task and model coverage remains necessary for generalization.

Extensibility. Researchers can construct tasks for their chosen repositories and goals by aligning instructions, tests, and reference patches, then compiling requirement dependencies into valid release schedules. New tasks reuse the same cumulative verification and trajectory metrics.

Reproducibility Statement

Sections 3 and 4.1 specify task construction, verification, metrics, and execution settings. Appendices A–G provide aggregation, exclusions, pricing, and reproduction details; Appendix H provides the complete stage instructions and requirement DAGs. Figure scripts and frozen plotting inputs accompany the manuscript.

AI Use Statement

We used generative AI to assist with task, test, and reference-code construction; feedback on methodology and experiments; data processing, analysis code, result interpretation, and figures; literature search and reference formatting; and manuscript drafting and revision. AI-assisted task artifacts were checked through environment builds, no-op and oracle runs, dry runs, and human task and DAG review. Analysis was checked against checkpoint records, and the authors reviewed the manuscript and take responsibility for its content.

References

Alibaba Cloud. Qwen3.8-Flash: Model information, 2026. URL https://help.aliyun.com/en/model-studio/qwen3-8-flash. Accessed September 26, 2026.

AutoClaw Team. GLM-5.3-Flash: More intelligence with less compute, 2026. URL https://autoclaw.z.ai/blog/model/glm-5.3-flash/. Accessed September 26, 2026.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

DeepSeek. DeepSeek API change log, 2026. URL https://api-docs.deepseek.com/updates/. Accessed September 26, 2026.

Gangda Deng, Zhaoling Chen, Zhongming Yu, Haoyang Fan, Yuhong Liu, Yuxin Yang, Dhruv Parikh, Rajgopal Kannan, Le Cong, Mengdi Wang, et al. Swe-milestone: Evaluating ai agents on continuous software evolution. Proceedings of the 43rd International Conference on Machine Learning (ICML 2026), 2026.

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

Earendil. Pi: Coding agent and session documentation, 2026. URL https://github.com/earendil-works/pi. Session documentation: https://github.com/earendil-works/pi/blob/main/packages/coding-agent/docs/sessions.md. Accessed September 26, 2026.

Harbor. RFC 0001: Agent trajectory interchange format (ATIF) specification, 2026a. URL https://github.com/harbor-framework/harbor/blob/main/rfcs/0001-trajectory-format.md. Accessed September 26, 2026.

Harbor. Harbor: Framework for evaluating and improving agents, 2026b. URL https://github.com/harbor-framework/harbor. Accessed September 26, 2026.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, et al. Data interpreter: An llm agent for data science. In *Findings of the association for computational linguistics: ACL 2025*, pp. 19796–19821, 2025.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026.

Geoffrey Huntley. Ralph Wiggum as a "software engineer", 2025. URL https://ghuntley.com/ralph/. Accessed September 26, 2026.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

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? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Sayash Kapoor, Benedikt Stroebl, Zachary S Siegel, Nitya Nadgir, and Arvind Narayanan. Ai agents that matter. *arXiv preprint arXiv:2407.01502*, 2024.

Tue Le, Minh VT Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi DQ Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025.

Han Li, Zhemin Fang, Rili Feng, Yingqi Zhao, Jiaheng Liu, Pengfei Gao, He Ye, Dayi Lin, Qingwei Lin, Saravan Rajmohan, et al. Loopsbench: From harness engineering to loop engineering in benchmarking coding agent. *arXiv preprint arXiv:2608.00267*, 2026.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

OpenAI. Introducing SWE-bench Verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/. Accessed September 26, 2026.

OpenAI. GPT-6 Astra: Model documentation, 2026a. URL https://developers.openai.com/api/docs/models/gpt-6-astra. Accessed September 26, 2026.

OpenAI. GPT-5.6 Luna: Model documentation, 2026b. URL https://developers.openai.com/api/docs/models/gpt-5.6-luna. Accessed September 26, 2026.

Mohit Raghavendra, Anisha Gunjal, Aakash Sabharwal, and Yunzhong He. Swe-interact: Reimagining swe benchmarks as user-driven long-horizon coding sessions. *arXiv preprint arXiv:2606.30573*, 2026.

Haiyang Shen, Xuanzhong Chen, Wendong Xu, Yun Ma, Liang Chen, and Kuan Li. Evocode-bench: Evaluating coding agents in multi-turn iterative interactions. *arXiv preprint arXiv:2605.24110*, 2026.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In *International conference on machine learning*, pp. 31210–31227. PMLR, 2023.

Tencent. Tencent releases and open-sources Tencent Hy4 Preview, 2026. URL https://www.tencent.com/tencent-releases-and-open-sources-tencent-hy4-preview/. Accessed September 26, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, et al. Swe-together: Evaluating coding agents in interactive user sessions. *arXiv preprint arXiv:2606.29957*, 2026.

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

Xiaomi. Xiaomi MiMo-V2.5 series open-sourced and Orbit 100 trillion token plan launched, 2026. URL `https://mimo.mi.com/docs/en-US/news/latest/v2.5-open-sourced`. Accessed September 26, 2026.

Xinbo Xu, Ruihan Yang, Haiyang Shen, Wendong Xu, Bofei Gao, Ruoyu Wu, Kean Shi, Weichu Xie, Xuanzhong Chen, Ming Wu, et al. Roadmapbench: Evaluating long-horizon agentic software development across version upgrades. *arXiv preprint arXiv:2605.15846*, 2026.

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

APPENDIX CONTENTS

A Task Execution and Intervention Profiles **14**
A.1 Execution Units and Persistent State 14
A.2 Verification and Validity 14
A.3 Matched Intervention Profiles 14
A.4 Requirement Verification Across Checkpoints 16

B Metric Definitions and Aggregation **16**
B.1 Functional State 16
B.2 Acquisition, Regression, and Recovery 16
B.3 Resource Accounting and AUPEC 17
B.4 Aggregation and Coverage 17

C Trajectory and Ranking Diagnostics **17**
C.1 Information Beyond the Final Score 18
C.2 Model Rankings and Efficiency 19

D Task-Level Calibration **20**

E Model Results and Cost Accounting **20**
E.1 Historical Cost and Public-Price Revaluation 20

F Repository Selection and Verification Pipeline **21**
F.1 Source Selection and Task Structure 21
F.2 Source-Specific Requirement Construction 22
F.3 Schedule Compilation and Edge Evidence 22
F.4 Qualification and Audit 24

G Reproducibility **24**

H V1 Task Instructions and Requirement DAGs **24**
H.1 Task instructions 25

A Task Execution and Intervention Profiles

A.1 Execution Units and Persistent State

A task fixes a problem; an execution profile fixes its messages and context policy; a trial executes that profile under an agent configuration. Its checkpoints form a trajectory. Profiles share initial workspace, requirements, and tests. Task counts exclude message variants and repeated trials.

We use Harbor (Harbor, 2026b) to execute each stage: instruction, agent turn, and verification checkpoint. The turn ends when the agent returns or reaches its limit and can include many model or tool calls. An agent step is a trace record that may summarize multiple model calls; it is distinct from a stage or an agent turn.

Workspace state persists across stages, including code, Git state, and saved files. Context either resumes the same session or starts afresh. Records identify this policy, access to prior instructions and handoff files, adapter version, and session lineage. Independent trials restart from W_0.

Pi supports both resumed sessions and fresh conversations over the persistent workspace. The latter resembles the context-reset component of Ralph-style loops (Huntley, 2025); the reported experiments use resumed sessions. DeepSWE uses mini-swe-agent for single-task rollouts (Huang et al., 2026), whereas this protocol continues execution after each requirement release.

Functional failure does not stop the schedule. Runtime interruptions retain completed checkpoints and receive a separate termination status. A complete trial reaches the profile's last planned checkpoint.

Model names follow the frozen experiment records. The model comparison and calibration use `deepseek/deepseek-v4-flash` for DS Flash V4-0731; the model comparison uses `deepseek/deepseek-flash` for DS Flash V4.1. The intervention batch uses the explicit `deepseek-v4-flash-0731` route.

A.2 Verification and Validity

The verifier reconstructs the current workspace in a separate environment, using a complete artifact or a fixed baseline with a binary-safe patch and untracked files. It uses pinned dependencies; hidden tests, future instructions, and reference solutions remain outside the agent workspace.

Test selection precedes collection and execution and equals $\mathcal{V}_t$ in Equation 3. A registry maps stable IDs to framework selectors and requirements. Checkpoint records save the scope, outcomes, logs, recoverable workspace, agent trace, and resource counters. Agent traces use the Agent Trajectory Interchange Format (ATIF) (Harbor, 2026a).

Functional failures and invalid records are distinct. Assertions and timeouts caused by agent code count as failures. Transfer, selection, verifier, or infrastructure errors invalidate affected observations; missing resource fields remain missing on that axis. The model-comparison batch retains 123 of 135 scheduled trials. Eight exclusions are marked infrastructure error only, two nonfunctional test status only, one both, and one is right-censored before the terminal checkpoint. These labels describe record validity, not functional failure. All 135 records identify Harbor 0.20.0 and Pi 0.84.2. The separate intervention batch has the exclusions below.

A.3 Matched Intervention Profiles

The completed intervention batch uses Harbor 0.20.0 and bundled Pi 0.73.1 with resumed sessions and High effort. It crosses four source tasks (typescript-03 async container initialization, rust-01 cached recipes, python-06 request coalescing v3, and python-07 RaisesGroup evolution), three profiles, two models, and three attempts, giving 72 planned trials. The models are Luna and DS Flash V4-0731; the latter is the configured `deepseek-v4-flash-0731` endpoint. Each trial uses one CPU and 4 GiB of memory; retries are disabled. This batch is separate from the model comparison.

Profiles fix message content and insertion positions before execution. Clean (C) supplies requirements alone. The review control (E), stored as `encouragement-every-step-v2`, asks the agent to check its preceding change against requirements already received and fix any omissions. It supplies no task-specific facts or implementation advice. Semantic interference (I) presents relevant but

incorrect advice about the preceding agent-authored change. E and I use matched positions, execution limits, context policies, and cumulative test scopes. They introduce no requirements. In this batch, E and I insert one message-only stage after every requirement-bearing stage, including the last; C adds none.

All trials have terminal records, but only 59 satisfy the complete-trial validity criteria. Eleven are excluded for out-of-scope test reports (including all nine DS Flash V4-0731 typescript-03 trials), one Luna typescript-03 trial has a non-binary test outcome, and one Luna python-06 trial has a verifier exception. Valid C/E/I counts are 12/12/10 for Luna and 8/9/8 for DS Flash V4-0731. Consequently, Luna covers all four source tasks, whereas DS Flash V4-0731 covers rust-01, python-06, and python-07. Excluded trials do not contribute failures or zero scores.

Profile comparisons hold source task, model, harness, and effort fixed. The primary I–E contrast compares semantic interference with the review control under equal numbers of agent turns and verification opportunities. I–C also changes both counts. Profiles run as separate trials; the configuration does not specify shared-prefix branches. We average valid attempts within each task and profile, subtract E from I on tasks observed in both conditions, and average the task differences. The 95% percentile intervals enumerate all ordered bootstrap resamples of the eligible source tasks (four for Luna, three for DS Flash V4-0731); they describe variation within this small task set. Each outcome reports eligible coverage and actual resource use.

For RQ1, Figure 4 counts events over complete valid trials. A regression checkpoint contains at least one pass-to-fail transition; its loss is concealed by net gains when $\Delta S > 0$. Terminal recovery counts distinct regressed trial–test pairs that pass at the final checkpoint, even if they regressed more than once. These whole-trial summaries are separate from the immediate intervention effect in Table 2. For this batch, the at-risk set contains all already released F2P and P2P tests passing immediately before a message. A trial's immediate loss rate sums losses over its message checkpoints and divides by the summed at-risk counts. Recovery follows each message–test loss event to the final checkpoint; repeated losses of a test remain separate events in this measure. Empty denominators are not applicable. Each profile has its own at-risk set, so the contrast measures loss under that profile, not a shared-state intervention effect.

Token expenditure sums Pi's per-call `usage.totalTokens`, checked against Harbor's input-plus-output counters. In this adapter, input already includes cache reads; adding the cache counter again would double-count them. The shared budget is 34.95M tokens (34,949,490), the pooled P90 of terminal tokens over all 59 valid C/E/I trials. AUPEC uses checkpoint-held S and extends completed trials to this budget. This batch has its own budget; its AUPEC values are not directly comparable with the model table.

Table 2: Task-matched outcomes under review prompts (E) and semantic interference (I).

Outcome	E	I	I–E [95% CI]	Tasks; trials E/I
Luna				
Immediate test loss (%)	0.32	1.34	$+1.02$ $[0.00, 2.82]$	4; 12/10
Message losses restored at end (n/N)	0/5	1/22	–	4; 12/10
Terminal S (%)	85.64	61.27	-24.37 $[-47.82, -8.49]$	4; 12/10
Token AUPEC (%)	59.91	49.08	-10.83 $[-23.18, -0.82]$	4; 12/10
Total tokens (M)	19.94	17.36	-2.58 $[-7.57, 0.67]$	4; 12/10
DS Flash V4-0731				
Immediate test loss (%)	0.00	0.88	$+0.88$ $[0.31, 1.41]$	3; 9/8
Message losses restored at end (n/N)	N/A (0/0)	0/16	–	3; 9/8
Terminal S (%)	67.38	53.04	-14.34 $[-30.85, 0.36]$	3; 9/8
Token AUPEC (%)	43.84	37.88	-5.96 $[-16.23, 2.26]$	3; 9/8
Total tokens (M)	30.34	25.95	-4.39 $[-5.31, -3.68]$	3; 9/8

I–E averages within-task differences over common eligible tasks; 95% intervals resample tasks. Loss and score differences are percentage points; token differences are millions. Recovery counts message–test loss events; E/I can lose different tests, so no recovery-rate contrast is specified. N/A denotes an empty denominator. Tokens count cache reads once; the AUPEC budget is 34.95M tokens.

A.4 Requirement Verification Across Checkpoints

A supplementary calibration example, separate from the model-comparison batch, records eight checkpoints in Table 3 for `go-01` with Pi and DS Flash V4-0731 at High effort, attempt 1. Each cell shows passing/total tests: the first row tracks baseline P2P and the remaining rows track requirement F2P. R1 and R2 first complete at checkpoint 5. At checkpoint 8, P2P loses four passing tests and R1, R4, and R7 lose one, one, and eight; R3–R8 are incomplete at the end. The ratios show both completion and losses within incomplete requirements.

Table 3: Requirement × checkpoint verification for `go-01` (Pi / DS Flash V4-0731 / High, attempt 1). CPt is checkpoint t; cells give passed/total tests.

	CP1	CP2	CP3	CP4	CP5	CP6	CP7	CP8
Baseline P2P (13)	**13/13**	**13/13**	**13/13**	**13/13**	**13/13**	**13/13**	**13/13**	9/13
R1 (11 F2P)	8/11	8/11	8/11	8/11	**11/11**	**11/11**	**11/11**	10/11
R2 (3 F2P)	–	0/3	0/3	0/3	**3/3**	**3/3**	**3/3**	**3/3**
R3 (8 F2P)	–	–	0/8	0/8	7/8	7/8	7/8	7/8
R4 (7 F2P)	–	–	–	0/7	5/7	5/7	6/7	5/7
R5 (39 F2P)	–	–	–	–	35/39	36/39	36/39	36/39
R6 (29 F2P)	–	–	–	–	–	26/29	26/29	26/29
R7 (13 F2P)	–	–	–	–	–	–	12/13	4/13
R8 (22 F2P)	–	–	–	–	–	–	–	6/22

Green bold: all released tests pass. Red underline: fewer pass than at the previous checkpoint. Gray dash: unreleased.

B Metric Definitions and Aggregation

B.1 Functional State

Test states exist only within the released scope: $y_{i,t} = 1$ for a pass and 0 for a nonpassing outcome, with its reason retained. Invalid observations are excluded. Qualification of the initial workspace gives $C_{F,0} = C_{R,0} = S_0 = 0$ and $B_{P,0} = 1$.

Requirement completion uses all tests assigned to that requirement. Define

$$z_{r,t} = \begin{cases} \prod_{f \in \mathcal{F}_r} y_{f,t}, & r \in \mathcal{R}_t, \\ 0, & r \notin \mathcal{R}_t, \end{cases} \qquad C_{R,t} = \frac{1}{|\mathcal{R}|} \sum_{r \in \mathcal{R}} z_{r,t}. \tag{7}$$

Unreleased requirements receive no credit. Scores weight F2P tests equally in C_F, P2P tests equally in B_P, and requirements equally in C_R. If $\mathcal{P}$ is empty, we set $B_P = 1$ when computing S and report zero P2P tests.

B.2 Acquisition, Regression, and Recovery

Stable test IDs distinguish acquisition from recovery. If test f is first released at checkpoint $\rho(f)$, a pass there is new-scope acquisition. For $t > \rho(f)$, define the prior pass history $H_{f,t-1} = \max_{\rho(f) \le u < t} y_{f,u}$:

$$g^{\text{delayed}}_{f,t} = \mathbf{1}[y_{f,t-1} = 0,\ y_{f,t} = 1,\ H_{f,t-1} = 0], \tag{8}$$

$$g^{\text{recovery}}_{f,t} = \mathbf{1}[y_{f,t-1} = 0,\ y_{f,t} = 1,\ H_{f,t-1} = 1], \tag{9}$$

$$\ell^{\text{regression}}_{f,t} = \mathbf{1}[y_{f,t-1} = 1,\ y_{f,t} = 0]. \tag{10}$$

P2P tests start in a passing state, so a subsequent fail-to-pass transition is recovery. Requirement-level events use $z_{r,t}$ with the same history rule. Counts are calculated within each valid trial before aggregation.

B.3 Resource Accounting and AUPEC

Cumulative expenditure is $E_{t,d} = \sum_{j=1}^{t} e_{j,d}$ for nonnegative stage increments. Agent time excludes verification; wall time includes it. Cost uses recorded request and retry charges where available, otherwise the saved pricing snapshot. Missing increments make subsequent cumulative values unavailable on that axis.

Token expenditure is inclusive input plus output. Inclusive input already contains cache reads, so cache is counted once. The model comparison and intervention batch use this same convention; each batch retains its own frozen budget. Cache-hit fractions divide cache reads by inclusive input (Appendix E.1).

AUPEC integrates the checkpoint step function in Equation 6. For a complete trial and budget $B > 0$, its exact discrete form is

$$\mathrm{AUPEC}_{S,d}(B) = \frac{1}{B} \sum_{t=0}^{T} S_t \left[\min(B, E_{t+1,d}) - \min(B, E_{t,d}) \right], \qquad E_{T+1,d} = +\infty. \tag{11}$$

The last interval holds a complete trial's state to B; zero-length intervals contribute zero area. Incomplete trials retain only their observed prefixes and are excluded from the full-budget comparison. Their last state is never extended beyond observation.

Each comparison uses one saved budget per resource axis: the pooled P90 terminal expenditure of its valid complete trials (Section 3.3). This quantile weights trials equally, whereas reported scores weight tasks equally. With normalized expenditure $x = e/B$, AUPEC equals $\int_0^1 \phi_d(Bx)\,\mathrm{d}x$. A complete trial's terminal point $(E_{T,d}/B, S_T)$ may occur before or after $x = 1$.

B.4 Aggregation and Coverage

Scores are averaged within source tasks before averaging across Q eligible tasks. For K_q valid trials of task q and a trial-level metric M,

$$\overline{M}_q = \frac{1}{K_q} \sum_{k=1}^{K_q} M_{q,k}, \qquad \overline{M} = \frac{1}{Q} \sum_{q=1}^{Q} \overline{M}_q. \tag{12}$$

Each source task has equal weight. Profiles remain separate, and paired effects use shared tasks and matched configurations. Valid functional failures remain included; missing checkpoints, invalid records, and unavailable fields have separate coverage counts.

Uncertainty estimates preserve the source-task grouping. A task-level bootstrap samples source tasks with replacement, retaining their trial records; paired comparisons retain both conditions for each sampled task. Any additional resampling of repeated trials is specified with the reported interval.

Terminal-history comparisons use unordered pairs within the same task and profile. Exact matching uses $C_{F,T}$, $(C_{F,T}, B_{P,T})$, or the complete final test vector. Each entry averages absolute trial-metric differences within tasks, then across tasks with at least one pair. Resource comparisons require complete records on the same axis and budget.

The descriptive rankings in Appendix C.2 use each model's available tasks, matching the main results. They can reflect differences in task coverage as well as model behavior; they are not a comparison on a common complete cohort. Kendall's τ_b is $(n_c - n_d)/\sqrt{(n_c + n_d + n_x)(n_c + n_d + n_y)}$, where n_c and n_d count concordant and discordant configuration pairs and n_x, n_y count pairs tied only in one ranking. Pairs tied in both rankings contribute to neither term. Reversed pairs have opposite strict orderings; tied pairs are reported separately. A zero denominator makes τ_b undefined.

C Trajectory and Ranking Diagnostics

The diagnostics use the clean/Pi/High model-comparison trajectories. Terminal-state comparisons hold task, harness, effort, and execution profile fixed.

Table 4: Pooled and per-task regression accounting (clean/Pi/High).

Task	Regressed trials	F2P P→F	P2P P→F	Rising S	Final pass
go-02	1/13	0	3	0/1	0/3
javascript-01	1/15	3	0	1/1	0/3
javascript-02	0/15	0	0	–	–
python-02	13/14	4	42	24/24	20/45
python-03	0/14	0	0	–	–
python-05	1/11	10	1	1/2	1/11
python-06	1/11	1	0	1/1	0/1
rust-02	0/15	0	0	–	–
typescript-02	0/15	0	0	–	–
Pooled total	17/123	18	46	27/29	21/63

Regressed trials = trials with at least one pass-to-fail event / valid trials. F2P and P2P columns count transitions. Rising S = regression checkpoints with positive net ΔS / all regression checkpoints. Final pass counts distinct regressed trial–test pairs passing at the final checkpoint; one test regressed twice, so 64 transitions represent 63 distinct pairs. A dash in a measured row means there is no regression denominator.

The pooled counts establish that regression occurs and can be hidden by aggregate or terminal scores. The per-task rows describe its distribution in these runs; they do not estimate a common regression rate across tasks.

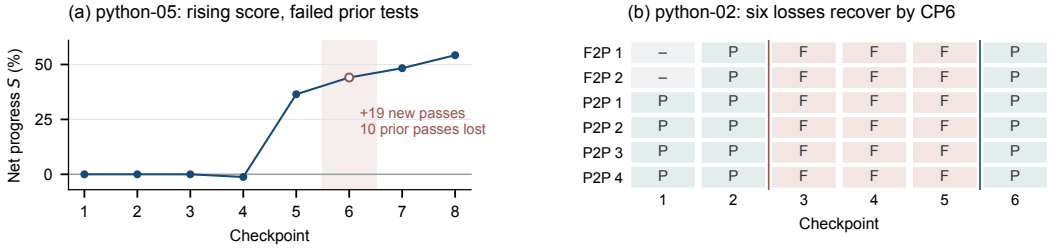


Figure 8: Two valid trial histories. (a) MiMo on python-05: at checkpoint 6, 19 new F2P tests pass while ten prior passes fail, despite rising S. (b) GLM on python-02: two F2P tests first released at checkpoint 2 and four baseline P2P tests pass, all six fail at 3, and pass again at 6. P/F mean pass/fail; gray dashes precede F2P release.

In python-05, the stage-6 patch changes request matching in the app context and re-raises `NotFound` for unmatched URLs. The verifier reports that exception in previously passing request tests. Nineteen newly released F2P tests pass at the same checkpoint, so S rises by 7.63 points despite ten F2P losses; those ten remain failed at the terminal checkpoint. In python-02, the stage-3 patch parses structured `nosec` directives by physical source line. The verifier fails tests for an end marker inside a multiline statement and for combining directive scopes in the metrics path. At stage 6 the patch propagates suppression across logical statement spans, and the six selected tests pass again; three other tests regress at checkpoint 6. We identify these mechanisms from the patches and verifier logs.

C.1 Information Beyond the Final Score

Matching terminal states reveals history omitted by endpoint scores. Within a task at fixed profile, harness, and effort, we match terminal C_F, (C_F, B_P), or the full test-verdict vector. Figure 9 shows task-level mean absolute token-AUPEC differences; diamonds average tasks equally. Final-verdict matches cover 63 trial pairs on eight tasks and differ by 17.65 percentage points on average. Pairs need not be independent because a trial may participate in several matches.

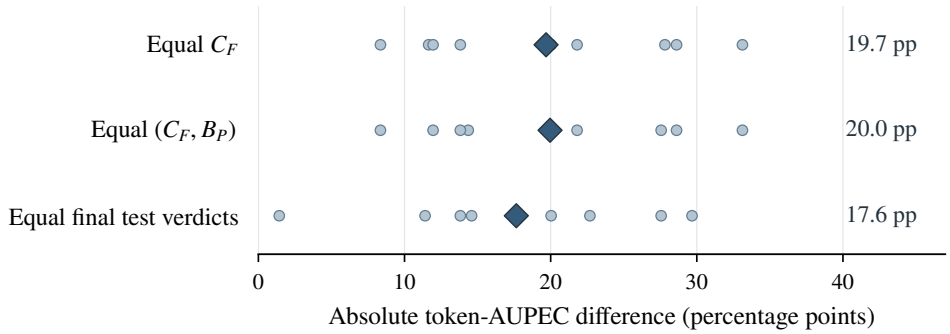


Figure 9: Token-AUPEC differences between terminal-matched trials. Pale circles are within-task pair means; diamonds are task-macro means. The three matching levels contain 101, 86, and 63 pairs, respectively.

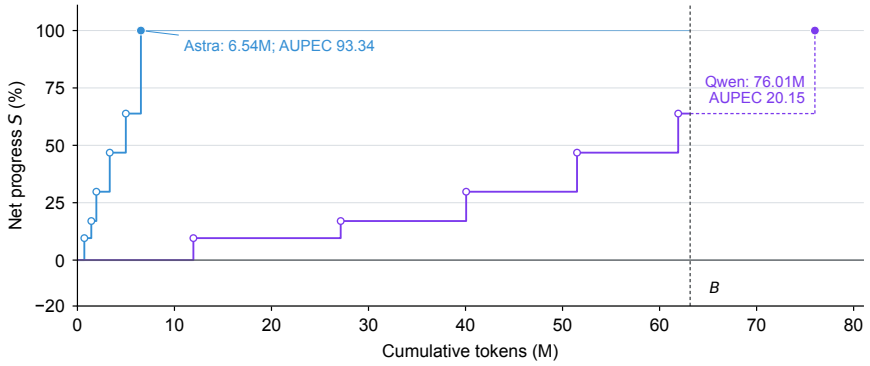


Figure 10: Matched javascript-02 trials with identical final verdicts and checkpoint scores. Astra's completed state is held to the shared budget $B = 63.15$M; Qwen's observed post-budget tail is dashed.

In Figure 10, Astra and Qwen pass the same 94 F2P and 12 P2P tests at the end. Astra finishes after 6.54M tokens; Qwen uses 76.01M. Their token AUPEC values at the shared budget are 93.34 and 20.15, respectively.

Comparable expenditure. Eight of the 63 exact-verdict pairs also have a ratio of larger to smaller total token expenditure at most 1.10, covering five tasks. Their within-task mean absolute AUPEC gaps average 3.55 points across tasks. For example, two trials on python-03 have identical final verdicts and use 45.81M and 44.91M tokens, a 2.01% difference, but differ by 9.44 AUPEC points. Integrating each trajectory to its own terminal expenditure and normalizing by that expenditure gives a 14.36-point difference, isolating different progress timing after rescaling total spend. This supplementary calculation describes path shape; the reported comparison still uses one common budget. Pairs share trials and are not independent observations.

C.2 Model Rankings and Efficiency

For the eight High-effort models, terminal C_F and S have identical rankings (Kendall's $\tau_b = 1.00$). Comparing either terminal score with token AUPEC gives $\tau_b = 0.00$: 14 of 28 model pairs reverse strict order, with no ties. The comparison uses each model's coverage specified in Section 4.1. Figure 7(d–f) places task-macro progress $S(B)$ against expenditure through each axis's pooled P90 budget, with color indicating AUPEC computed from checkpoint histories. These endpoint coordinates do not uniquely determine AUPEC. On the agent-steps axis, Astra has 81.62 AUPEC points versus 42.66 for DS Flash V4.1; one DS Flash V4.1 trial without step counts is excluded.

Endpoint association and budget sensitivity. The eight model means show a strong association between AUPEC, $S(B)$, and log expenditure through B (token-axis $R^2 = 0.997$). This is a descriptive fit to integrated scores, not a way to define or uniquely recover a trajectory's area from its endpoint. The trial matches above show the remaining path variation. Recomputing the common token budget at P80, P85, P90, and P95 preserves all eight models' AUPEC order. At P75, GLM and Hy4 exchange positions; at the maximum observed expenditure, DS Flash V4.1 moves from seventh to fourth. Larger budgets put more weight on completed terminal states. P90 bounds this long-tail influence while retaining most observed expenditure.

D Task-Level Calibration

Figure 11 summarizes a 16-task selection made on 6 September 2026 from 23 candidate tasks. Selection and scoring use the same calibration run, which evaluates Luna and DS Flash V4-0731 at High effort under the clean profile with Pi 0.84.2.

Agent-time AUPEC uses $B = 6540.293$ seconds (109.005 minutes), the pooled P90 terminal agent time of valid trials in the full 23-task calibration run. This calibration has its own task cohort and budget, separate from the model comparison. Invalid and right-censored trials are excluded. These repeated runs informed the v1 selection and describe its task-level behavior.

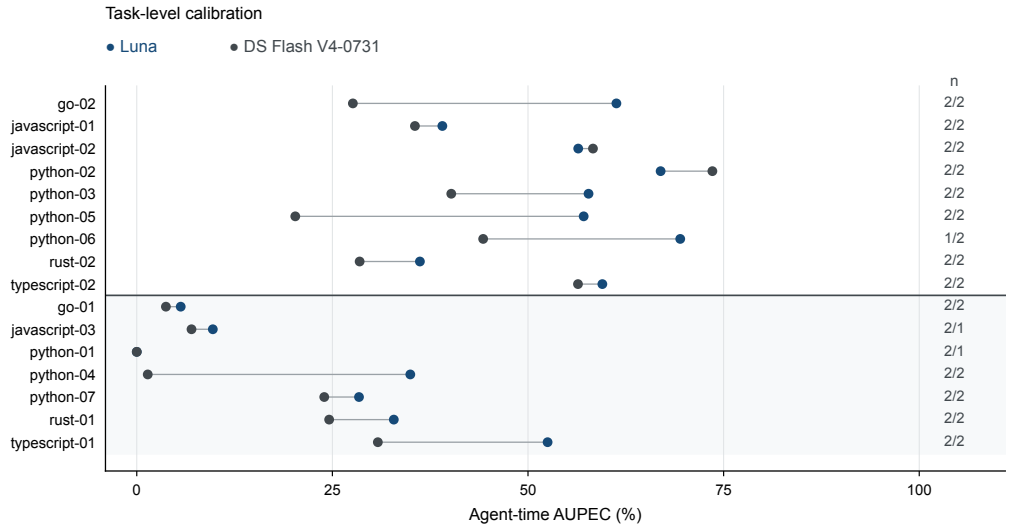


Figure 11: Agent-time AUPEC$_S$ for the selected task collection. Paired points are task means for Luna and DS Flash V4-0731; n lists valid-trial counts in legend order. Together they contribute 61 of 64 planned trials under a common 109.005-minute budget.

E Model Results and Cost Accounting

Table 5 gives the numerical results underlying Figure 5, including B_P. Cost AUPEC in panel (j) uses the separate public-price calculation below. These results cover the evaluated Pi/High/resume configuration; no harness or effort sweep is reported.

E.1 Historical Cost and Public-Price Revaluation

Historical costs in Figure 5(d) and Table 5 combine adapter-reported charges for Astra, Luna, and DS Flash V4-0731 with estimates for the other models. GLM, Qwen, and Hy4 estimates convert saved CNY rates at $1\,\text{USD} = 6.7080\,\text{CNY}$. Cost AUPEC revalues every checkpoint using the public USD rates in Table 6. For inclusive input I, cache-hit input H, and output O, the estimate is

$$\widehat{\text{Cost}} = \frac{(I-H)P_{\text{miss}} + HP_{\text{hit}} + OP_{\text{out}}}{10^6}, \tag{13}$$

Table 5: Model comparison (Pi harness, High effort, clean profile). Tk: tokens in millions; wall: minutes; cost: historical USD; Comp.: context compactions. Progress metrics are percentages; token AUPEC uses $B = 63.15$M. Means use the aggregation and coverage in Section 4.1. Blue and underlining mark the best and second-best displayed values in each column.

Model	Expenditures					Progress Metrics ($\times 100$)				
	↓Steps	↓Tk	↓Wall	↓Cost	↓Comp.	↑C_F	↑C_R	↑B_P	↑S	↑AUPEC$_{\text{tok}}$
Astra	**138.3**	**10.60**	67.2	18.47	**1.33**	**92.74**	**76.65**	99.96	**92.70**	**84.77**
Luna	231.2	18.39	**56.4**	0.61	1.89	89.24	60.48	99.80	89.05	73.11
DS Flash V4-0731	449.7	33.90	85.9	1.24	5.94	79.82	52.58	99.96	79.78	52.28
DS Flash V4.1	609.2	49.26	62.6	1.14	7.94	92.67	72.59	**99.98**	92.65	50.30
MiMo	389.9	27.87	110.9	**0.17**	2.36	76.28	36.05	99.90	76.18	54.62
GLM	259.8	17.17	110.4	0.71	2.28	74.06	50.20	98.85	72.91	59.31
Qwen	862.5	72.02	281.8	1.92	10.07	91.49	66.44	99.96	91.44	29.15
Hy4	354.6	26.35	138.8	2.72	4.17	90.97	67.96	99.94	90.91	65.29

where the three P terms are prices per million tokens. Cached input is charged once. Costs are computed per trial, averaged within tasks, and then averaged across eligible tasks.

Table 6: Public prices used for cost-AUPEC revaluation, in USD per million tokens (September 24, 2026 snapshot).

Model	Uncached input	Cached input	Output
Astra	10.00	1.00	50.00
Luna	0.20	0.02	1.20
DS Flash V4-0731	0.15	0.003	0.60
DS Flash V4.1	0.15	0.003	0.60
MiMo	0.14	0.0028	0.28
GLM	0.15	0.03	0.50
Qwen	0.15	0.016	0.47
Hy4	0.834	0.042	2.501

Model links identify public revaluation sources. Historical Qwen rates come from the Alibaba Cloud Singapore catalog; historical GLM and Hy4 rates come from Tencent TokenHub. The Federal Reserve H.10 series supplies the saved exchange-rate reference.

Revaluation uses the published off-peak tier for both DeepSeek IDs. Applying one price tier to a legacy ID does not establish its historical charges or model weights. Qwen, Hy4, MiMo, and GLM used an intermediary route; GLM's route suffix has no separately listed vendor rate. Available counters do not resolve every request-level surcharge, including cache writes, long-context multipliers, and tools. The resulting costs are conditional estimates.

The pooled P90 revalued terminal cost is \$3.3229536696 across 123 valid complete trials. Figures 5(j) and 7(c,f) integrate the revalued checkpoint trajectories through this budget. Historical costs are unchanged by this calculation. Cache-hit fractions use H/I, with inclusive input in the denominator and output excluded; token shares do not measure the fraction of requests that hit the cache.

F Repository Selection and Verification Pipeline

F.1 Source Selection and Task Structure

Repository-based construction starts from a pinned source commit, a usable license, reproducible dependencies, and executable tests. Source selection and qualification are separate: repository size or coverage does not establish instruction–test alignment. The released collection has six to eight

requirement-bearing stages per task; dependency depth ranges from two to seven edges. Revision links record changes to earlier code while preserving prior requirements.

The per-task counts in Table 7 are medians [min, max]; dependency depth counts edges on the longest path in G_{dep}.

Table 7: Corpus statistics.

Statistic	Value
Source tasks	16
Requirements	111
Requirement-bearing stages	110
Repositories	16
Languages	5
Requirements / task	7 [6, 8]
F2P tests / task	80.5 [19, 132]
P2P tests / task	14 [10, 643]
Dependency depth	4 [2, 7]

F.2 Source-Specific Requirement Construction

DeepSWE conversion groups tests by API behavior, state changes, error handling, and compatibility. Each group becomes a requirement with fixed test ownership. The source instruction, test changes, and reference patch support its scope and declared prerequisites.

Coverage and restoration order. Reconstruction uses per-test coverage from the complete implementation. Language adapters locate function and method ranges (using the abstract syntax tree for Python) and map executed lines to source units. For test i,

$$\mathcal{C}(i) = \{u \in \mathcal{U} : L(u) \cap L_i \neq \varnothing\}, \tag{14}$$

where $\mathcal{U}$ contains source units, $L(u)$ is a unit's line range, and L_i the lines executed by i. This records dynamic execution coverage. Tests with identical $\mathcal{C}(i)$ form a group.

Strict coverage-set inclusion defines a DAG: $\mathcal{C}_a \to \mathcal{C}_b$ when $\mathcal{C}_a \subsetneq \mathcal{C}_b$ and no group lies between them. Groups are ordered by set size, with stable test IDs breaking ties. A group creates a restoration step if it adds new units; otherwise it joins the earliest step covering its units. This produces a cumulative restoration chain. Inclusion describes covered-unit overlap; the chain fixes restoration order. Semantic prerequisite claims require separate evidence (Appendix F.3).

Removal and test ownership. Selected functions are stubbed, or the target feature is removed, to form W_0. Running tests on W_0 identifies F2P and P2P candidates; coverage alone does not determine this partition. Reference patches restore code incrementally. Each F2P belongs to the first restoration step at which it passes, and steps without new F2P are merged. Cumulative checks then verify preservation. This ownership is frozen before agent evaluation. Construction records retain selected units and all incremental patches. Unit selection and stubbing are language-specific. Build or collection failures require repair before test outcomes can establish ownership; they are not F2P evidence. A reproducible construction record must retain these choices.

Feature contracts. Feature contracts describe capabilities built from consecutive restoration steps. A verified pull request or release fixes source and target commits. Changed units, tests, coverage, and restoration steps support model-proposed descriptions. Every selected unit must belong to a capability, and the plan must cover the sequence. Unsupported observations remain outside the contracts as `unverified_observations`; commit history supplies context only.

F.3 Schedule Compilation and Edge Evidence

The schedule compiler applies Algorithm 1 or validates a supplied schedule against Equation 4. An E_{dep} pair may share a stage; an E_{rev} pair must span stages. The task scheduler follows the frozen

instructions and cumulative test scopes regardless of agent success. Message-only stages retain the preceding scope. The file `tests/test_partition.json` fixes F2P ownership.

Schedule compiler. Algorithm 1 returns an ordered list L of nonempty stages, with $E = E_{\text{dep}} \cup E_{\text{rev}}$. The saved policy π selects a nonempty subset of the ready requirements; `SortIDs` orders their stable IDs within a stage. The stage index of r is $s(r)$, and every edge in E spans stages as a stricter sufficient policy for Equation 4. It does not minimize the number of stages or enumerate all valid schedules.

Algorithm 1 Compile a requirement DAG into ordered stages

Require: Requirements $\mathcal{R}$; precedence edges E; deterministic policy π
Ensure: Stages L with $s(u) < s(v)$ for $(u, v) \in E$, or rejection

Reject duplicate IDs, invalid endpoints, self-loops, or cycles in $(\mathcal{R}, E)$
remaining $\leftarrow \mathcal{R}$; released $\leftarrow \varnothing$; $L \leftarrow [\,]$
while remaining $\neq \varnothing$ **do**
    ready $\leftarrow \{v \in \text{remaining} : \{u : (u, v) \in E\} \subseteq \text{released}\}$
    Require ready $\neq \varnothing$
    chosen $\leftarrow \pi.\text{Select}(\text{ready}, \text{released})$
    Require chosen $\neq \varnothing$, unique IDs, and chosen $\subseteq$ ready
    Append $\text{SortIDs}(\text{chosen})$ to L
    released $\leftarrow$ released $\cup$ chosen
    remaining $\leftarrow$ remaining $\setminus$ chosen
end while
Verify that L partitions $\mathcal{R}$
Derive $s(r)$ from L; require $s(u) < s(v)$ for $(u, v) \in E$
return L

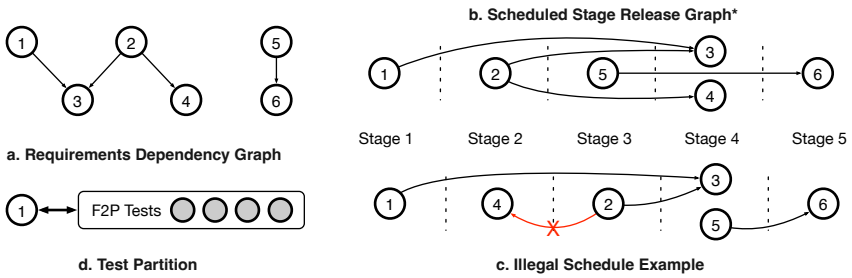


Figure 12: Compiling requirements into stages. (a) Upstream dependencies. (b) A valid schedule; a stage may contain several requirements. (c) An invalid schedule releases requirement 4 before prerequisite 2. (d) Fixed F2P test ownership.

Dependency records distinguish specification, code, and artifact dependencies and retain their validation status. Automatically generated chain edges start as declared prerequisites. For an edge check, downstream tests run on the reference layer sequence and again with the proposed prerequisite layer removed. A passing baseline followed by failure supports the edge; a passing ablation shows that the layer was unnecessary in that sequence. Patch conflicts and baseline failures are recorded separately. These checks distinguish declared prerequisites from dependencies supported by ablation.

Construction provenance links requirements to coverage groups, inclusion edges, stage groups, test ownership, reference patches, schedule versions, and file hashes. Review checks these records against behavior descriptions and release order. It also checks that messages reveal no hidden verifier content, reference patches, or future requirements.

F.4 Qualification and Audit

Reference checks establish initial failure and staged solvability. The no-op workspace must fail every F2P and pass every P2P. The full reference must pass the complete suite in two fresh environments; staged execution must pass every cumulative scope. Each incremental patch must turn its newly assigned F2P tests from failing to passing.

Review checks agreement among each requirement, its instruction, tests, and reference behavior. It also examines prerequisite evidence and substantive changes between stages. Automated checks cover test selection, workspace reconstruction, information visibility, repeatability, and result recording. Repairs create a new task version and repeat affected checks.

The v1 selection uses environment-build checks, no-op and oracle execution, dry runs, and repeated agent attempts. Human inspection covers the DAG, requirement order, and instruction–test–reference alignment, followed by a task audit before release.

G Reproducibility

Exact reruns require task and manifest versions, environment digests, runner and adapter versions, model IDs, effort, profile, context policy, and resource limits. Attempt counts, seeds, concurrency, retries, metric versions, token conventions, and pricing sources must accompany the results.

Reproduction checks the requirement–test mapping, reconstructs checkpoint states, and recomputes trial-level metrics before task-level aggregation. Saved exclusions and per-field coverage account for the difference between planned and reported trials. The manuscript bundle includes figure scripts and their frozen plotting inputs; `iclr27/README.md` documents local figure and PDF reproduction. A public execution release with pinned environments and a run command is not yet available.

Planned tools and specification release. We will open-source the interactive visual analysis suite and profiling tools, together with a detailed task-building specification. The analysis suite supports model and task comparisons, resource-axis selection, and inspection of individual trial trajectories, checkpoint states, and requirement- and test-level transitions. The profiling tools compute progress, continuity events, resource curves, and AUPEC from saved checkpoint records. The specification documents task artifacts, requirement–test ownership, instruction–reference alignment, DAG and release-schedule constraints, and validation procedures, so researchers can construct tasks and analyze runs under the same protocol.

H V1 Task Instructions and Requirement DAGs

Table 8 indexes the complete v1 collection. Entries preserve stage order and behavioral requirements. Language–number IDs label tasks; instructions use functional descriptions and neutral names in code examples. Diagrams show E_{dep}; labels s1, s2, … mark release stages. The separate semantic-interference batch uses its own task versions (Appendix A.3).

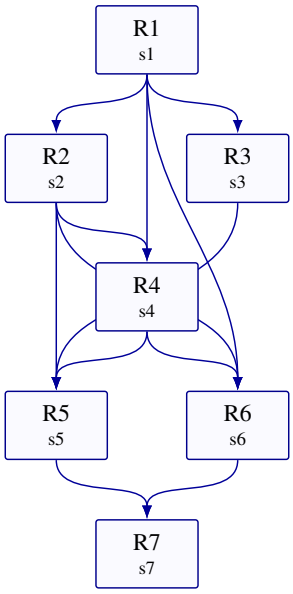
Table 8: The 16-task v1 evaluation collection. Counts come from each task's test partition. The task names link to full instructions and requirement DAGs below.

Task	Stages	F2P	P2P
go-02	7	91	15
javascript-01	6	79	12
javascript-02	6	94	12
python-02	6	69	282
python-03	7	38	14
python-05	8	118	14
python-06	7	50	229
rust-02	7	104	12
typescript-02	7	51	12
go-01	8	132	13
javascript-03	6	90	12
python-01	7	94	19
python-04	8	73	19
python-07	7	19	10
rust-01	7	40	26
typescript-01	6	82	643

H.1 Task instructions

H.1.1 go-02

7 stages, 7 requirements, 91 F2P tests, and 15 P2P tests.



Stage 1 instruction

```
A user-facing gap has been isolated in the language runtime and compiler.  The
surrounding APIs are already present, so please make this first behavior dependable
before we add more surface area.
Please start destructuring bindings for arrays.  Support [a, b] := value with
positional assignment, bind a missing position as undefined, and accept an empty
pattern.  Only the declaration form := is valid at this stage; a destructuring
assignment using = must continue to fail through the normal parser path.  Keep
ordinary scalar declarations and evaluation order unchanged.
```

Please keep earlier public names and error conventions. Exercise the ordinary case and the nearest invalid or empty case, then run the focused checks so a failure is visible rather than masked. Begin with the smallest fixture a caller would write, then confirm the same result when the feature is used through the public entry point.

Stage 2 instruction

The previous slice of the language runtime and compiler is usable in simple cases. We now need to handle the next scenario that appears in production, while keeping the earlier contract intact.

Please add map destructuring keyed by string fields. Support shorthand names, explicit renames, default expressions, and missing keys while preserving the map's existing lookup rules. A default should run only when the key is absent, not when it is present with an explicit undefined. Invalid pattern keys should produce the same syntax error style as other language patterns.

When you verify this, include one repeated or reordered invocation; the result should stay deterministic and caller-owned state should remain untouched unless mutation is explicitly part of the API. Use a repeated invocation and a reordered input as a regression case; both should observe the same stable contract.

Stage 3 instruction

We have a concrete report from a project using the language runtime and compiler: the edge case below is difficult to diagnose today. Please address it in the existing design instead of introducing a parallel API.

Please support ...name rest bindings in array patterns only. The rest binding must be final (and there may be at most one), collecting remaining values in order; map rest patterns remain rejected. Empty input, a rest-only pattern, and a rest after ordinary positional bindings should all have predictable values and scope.

Do not change unrelated modules or add a fallback that hides malformed input. A focused regression check should demonstrate both the useful result and the behavior at the boundary. Include one malformed, empty, or missing-value case so the established error or no-op behavior is explicit.

Stage 4 instruction

The core path in the language runtime and compiler now works, but callers still cross this boundary when they combine features. This step should make that composition predictable and observable.

Please make default expressions lazy and scope-aware. Evaluate a default only for an absent array position or map key, in left-to-right pattern order, with earlier bindings visible and later bindings unavailable. Preserve the distinction between absence and a present undefined, and do not evaluate defaults when the surrounding declaration fails.

Preserve supported runtime versions and existing output for callers that do not use this feature. Check the observable return value, error text, and ordering before handing it over. Compare this path with the equivalent operation from the preceding Step and record any intentional difference in the focused test.

Stage 5 instruction

We are preparing the language runtime and compiler for another release. Please close the behavior gap described below, including the failure path, without broad refactoring or silent compatibility changes.

Please allow arbitrary nesting of array and map patterns, including rest and defaults at any nested level. Nested failures should identify the real pattern location, while successful nested values retain their normal runtime types. Keep empty patterns and computed right-hand-side expressions compatible with the existing evaluator.

Please compare the new path with the equivalent old path where one exists. The implementation is ready only when the repository checks cover success, failure, and cleanup without relying on private test selectors. Exercise two features together, because the main risk here is a state or metadata leak across the boundary described above.

Stage 6 instruction

At this point the implementation needs a careful integration pass. In the language runtime and compiler, the following behavior is what a real caller sees when state changes mid-operation.

Please integrate pattern bindings with lexical scope, functions, conditionals, loops, closures, and computed right-hand sides. A binding must not leak outside its declaration scope, and a closure must observe the same captured value as an ordinary declaration. Preserve evaluation order and avoid leaving temporary rest values on the VM stack.

Keep the change local to the requested behavior and leave a reproducible check in the existing test entry point. If an operation is repeated, it must not accumulate stale state. Check cleanup and reuse after the operation completes or fails; the next caller should start from a clean state.

Stage 7 instruction

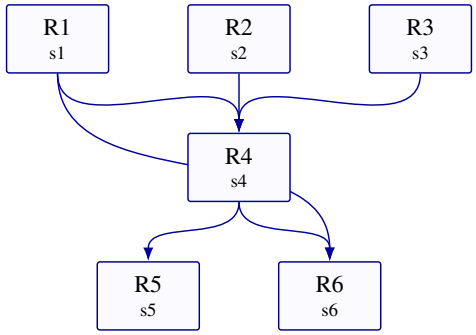
One last compatibility issue remains in the language runtime and compiler. Treat this as a request from an application maintainer: solve the concrete problem and leave unrelated behavior alone.

Please extend the same patterns to function parameters, including nested array/map patterns, one final rest parameter, and lazy defaults. Calls with too few or extra arguments must follow the existing argument-count contract, and parameter bindings must be available only inside the function body. Keep ordinary positional parameters source-compatible.

Before delivery, run the focused repository checks and one integration example. Report real failures instead of making the feature silently permissive. Keep the example small enough to run locally, but make its expected output precise enough for a reviewer to diagnose a regression.

H.1.2 JAVASCRIPT-01

6 stages, 6 requirements, 79 F2P tests, and 12 P2P tests.



Stage 1 instruction

A user-facing gap has been isolated in CSS shorthand processing. The surrounding APIs are already present, so please make this first behavior dependable before we add more surface area.

Please add the lexer APIs expandShorthand(propertyName, value) and compressShorthand(propertyName, longhands). Expansion should return an object of longhand name to value string, while compression should return the shortest equivalent shorthand string or null when the input is not representable. Keep property-name normalization and the existing fork syntax behavior intact. CSS-wide keywords must be treated consistently: a uniform keyword can compress, but a mixed keyword/value set must not silently change meaning. Preserve the lexer's existing null/error conventions so callers can use these methods during validation as well as serialization.

Please keep earlier public names and error conventions. Exercise the ordinary case and the nearest invalid or empty case, then run the focused checks so a failure is visible rather than masked. Begin with the smallest fixture a caller would write, then confirm the same result when the feature is used through the public entry point.

Stage 2 instruction

The previous slice of CSS shorthand processing is usable in simple cases. We now need to handle the next scenario that appears in production, while keeping the earlier contract intact.

Please implement shortest box-model compression for margin, padding, inset, and border-radius. Apply the one-, two-, three-, and four-value rules independently to each shorthand and preserve slash-separated horizontal/vertical radii for border-radius. Do not remove meaningful units, CSS variables, or calc() expressions merely to save characters. A value such as 8px 8px 8px 8px should become 8px, while unequal sides must retain the minimum form that reproduces the same longhands. Return null for incomplete or contradictory longhand sets.

When you verify this, include one repeated or reordered invocation; the result should stay deterministic and caller-owned state should remain untouched unless mutation is explicitly part of the API. Use a repeated invocation and a reordered input as a regression case; both should observe the same stable contract.

Stage 3 instruction

We have a concrete report from a project using CSS shorthand processing: the edge case below is difficult to diagnose today. Please address it in the existing design instead of introducing a parallel API.

Please extend compression to component shorthands, including background, border, font, outline, flex, flex-flow, and list-style. Each component must be emitted in the grammar order accepted by the lexer, and omitted defaults should be removed only when the resulting shorthand still expands to the supplied longhands. Keep layered background values aligned by layer and never merge layers with different lengths. If a component contains a custom property, variable, or syntax the lexer cannot prove equivalent, leave it uncompressed instead of guessing.

Do not change unrelated modules or add a fallback that hides malformed input. A focused regression check should demonstrate both the useful result and the behavior at the boundary. Include one malformed, empty, or missing-value case so the established error or no-op behavior is explicit.

Stage 4 instruction

The core path in CSS shorthand processing now works, but callers still cross this boundary when they combine features. This step should make that composition predictable and observable.

Please complete direct longhand expansion for the supported shorthands. Apply CSS defaults, two-value propagation, layered background repetition, and the syntax validation rules already used by the lexer. Expansion must reject malformed token sequences and must not invent a value for a required component. Keep the returned object deterministic in property order, because downstream serialization compares it byte-for-byte. Check both ordinary declarations and fork-defined custom syntax, including a shorthand that contains an explicit initial, inherit, or unset keyword.

Preserve supported runtime versions and existing output for callers that do not use this feature. Check the observable return value, error text, and ordering before handing it over. Compare this path with the equivalent operation from the preceding Step and record any intentional difference in the focused test.

Stage 5 instruction

We are preparing CSS shorthand processing for another release. Please close the behavior gap described below, including the failure path, without broad refactoring or silent compatibility changes.

Please make expansion and compression round-trip safely for the common shorthands and for fork custom syntax. Compressing an expanded value and expanding it again should produce equivalent longhands, even when the shortest spelling uses propagated values or a slash separator. Preserve token spelling where the grammar permits variables and functions, and do not normalize a value in a way that changes case-sensitive identifiers. Include nested and layered examples so a caller can rely on the APIs for a formatter rather than only for isolated unit values.

Please compare the new path with the equivalent old path where one exists. The implementation is ready only when the repository checks cover success, failure, and cleanup without relying on private test selectors. Exercise two features together, because the main risk here is a state or metadata leak across the boundary described above.

Stage 6 instruction

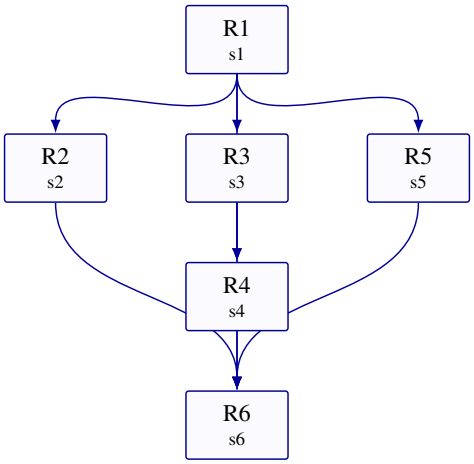
At this point the implementation needs a careful integration pass. In CSS shorthand processing, the following behavior is what a real caller sees when state changes mid-operation.

Please finish the null-result and validation contract. Unknown properties, missing longhands, incomplete component sets, CSS-wide keyword mixtures, and out-of-range values should return null (or the existing lexer error) consistently instead of producing a partial shorthand. A uniform keyword remains compressible, but mixing inherit with an ordinary value must be rejected. Keep valid zero, negative, variable, and function values working, and ensure expansion never mutates the caller's object.

Keep the change local to the requested behavior and leave a reproducible check in the existing test entry point. If an operation is repeated, it must not accumulate stale state. Check cleanup and reuse after the operation completes or fails; the next caller should start from a clean state.

H.1.3 JAVASCRIPT-02

6 stages, 6 requirements, 94 F2P tests, and 12 P2P tests.



Stage 1 instruction

A user-facing gap has been isolated in math array rendering. The surrounding APIs are already present, so please make this first behavior dependable before we add more surface area.

Please add \multicolumn{n}{alignment}{content} to math array parsing. The alignment argument must contain exactly one of l, c, or r, with optional | markers for vertical rules; n must be a positive integer that fits within the remaining columns. The cell should occupy the requested columns while preserving row separators and the surrounding environment's delimiters. Keep ordinary \, &, styling nodes, and existing array errors unchanged. A normal two-column table should render exactly as before when no multicolumn command is present.

Please keep earlier public names and error conventions. Exercise the ordinary case and the nearest invalid or empty case, then run the focused checks so a failure is visible rather than masked. Begin with the smallest fixture a caller would write, then confirm the same result when the feature is used through the public entry point.

Stage 2 instruction

The previous slice of math array rendering is usable in simple cases. We now need to handle the next scenario that appears in production, while keeping the earlier contract intact.

Please carry the span through the MathML tree. A multicolumn cell should expose the correct columnspan and columnalign attributes, and vertical rules should be represented at the row boundaries rather than duplicated inside the spanned cell. Keep HTML and MathML column counts consistent for mixed ordinary and spanning rows. Verify left, center, and right alignment plus optional leading and trailing | markers, including a span that reaches the final column.

When you verify this, include one repeated or reordered invocation; the result should stay deterministic and caller-owned state should remain untouched unless mutation is explicitly part of the API. Use a repeated invocation and a reordered input as a regression case; both should observe the same stable contract.

Stage 3 instruction

We have a concrete report from a project using math array rendering: the edge case below is difficult to diagnose today. Please address it in the existing design instead of introducing a parallel API.

Please make the same multicolumn behavior available in matrix, pmatrix, bmatrix, Bmatrix, vmatrix, Vmatrix, cases, rcases, aligned, and smallmatrix. Reuse the common array machinery so delimiters and environment-specific spacing remain unchanged. A command nested in a cell should affect only that cell's row and should not leak alignment or rule state into the parent environment.

Do not change unrelated modules or add a fallback that hides malformed input. A focused regression check should demonstrate both the useful result and the behavior at the boundary. Include one malformed, empty, or missing-value case so the established error or no-op behavior is explicit.

Stage 4 instruction

The core path in math array rendering now works, but callers still cross this boundary when they combine features. This step should make that composition predictable and observable.

Please exercise the boundary cases around spans: n=1, a span in the first or last cell, adjacent spans, empty content, nested arrays, style nodes, and complex formulas. These cases must preserve the same spacing and delimiter behavior as an equivalent unspanned table. Do not special-case only text content; fractions, operators, accents, and scripts inside the cell must render normally.

Preserve supported runtime versions and existing output for callers that do not use this feature. Check the observable return value, error text, and ordering before handing it over. Compare this path with the equivalent operation from the preceding Step and record any intentional difference in the focused test.

Stage 5 instruction

We are preparing math array rendering for another release. Please close the behavior gap described below, including the failure path, without broad refactoring or silent compatibility changes.

Please make invalid multicolumn syntax fail with the renderer's normal ParseError. Reject zero or negative spans, spans that exceed the remaining columns, alignment strings containing multiple alignment letters, use outside an array-like environment, and illegal nesting positions. Keep the error location and message style consistent with nearby array parse errors so editor integrations can report the problem accurately.

Please compare the new path with the equivalent old path where one exists. The implementation is ready only when the repository checks cover success, failure, and cleanup without relying on private test selectors. Exercise two features together, because the main risk here is a state or metadata leak across the boundary described above.

Stage 6 instruction

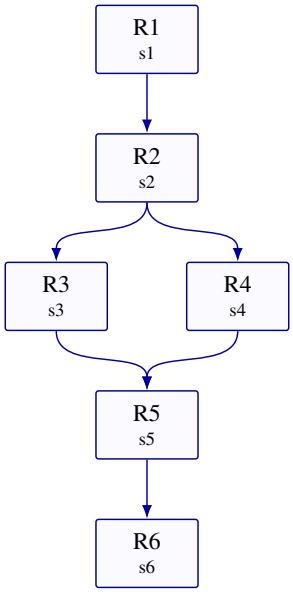
At this point the implementation needs a careful integration pass. In math array rendering, the following behavior is what a real caller sees when state changes mid-operation.

Please finish the compatibility details: horizontal rules, alignment overrides, row structure, delimiter suppression, CD environments, multi-digit spans, and precise MathML attributes. A row rule must still start and end at the logical table boundary even when the adjacent cell spans columns. Existing arrays that do not use multicolumn must produce byte-for-byte compatible output.

Keep the change local to the requested behavior and leave a reproducible check in the existing test entry point. If an operation is repeated, it must not accumulate stale state. Check cleanup and reuse after the operation completes or fails; the next caller should start from a clean state.

H.1.4 PYTHON-02

6 stages, 6 requirements, 69 F2P tests, and 282 P2P tests.



Stage 1 instruction

A user-facing gap has been isolated in linter suppression directives. The surrounding APIs are already present, so please make this first behavior dependable before we add more surface area.

Please extend the current implementation with this next requirement.

The linter currently supports inline # nosec, but suppressing an entire span of code or only the next statement requires repeating inline markers. Extend the linter with structured region and next-statement suppression while preserving existing inline behavior.

For the first increment, support an unqualified # nosec-begin / # nosec-end region. The begin directive applies from the following physical line, does not suppress findings on the directive line itself, and blanket-suppresses findings inside the region until its end.

Please keep earlier public names and error conventions. Exercise the ordinary case and the nearest invalid or empty case, then run the focused checks so a failure is visible rather than masked. Begin with the smallest fixture a caller would write, then confirm the same result when the feature is used through the public entry point.

Stage 2 instruction

The previous slice of linter suppression directives is usable in simple cases. We now need to handle the next scenario that appears in production, while keeping the earlier contract intact.

Please extend the current implementation with this next requirement.

Recognize nosec-begin, nosec-end, and nosec-next-line case-insensitively. Accept the supported whitespace variants, directives placed after code on a line, and selector text followed by a trailing comment. nosec-end must not behave as an ordinary inline nosec. When the linter runs with ignore-nosec, ignore every structured nosec directive.

When you verify this, include one repeated or reordered invocation; the result should stay deterministic and caller-owned state should remain untouched unless mutation is explicitly part of the API. Use a repeated invocation and a reordered input as a regression case; both should observe the same stable contract.

Stage 3 instruction

We have a concrete report from a project using linter suppression directives: the edge case below is difficult to diagnose today. Please address it in the existing design instead of introducing a parallel API.

Please extend the current implementation with this next requirement.

Add optional selectors immediately after nosec-begin and nosec-next-line. A missing or empty selector and all mean blanket suppression; none has no effect. Support test IDs, test names, ID-prefix globs, implicit union by whitespace or commas, |, &, -, !, and parentheses. If expression parsing fails, fall back to a plain whitespace/comma-separated union.

Do not change unrelated modules or add a fallback that hides malformed input. A focused regression check should demonstrate both the useful result and the behavior at the boundary. Include one malformed, empty, or missing-value case so the established error or no-op behavior is explicit.

Stage 4 instruction

The core path in linter suppression directives now works, but callers still cross this boundary when they combine features. This step should make that composition predictable and observable.

Please extend the current implementation with this next requirement.

Implement full region and next-statement targeting. Regions begin on the following physical line, may nest, close in LIFO order, and an indented unterminated region auto-closes at a later line with smaller leading indentation; otherwise it runs to EOF. An unmatched end is a no-op, and an end closes its region before the line containing the directive. Extra text after nosec-end is ignored.

For nosec-next-line, target the next statement after skipping blank lines, comment-only lines, grouping-only lines, semicolons, and ellipsis-only lines. Preserve these rules across file boundaries and Windows newlines.

Preserve supported runtime versions and existing output for callers that do not use this feature. Check the observable return value, error text, and ordering before handing it over. Compare this path with the equivalent operation from the preceding Step and record any intentional difference in the focused test.

Stage 5 instruction

We are preparing linter suppression directives for another release. Please close the behavior gap described below, including the failure path, without broad refactoring or silent compatibility changes.

Please extend the current implementation with this next requirement.

This requirement revises R2 and R4: their inline and region suppression rules now compose at statement scope as described below, while their selector matching and nesting guarantees remain unchanged.

Make suppression statement-wide: if any physical line of a multi-line statement is suppressed, suppress findings for the whole statement, even when nosec-end appears later within that statement. Combine all applicable inline, region, and next-statement suppressions; union specific selector sets and let any blanket suppression dominate.

Please compare the new path with the equivalent old path where one exists. The implementation is ready only when the repository checks cover success, failure, and cleanup without relying on private test selectors. Exercise two features together, because the main risk here is a state or metadata leak across the boundary described above.

Stage 6 instruction

At this point the implementation needs a careful integration pass. In linter suppression directives, the following behavior is what a real caller sees when state changes mid-operation.

Please extend the current implementation with this next requirement.

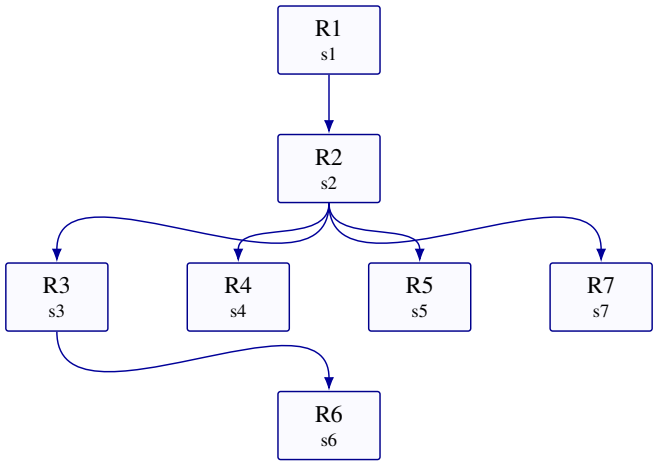
Classify metrics from the resolved suppression set. Blanket suppression increments nosec; a non-empty specific set increments skipped_tests. Combined selectors that resolve to the full enabled test set are blanket, while expressions such as all & B602 remain specific when they resolve only to that test.

Keep the change local to the requested behavior and leave a reproducible check in the existing test entry point. If an operation is repeated, it must not accumulate stale

state. Check cleanup and reuse after the operation completes or fails; the next caller should start from a clean state.

H.1.5 PYTHON-03

7 stages, 7 requirements, 38 F2P tests, and 14 P2P tests.



Stage 1 instruction

A user-facing gap has been isolated in the command-line framework. The surrounding APIs are already present, so please make this first behavior dependable before we add more surface area.

CLI authors need defaults and flags to behave the same whether values come from declared defaults, a context default map, environment-like text, or explicit command-line input. Restore this behavior while keeping callbacks able to distinguish an omitted option from an explicitly supplied flag.

The command decorator should remain convenient when used directly without parentheses. Option defaults must be converted through their declared types, including boolean values, and composite defaults must preserve the expected tuple or multiple-value shape while converting each element. Text supplied through a default map for multi-value options should be split appropriately; already structured sequences should pass through, and command-line values must take precedence. Default-map entries should also correctly override automatically aligned flag values, invoke callable entries, and allow an explicit null value; an unset sentinel should fall back to the option's own default.

Eager callbacks must be able to query the parameter source while they run. Defaults and default-map values should not be mistaken for explicit input when callbacks use that source to decide whether to update external state. For example, @cli_api.command\ndef cli(): cli_api.echo("hello") should be directly invokable and print hello followed by a newline.

Please keep earlier public names and error conventions. Exercise the ordinary case and the nearest invalid or empty case, then run the focused checks so a failure is visible rather than masked. Begin with the smallest fixture a caller would write, then confirm the same result when the feature is used through the public entry point.

Stage 2 instruction

The previous slice of the command-line framework is usable in simple cases. We now need to handle the next scenario that appears in production, while keeping the earlier contract intact.

Building on the existing command and option facilities, make command registration and default handling dependable for applications that customize groups or obtain values from configuration. A group's command decorator should work both as a direct decorator and with a custom command class, while still producing an invokable command with the expected output.

Option defaults should be converted through the declared type, including booleans and numeric values. Defaults supplied through a default map must also work for multi-value options: split a textual value into the expected number of values,

preserve already structured sequences, and let explicit command-line arguments take precedence. When several flag-value options share one parameter, the option marked as the default should determine the value when no flag is supplied, regardless of decorator ordering; an explicitly supplied flag must still select its own value.

For example, @grp.command\ndef cli():\n cli_api.echo("hello custom command class") should register and invoke cli successfully when grp uses a custom command class. Keep invalid or incomplete multi-value input subject to normal parameter validation rather than silently accepting it.

When you verify this, include one repeated or reordered invocation; the result should stay deterministic and caller-owned state should remain untouched unless mutation is explicitly part of the API. Use a repeated invocation and a reordered input as a regression case; both should observe the same stable contract.

Stage 3 instruction

We have a concrete report from a project using the command-line framework: the edge case below is difficult to diagnose today. Please address it in the existing design instead of introducing a parallel API.

Users need commands to treat mapped defaults as a first-class source of parameter values, without losing type conversion or provenance information along the way. Building on the existing option-default behavior, make default_map resolution consistent for multi-value options, flags, custom parameter types, and eager callbacks.

A mapped value should take precedence over the declared default even when it is None or otherwise falsy, while an explicit command-line value must still win. Strings supplied for options accepting multiple values should be split and converted just as environment-variable values are, whereas already structured sequences should remain usable. Callable values from a map should be evaluated when resolving the command, but help or other non-evaluating inspection must not accidentally invoke them. Flag values must preserve their normal command-line activation semantics while allowing a mapped value to override the automatically aligned flag value.

For example, with an option accepting two integers, default_map={"point": "3 4"} should produce (3, 4) when no CLI value is given. Context parameter-source queries should report DEFAULT_MAP, DEFAULT, or COMMANDLINE accurately, including while a custom type converts a value and while an eager callback runs; values populated through unrelated callback-side changes should not acquire a misleading source.

Do not change unrelated modules or add a fallback that hides malformed input. A focused regression check should demonstrate both the useful result and the behavior at the boundary. Include one malformed, empty, or missing-value case so the established error or no-op behavior is explicit.

Stage 4 instruction

The core path in the command-line framework now works, but callers still cross this boundary when they combine features. This step should make that composition predictable and observable.

Users need to combine positional arguments supplied when a command is created with positional arguments attached by decorators, without losing declarations or changing parsing order. Building on the existing command decoration and default handling, make mixed argument registration behave as a single coherent declaration list.

When cli_api.command receives an existing positional parameter through its params setting and cli_api.argument adds another parameter to the callback, the resulting command must retain both. Explicitly supplied parameters should remain first, while decorator-added arguments follow in declaration order. Invocation must assign each command-line value to the corresponding callback parameter, so the callback sees the same order exposed by the command's public parameter list.

For example, a command with explicit positional \a" plus decorated positional \b": cli 1 2 -> output "1 2".

Preserve normal positional-argument validation: if a required value is missing, report the usual missing-argument usage error rather than silently dropping a declaration, rebinding a value, or failing during command construction. This behavior should remain compatible with commands that use only one declaration style.

Preserve supported runtime versions and existing output for callers that do not use this feature. Check the observable return value, error text, and ordering before handing it over. Compare this path with the equivalent operation from the preceding Step and record any intentional difference in the focused test.

Stage 5 instruction

We are preparing the command-line framework for another release. Please close the behavior gap described below, including the failure path, without broad refactoring or silent compatibility changes.

CLI authors should be able to define command hierarchies with the concise decorator form, without adding empty parentheses at every level. Building on the existing command decorator behavior, group creation and group-bound command or subgroup registration should accept a callback directly and produce the same usable hierarchy as their called forms.

For example, `@cli_api.group def cli(): cli_api.echo("root")

@cli.group def admin(): cli_api.echo("admin")

@admin.command def sync(): cli_api.echo("sync") should allow invoking cli admin sync and emit root, admin, then sync`, each on its own line.

This shorthand must work consistently for both direct child commands and nested groups. Invocation should resolve the requested path without decorator-related errors, execute every selected group callback in traversal order, and finally execute the leaf command exactly once.

Please compare the new path with the equivalent old path where one exists. The implementation is ready only when the repository checks cover success, failure, and cleanup without relying on private test selectors. Exercise two features together, because the main risk here is a state or metadata leak across the boundary described above.

Stage 6 instruction

At this point the implementation needs a careful integration pass. In the command-line framework, the following behavior is what a real caller sees when state changes mid-operation.

Users need to configure commands through a default map without having to encode multi-value options in a special Python-only shape. Building on the existing default and environment-variable resolution behavior, make fixed-arity options interpret string values from default_map as separate values using the same splitting rules as environment-variable input, then apply the option's declared conversion and return the normal tuple-shaped result.

Already structured tuple or list defaults should continue through the usual multi-value conversion without being split or flattened. Explicit command-line values must take precedence over the mapped value, and tuple-typed options should honor the arity implied by that type. Invalid value counts should still produce the normal option validation error rather than silently accepting a partial value.

For example, an option declared with nargs=2 and integer conversion should resolve default_map={"point": "3 4"} to the command callback as (3, 4) when no CLI value is supplied.

Keep the change local to the requested behavior and leave a reproducible check in the existing test entry point. If an operation is repeated, it must not accumulate stale state. Check cleanup and reuse after the operation completes or fails; the next caller should start from a clean state.

Stage 7 instruction

One last compatibility issue remains in the command-line framework. Treat this as a request from an application maintainer: solve the concrete problem and leave unrelated behavior alone.

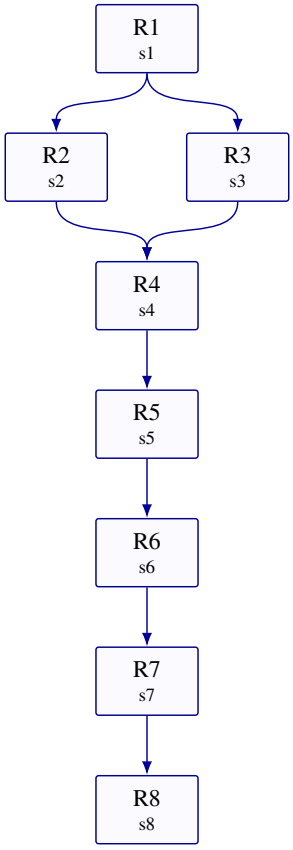
Users need command defaults to behave consistently whether a value comes from ordinary option configuration, a supplied default map, or a context that customizes default lookup. Building on the existing command and option behavior, make default resolution honor mapped boolean values, invoke lazy defaults only when resolving an actual command value, and preserve the callable as dynamic when rendering help. Customized lookup behavior must be respected by both execution and help generation rather than being bypassed, so a context can provide fallback values while still producing accurate displayed defaults. Explicit command-line values should continue to take precedence, and help should remain usable even when a default is callable or comes from a nested map. For example, invoking a command with default_map={"name": lambda: "lazy-value"} should use lazy-value at runtime while its help text identifies the default as (dynamic).

Before delivery, run the focused repository checks and one integration example. Report real failures instead of making the feature silently permissive. Keep the

example small enough to run locally, but make its expected output precise enough for a reviewer to diagnose a regression.

H.1.6 PYTHON-05

8 stages, 8 requirements, 118 F2P tests, and 14 P2P tests.



Stage 1 instruction

Applications need a reliable place to keep values scoped to the active application context, and the development server must respect the application's exception policy when debug features are enabled. Restore these runtime behaviors without changing the surrounding web framework programming model.

Within an active application context, web_api.g should work as an attribute-backed namespace while also supporting useful collection behavior: missing values can be read with a default, membership reflects stored attributes, and iteration exposes the names that have been stored. Values must remain tied to the current context, with normal framework behavior when no application context is available. For example, inside with app.app_context(): web_api.g.request_id = "abc"; assert web_api.g.get("request_id") == "abc"; assert "request_id" in web_api.g.

Building on the existing application runner, app.run should continue forwarding the requested host and port to the Werkzeug development server. It must also pass through the effective exception-propagation choice: an explicit PROPAGATE_EXCEPTIONS setting takes precedence, while an unset setting follows the selected debug behavior. This decision should remain correct regardless of whether the debugger or reloader is enabled, so propagated errors are not unexpectedly swallowed.

Stage 2 instruction

Building on the existing application-context behavior, restore the request-facing session and JSON capabilities users rely on when handling web requests. JSON encoding and decoding should support ordinary mappings as well as dataclass values, preserving their fields in the decoded result. Sessions should remain available for

reads inside a request context, while changing or removing session data must fail clearly when the application has no secret key and therefore cannot provide a signed session; normal configured applications should continue to use the session mechanism through the request lifecycle. For example, encoding a dataclass containing name= "Example" and decoding it again should produce {"name": "Example"}. Keep the behavior compatible with the existing application and request interfaces, including meaningful failure behavior rather than silently discarding attempted session changes.

Stage 3 instruction

Applications need a predictable way to turn common return values into HTTP responses while preserving application JSON settings and session-backed notifications. Building on the existing request/session and JSON handling, restore the response and flash behavior exposed to callers.

The response-building interface should accept an empty body, text, byte or text iterables, and existing response objects, with sensible defaults for status and content type. Supplied status codes and headers must be applied without discarding headers or changing an existing response's body or media type. JSON-produced responses should retain the application's configured JSON media type, while attempts to provide both positional JSON data and keyword fields should fail clearly. Flashing a message should mark the session as modified when appropriate, and retrieving flashed messages should preserve insertion order.

For example, web_api.make_response("W00t", 404) should produce a 404 response with body W00t and a text/html media type.

Stage 4 instruction

Web applications need request dispatch to preserve serious endpoint failures and reject invalid view results clearly, rather than masking them or failing later in the WSGI stack. Building on the existing application, request, and response behavior, ensure the test client invokes endpoints consistently: BaseException subclasses raised by a view, including interruptions, must escape the request even when ordinary exception propagation is disabled, while supported return values continue through normal response construction. View results that are missing, have an invalid tuple shape, use an unsupported scalar type, or are not a valid WSGI response must raise TypeError with a useful explanation that identifies the problem and, where applicable, the endpoint. For example, when an endpoint falls through without returning a value, client.get("/") should raise TypeError mentioning that the view returned None. Keep this behavior consistent for requests dispatched through the application's normal client interface.

Stage 5 instruction

Applications need reliable URL generation and static assets when deployed across ordinary paths, subdomains, or dedicated asset hosts. Restore routing behavior so host matching and subdomain matching honor the configured server name, including ports, while ordinary routes continue to resolve on the default host. Static files should be served from path-like folders, remain usable with empty folder or URL-path values, normalize trailing slashes in the public static URL, and generate correct relative or external URLs when a static host is configured. Invalid combinations of host matching, static hosting, and disabled static files should fail clearly during application setup.

URL generation must also handle unresolved endpoints predictably: without a successful fallback it raises the normal build error, while registered build-error handlers are tried and may return a replacement URL. If a handler re-raises the build error, or if the error is being handled outside its original exception context, the build error must remain visible rather than being silently converted.

For example, app = WebApp(__name__, host_matching=True, static_host="example.com"); client.get("http://example.com/static/index.html").status_code == 200.

Keep route variable values safely encoded when building URLs and preserve the established request and response behavior.

Stage 6 instruction

Applications need a dependable request lifecycle: they should be able to modify a response after the view runs, short-circuit dispatch from a before-request callback, and perform teardown work without changing the response sent to the client. Building on the existing routing and dispatch behavior, complete HTTP method negotiation as

well, including implicit HEAD and automatic OPTIONS responses for routes that do not implement OPTIONS themselves.

Automatic OPTIONS should advertise the route's allowed methods and return an empty response, while explicit OPTIONS handlers must take precedence. Applications must be able to disable automatic OPTIONS globally or for an individual route, and a view-level override should be honored when it conflicts with the default. Unsupported methods should continue to produce a useful 405 response with an accurate Allow header; teardown callback return values are ignored.

For example, @app.route("/")\n@web_api.after_this_request\ndef add_header(response):\n response.headers["X-Ready"] = "yes"\n return response should make a successful request include X-Ready: yes.

Stage 7 instruction

Applications that are split into blueprints need those components to remain first-class after registration: URL generation should honor blueprint-provided defaults, and failures should be routed to the most appropriate registered handler rather than a merely broad fallback. Building on the existing request dispatch and lifecycle behavior, make blueprint registration preserve the relationship between routes, defaults, and handlers.

A URL-default callback attached to a blueprint should be invoked for endpoints owned by that blueprint, including when values are supplied through normal URL building or explicitly injected. Blueprint-local error handlers should affect requests handled by that blueprint, while handlers intentionally registered for the application should remain global. When multiple exception handlers could match, prefer the most specific exception class; HTTP status handlers should continue to distinguish the actual HTTP exception being handled from a subclass-specific registration. Registration should also remain safe and predictable when the application state is initialized or reused.

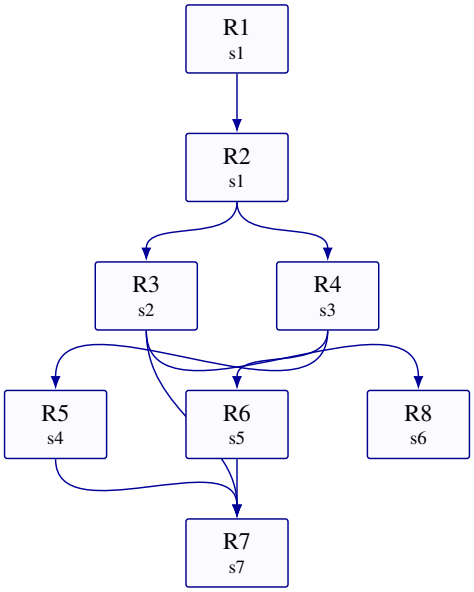
For example, registering a blueprint whose route contains <page> and whose default callback supplies page="login" should make url_for("foo.view") return /login.

Stage 8 instruction

Applications need a dependable request boundary so API endpoints and browser forms can consume submitted data without ambiguous decoding or opaque failures. Building on the existing response and session behavior, restore consistent handling of JSON, URL-encoded form data, and multipart uploads. Valid JSON should be available through the request API when its media type identifies JSON; invalid JSON or a non-JSON request should follow the framework's normal client-error behavior rather than being silently treated as a different payload. Form fields and genuine uploaded files must remain distinguishable. In debug mode, trying to use a file field that was submitted without file contents should raise a useful diagnostic that identifies the submitted filename. For example, @app.post("/echo")\ndef echo():\n return web_api.request.get_json()["name"] should return the submitted name for a JSON request containing {"name": "Ada"}. Keep these semantics compatible with the application's configured error handling and the existing request helpers.

H.1.7 PYTHON-06

7 stages, 8 requirements, 50 F2P tests, and 229 P2P tests.



Stage 1 instruction

The workflow library currently has no way to deduplicate concurrent identical requests. Add Runnable.with_coalesce(*, backend=None) so concurrent synchronous invoke calls with the same input execute the wrapped Runnable only once and every caller receives the shared result.

Define CoalesceBackend, CoalesceStats, and InMemoryCoalesceBackend in workflow_core.runnables.coalesce. Export exactly those three public coalescing types from workflow_core.runnables, and keep wrapper implementation classes private.

Different inputs must remain independent. Once an execution completes, a later call with the same input must run fresh; this is not a persistent result cache. The coalescing key is the input value only: configuration and keyword arguments must not affect it, and dictionaries with the same content but different key ordering must coalesce.

Stage 2 instruction

Continue working in the same workspace.

Define the backend lifecycle precisely:

- register(key) -> bool
- join(key)
- complete(key, *, result=None, error=None)
- is_active(key) -> bool
- stats -> CoalesceStats(active, coalesced, total)

Provide the async counterparts aregister, ajoin, acomplete, and ais_active. InMemoryCoalesceBackend must be thread-safe, and synchronous and asynchronous operations must observe the same in-flight state and statistics.

All supported methods on one coalescing wrapper must share its backend. Separate wrappers are independent by default, but wrappers given the same explicit backend must share in-flight state.

Stage 3 instruction

Continue working in the same workspace.

Extend coalescing to asynchronous invocation. Concurrent ainvoke calls with the same input must share one underlying execution. Async joiners must receive the same result as the leader, or the same error if the leader fails. The async backend lifecycle must mirror the synchronous behavior.

Stage 4 instruction

```
Continue working in the same workspace.

Extend coalescing to stream and astream while a stream is active.  Every joined
caller, including one that joins late, must receive the complete chunk sequence
replayed from the beginning.  A stream failure must propagate to every leader and
joined caller.
```

Stage 5 instruction

```
Continue working in the same workspace.

Extend coalescing to the batch APIs.  batch and abatch must coalesce each input item
independently while preserving positional output order; an empty batch must return an
empty output.

batch_as_completed and abatch_as_completed must coalesce duplicate inputs, yield those
duplicate results consecutively, and report the correct original index for every
item.
```

Stage 6 instruction

```
Continue working in the same workspace.

This requirement revises R2 and R3:  extend their synchronous and asynchronous
in-flight registries with lifecycle statistics, explicit clearing, cancellation, and
post-completion cleanup.  Their existing leader/joiner result-sharing guarantees
remain in force.

Keep this strictly as in-flight coalescing rather than a persistent result cache.
Completed calls must not be reused by later calls.

Expose coalesce_info() to return the current statistics and coalesce_clear() to clear
the coalescing state.  Clearing with no active work must be safe.  With active work,
it must cancel both synchronous and asynchronous waiters using asyncio.CancelledError
and reset the statistics.  Leader errors must propagate to current joiners but must
not poison later calls.
```

Stage 7 instruction

```
Continue working in the same workspace.

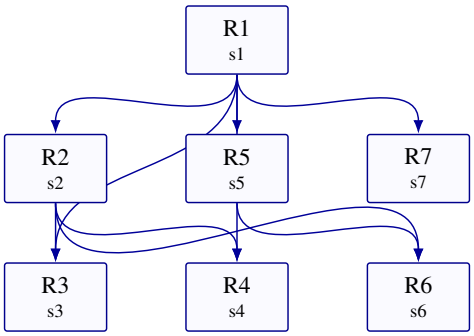
This requirement revises R1:  preserve its wrapper delegation contract while adding
the per-caller callback, graph, transform, and event-stream guarantees below.

Preserve Runnable integration behavior.  Every joined caller must still fire its own
chain-start and chain-end callbacks.  Graph delegation must remain transparent,
including when the wrapper participates in a larger chain.

transform and atransform must pass through without coalescing.  Event streaming,
including astream_events, must continue to work normally and must not invent extra
coalesced executions.
```

H.1.8 RUST-02

7 stages, 7 requirements, 104 F2P tests, and 12 P2P tests.



Stage 1 instruction

A user-facing gap has been isolated in parser expression optimization. The surrounding APIs are already present, so please make this first behavior dependable before we add more surface area.

Add and display CharClass(Vec<(String, String)>) in OptimizedExpr. As the final optimizer pass, coalesce qualifying full choice chains of single-character strings and ranges into sorted, merged ranges. Overlapping and adjacent ranges merge; a single result range simplifies to Range when endpoints differ or Str when equal.

Please ask for the eligibility, partial-chain, traversal, and negated-class details as you work. Preserve PEG behavior and existing optimizer output when coalescing is not valid.

Please keep earlier public names and error conventions. Exercise the ordinary case and the nearest invalid or empty case, then run the focused checks so a failure is visible rather than masked. Begin with the smallest fixture a caller would write, then confirm the same result when the feature is used through the public entry point.

Stage 2 instruction

The previous slice of parser expression optimization is usable in simple cases. We now need to handle the next scenario that appears in production, while keeping the earlier contract intact.

Only single-character Str/Insens, Range, and CharClass qualify. Empty/multi-character strings, identifiers, and other expressions block the relevant chain. Emit a result only when merging yields fewer ranges than the original alternatives; otherwise preserve Choice and PEG order.

When you verify this, include one repeated or reordered invocation; the result should stay deterministic and caller-owned state should remain untouched unless mutation is explicitly part of the API. Use a repeated invocation and a reordered input as a regression case; both should observe the same stable contract.

Stage 3 instruction

We have a concrete report from a project using parser expression optimization: the edge case below is difficult to diagnose today. Please address it in the existing design instead of introducing a parallel API.

A single-character Insens qualifies. Alphabetic characters expand to both letter cases before merging; upper/lower input is equivalent. Digits and symbols do not gain case variants. Multi-character Insens blocks, and profitability still applies.

Do not change unrelated modules or add a fallback that hides malformed input. A focused regression check should demonstrate both the useful result and the behavior at the boundary. Include one malformed, empty, or missing-value case so the established error or no-op behavior is explicit.

Stage 4 instruction

The core path in parser expression optimization now works, but callers still cross this boundary when they combine features. This step should make that composition predictable and observable.

When only some alternatives qualify, coalesce each contiguous qualifying run of at least three alternatives if that run is profitable. Runs of one/two and nonprofitable runs remain choices; multiple runs and blockers preserve PEG order.

Preserve supported runtime versions and existing output for callers that do not use this feature. Check the observable return value, error text, and ordering before handing it over. Compare this path with the equivalent operation from the preceding Step and record any intentional difference in the focused test.

Stage 5 instruction

We are preparing parser expression optimization for another release. Please close the behavior gap described below, including the failure path, without broad refactoring or silent compatibility changes.

Run coalescing top-down as the final optimizer pass across nested choices and children of Seq, Rep, Opt, Push and predicates, independently per rule/branch. Do not rewrite unrelated non-choice/range structure, and retain compatibility with earlier factorizer/restorer behavior.

Please compare the new path with the equivalent old path where one exists. The implementation is ready only when the repository checks cover success, failure, and

cleanup without relying on private test selectors. Exercise two features together, because the main risk here is a state or metadata leak across the boundary described above.

Stage 6 instruction

At this point the implementation needs a careful integration pass. In parser expression optimization, the following behavior is what a real caller sees when state changes mid-operation.

A RestoreOnErr-wrapped alternative qualifies exactly when its inner expression qualifies, and the wrapper is stripped from the coalesced output. Nonqualifying wrapped Push remains wrapped. Restorer runs before coalescer.

Keep the change local to the requested behavior and leave a reproducible check in the existing test entry point. If an operation is repeated, it must not accumulate stale state. Check cleanup and reuse after the operation completes or fails; the next caller should start from a clean state.

Stage 7 instruction

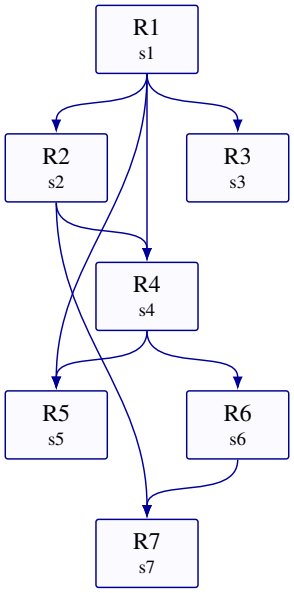
One last compatibility issue remains in parser expression optimization. Treat this as a request from an application maintainer: solve the concrete problem and leave unrelated behavior alone.

Add NegCharClass(Vec<(String,String)>). A negated predicate over qualifying alternatives immediately followed by ANY collapses to merged excluded ranges. Missing ANY or nonqualifying content remains unchanged.

Before delivery, run the focused repository checks and one integration example. Report real failures instead of making the feature silently permissive. Keep the example small enough to run locally, but make its expected output precise enough for a reviewer to diagnose a regression.

H.1.9 TYPESCRIPT-02

7 stages, 7 requirements, 51 F2P tests, and 12 P2P tests.



Stage 1 instruction

A user-facing gap has been isolated in composite aspects. The surrounding APIs are already present, so please make this first behavior dependable before we add more surface area.

Export a new createAspect API from the library's core module. It accepts at least two traits and returns a distinct aspect exposing id, flattened constituent traits, and a merged schema. Tag traits are valid, while relations and overlapping field names must be rejected.

This is an interactive task. Ask the user for details about entity operations, add/remove behavior, query projection, query modifiers, or subscriptions as those areas become relevant.

IMPORTANT: Please work on this in a new branch from main and commit everything when you are done.

Please keep earlier public names and error conventions. Exercise the ordinary case and the nearest invalid or empty case, then run the focused checks so a failure is visible rather than masked. Begin with the smallest fixture a caller would write, then confirm the same result when the feature is used through the public entry point.

Stage 2 instruction

The previous slice of composite aspects is usable in simple cases. We now need to handle the next scenario that appears in production, while keeping the earlier contract intact.

Entity.has(aspect) requires every constituent. get(aspect) merges data or returns undefined when incomplete and excludes tag fields. set(aspect, partial) distributes owned fields and performs per-constituent change detection. Tag constituents participate in completeness.

When you verify this, include one repeated or reordered invocation; the result should stay deterministic and caller-owned state should remain untouched unless mutation is explicitly part of the API. Use a repeated invocation and a reordered input as a regression case; both should observe the same stable contract.

Stage 3 instruction

We have a concrete report from a project using composite aspects: the edge case below is difficult to diagnose today. Please address it in the existing design instead of introducing a parallel API.

Entity.add(aspect) adds only missing constituents, uses defaults when no arguments are supplied, and distributes initial fields without overwriting existing constituents. Entity.remove(aspect) removes every constituent and leaves unrelated traits intact.

Do not change unrelated modules or add a fallback that hides malformed input. A focused regression check should demonstrate both the useful result and the behavior at the boundary. Include one malformed, empty, or missing-value case so the established error or no-op behavior is explicit.

Stage 4 instruction

The core path in composite aspects now works, but callers still cross this boundary when they combine features. This step should make that composition predictable and observable.

An aspect query requires all constituents and updates as completeness changes. readEach returns merged aspect data in parameter order alongside normal traits. updateEach distributes writes and triggers change only for modified constituents. Aspect queries remain valid across world reset.

Preserve supported runtime versions and existing output for callers that do not use this feature. Check the observable return value, error text, and ordering before handing it over. Compare this path with the equivalent operation from the preceding Step and record any intentional difference in the focused test.

Stage 5 instruction

We are preparing composite aspects for another release. Please close the behavior gap described below, including the failure path, without broad refactoring or silent compatibility changes.

Not(aspect) matches entities missing at least one constituent and updates as completeness changes. Aspects compose with required traits, Or, multiple aspects, nested aspects, and tag-only aspects.

Please compare the new path with the equivalent old path where one exists. The implementation is ready only when the repository checks cover success, failure, and cleanup without relying on private test selectors. Exercise two features together, because the main risk here is a state or metadata leak across the boundary described above.

Stage 6 instruction

```
At this point the implementation needs a careful integration pass.  In composite
aspects, the following behavior is what a real caller sees when state changes
mid-operation.

Changed(aspect) matches when any constituent data changes.  Added(aspect) matches
transition to all-present including spawn-complete, but not an already-complete
entity.  Removed(aspect) matches transition from complete to incomplete, not entities
that were never complete.

Keep the change local to the requested behavior and leave a reproducible check in the
existing test entry point.  If an operation is repeated, it must not accumulate stale
state.  Check cleanup and reuse after the operation completes or fails; the next
caller should start from a clean state.
```

Stage 7 instruction

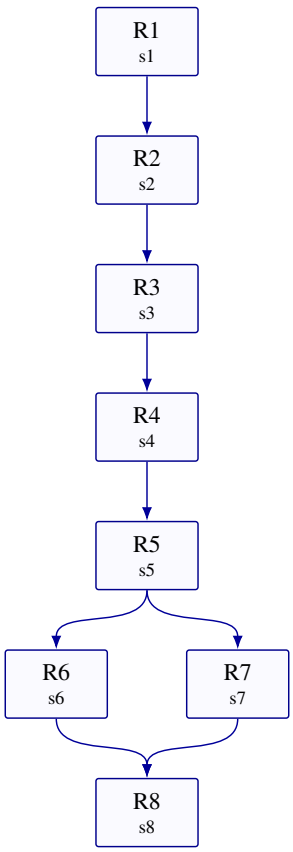
```
One last compatibility issue remains in composite aspects.  Treat this as a request
from an application maintainer:  solve the concrete problem and leave unrelated
behavior alone.

onAdd fires exactly on transition from incomplete to complete, including complete
spawn.  onRemove fires only from complete to incomplete.  onChange fires for a
constituent change only while complete.  Subscription methods return working
unsubscribe functions.

Before delivery, run the focused repository checks and one integration example.
Report real failures instead of making the feature silently permissive.  Keep the
example small enough to run locally, but make its expected output precise enough for
a reviewer to diagnose a regression.
```

H.1.10 `go-01`

8 stages, 8 requirements, 132 F2P tests, and 13 P2P tests.



Stage 1 instruction

A user-facing gap has been isolated in the structured logging package. The surrounding APIs are already present, so please make this first behavior dependable before we add more surface area.

Users need the logging package to behave predictably when configuration, structured fields, and shutdown cleanup cross API boundaries. Restore the observable compatibility of level values, JSON-formatted entries, and registered exit handlers without requiring callers to work around serialization or ordering surprises.

Levels should use their established names when converted to and from text, while invalid input reports an error. Structured fields must remain usable in JSON output: error values should be represented by their messages, unsupported values should not make logging panic, and normal fields should still produce valid JSON. JSON output should honor the formatter's HTML-escaping option, including values containing markup-sensitive characters. Registered cleanup handlers must run when the process exits through the logging API, preserve registration order, keep deferred handlers in reverse order, and retain the requested exit status even if a handler panics.

For example, (&log_api.JSONFormatter{DisableHTMLEscape: true}).Format(&log_api.Entry{Message: "<ok>", Data: log_api.Fields{"error": errors.New("bad")}}) should emit valid JSON containing the literal <ok> and the error message bad.

Keep the behavior compatible with existing logger and entry usage, including custom field names and ordinary non-JSON formatting.

Please keep earlier public names and error conventions. Exercise the ordinary case and the nearest invalid or empty case, then run the focused checks so a failure is visible rather than masked. Begin with the smallest fixture a caller would write, then confirm the same result when the feature is used through the public entry point.

Stage 2 instruction

The previous slice of the structured logging package is usable in simple cases. We now need to handle the next scenario that appears in production, while keeping the earlier contract intact.

Applications need logging that remains useful across request boundaries, terminal environments, and concurrent workers. Extend the existing structured logger so a context attached to a logger or entry is preserved by derived entries and is available to hooks and other logging integrations. Respect the configured severity threshold consistently, while keeping enabled entries' fields and messages intact.

Text output should make terminal presentation predictable: automatic color detection should work as expected, forced coloring should be honored, and explicitly disabling colors must produce plain output even when coloring would otherwise be selected. Formatter options should continue to govern the rendered level, fields, and timestamps without corrupting user data. Writes and formatting should be serialized by default so shared output remains coherent under concurrent logging; callers that deliberately manage their own synchronization should still be able to opt out.

For example:

```
entry := log_api.New().WithContext(ctx).WithField("request_id", "42")
entry.Info("started")
```

The emitted text contains the request field without unwanted ANSI escapes when colors are disabled, and integrations receiving the entry can still retrieve ctx. Nil or unsupported configuration should fail in the same conventional way as the rest of the public logger API rather than silently changing the message.

When you verify this, include one repeated or reordered invocation; the result should stay deterministic and caller-owned state should remain untouched unless mutation is explicitly part of the API. Use a repeated invocation and a reordered input as a regression case; both should observe the same stable contract.

Stage 3 instruction

We have a concrete report from a project using the structured logging package: the edge case below is difficult to diagnose today. Please address it in the existing design instead of introducing a parallel API.

Users need to fit log output into their own schemas and terminal conventions without post-processing every record. Building on the existing structured and context-aware logging behavior, restore the formatter options that control timestamps, field naming, terminal presentation, and the bytes returned for each entry.

JSON output should honor custom field mappings and timestamp settings while remaining valid, newline-terminated JSON. Text output should support the expected colored or uncolored presentation, optional timestamp suppression, and custom caller formatting, with fields and special values rendered consistently. Formatter output must remain safe to consume as a byte slice, including when buffers are reused, and customizations should not cause metadata or message content to disappear. Disabling a presentation feature should affect only that feature, not the other record fields.

For example, configuring a JSON formatter with a field map that renames the message field should produce a record containing the renamed key, the original message value, and a trailing newline: {"msg_text":"ready"}\n.

Do not change unrelated modules or add a fallback that hides malformed input. A focused regression check should demonstrate both the useful result and the behavior at the boundary. Include one malformed, empty, or missing-value case so the established error or no-op behavior is explicit.

Stage 4 instruction

The core path in the structured logging package now works, but callers still cross this boundary when they combine features. This step should make that composition predictable and observable.

Users need the convenient println-style logging APIs to produce the same readable messages as ordinary Go printing while still honoring configured log levels and hooks. Building on the existing structured and customizable logging behavior, restore the variadic line-oriented methods so callers can mix strings and non-string values without surprising spacing or extra blank lines.

The resulting message should preserve Go's Sprintln-style rule: adjacent string operands remain adjacent, while spaces are inserted between non-string operands as appropriate. A disabled level must produce no output and must not notify hooks; an enabled level should deliver one entry containing the final message. The generic level-based form should log at the requested severity without turning a Panic or Fatal level into an implicit panic or process exit; callers using the dedicated control-flow helpers must retain their established behavior.

For example, logger.Infoln("ready", 3, "items") should emit one info record whose message is ready 3 items, with the formatter supplying the single record terminator rather than an extra newline. Ensure the behavior is consistent when called directly or through derived entries carrying fields and context.

Preserve supported runtime versions and existing output for callers that do not use this feature. Check the observable return value, error text, and ordering before handing it over. Compare this path with the equivalent operation from the preceding Step and record any intentional difference in the focused test.

Stage 5 instruction

We are preparing the structured logging package for another release. Please close the behavior gap described below, including the failure path, without broad refactoring or silent compatibility changes.

Building on the existing logging and formatter compatibility, make structured entry enrichment and the remaining message-producing paths dependable for callers that need rich, deferred events. Fields added through an entry or logger should be merged into a new entry without mutating the original; later values replace earlier keys, and the configured error field should turn error-like values into readable error text while ordinary unsupported values remain safe to log. An explicit time override must travel with the derived entry and be the timestamp seen by formatters and hooks.

Formatted and lazy logging should honor the logger's level before doing work, emit the callback's returned arguments with normal message semantics, and preserve fields, context, error data, and time when the entry is delivered. Generic level-based emission should remain a logging operation rather than unexpectedly panicking or terminating; callers that need those side effects can use the dedicated convenience methods. For example, logger.WithFields(log_api.Fields{"request": id}).WithTime(start).LogFn(log_api.InfoLevel, func() []interface{} { return []interface{}{"started"} }) should produce one info event carrying the request and overridden timestamp, while the callback is skipped when info is disabled.

Please compare the new path with the equivalent old path where one exists. The implementation is ready only when the repository checks cover success, failure, and cleanup without relying on private test selectors. Exercise two features together, because the main risk here is a state or metadata leak across the boundary described above.

Stage 6 instruction

At this point the implementation needs a careful integration pass. In the structured logging package, the following behavior is what a real caller sees when state changes mid-operation.

Users need structured logs to remain trustworthy when entries carry errors, conflicting fields, or application-failure handlers. Building on the existing logging and entry APIs, restore the behavior that accepts genuine error values consistently, rejects unsupported function-like values without corrupting the entry, and records any resulting field problem in the established logger error field. Field enrichment must preserve existing metadata while applying predictable precedence when keys clash, and output must still honor the logger's configured level and writer destination.

Failure handling should also be dependable: registered exit handlers run in registration order before the process terminates with the requested status, deferred exit handlers run in reverse order, and panic-oriented logging retains the panic information while invoking the configured hooks without silently losing the entry. These behaviors must remain safe when custom writers or hooks are used.

For example, an entry created with WithError(errors.New("disk full")) and emitted as JSON should contain the configured error key with the message "disk full", while unrelated fields remain intact.

Keep the change local to the requested behavior and leave a reproducible check in the existing test entry point. If an operation is repeated, it must not accumulate stale state. Check cleanup and reuse after the operation completes or fails; the next caller should start from a clean state.

Stage 7 instruction

One last compatibility issue remains in the structured logging package. Treat this as a request from an application maintainer: solve the concrete problem and leave unrelated behavior alone.

Applications using this logging package should be able to adopt Go's standard log/slog API without replacing their configured logger, hooks, formatter, or output. Building on the existing logging behavior, add a slog handler bridge and compatible level conversion.

NewHandler should return a slog-compatible handler backed by the supplied logger. For example, base := log_api.New(); logger := slog.New(slog_bridge.NewHandler(base, nil)); logger.Info("served", "request_id", 42) should emit an info entry through base with the request_id field attached.

The handler must use mapped logger severity for enablement and record delivery, preserving the record message, timestamp, context, and structured attributes. Attributes added before logging and nested groups must remain isolated across derived handlers, with group names represented as dotted field prefixes. Empty groups and resolved values should follow normal slog semantics. Optional source reporting should preserve the caller carried by the slog record, and callers must be able to supply a custom level mapper.

Provide sensible bidirectional defaults across trace, debug, info, warning, error, fatal, and panic, including stable handling of intermediate or out-of-range values. Records mapped to fatal or panic must retain that severity without terminating or panicking. Reject a nil backing logger clearly, and return errors from normal log record processing rather than silently discarding them.

Before delivery, run the focused repository checks and one integration example. Report real failures instead of making the feature silently permissive. Keep the example small enough to run locally, but make its expected output precise enough for a reviewer to diagnose a regression.

Stage 8 instruction

This is the next small change requested by users of the structured logging package. The goal is a stable, documented contract that behaves the same through the public and internal entry points.

Applications using the existing standard-library slog bridge need to build contextual loggers without losing structured data or contaminating sibling loggers. Building on the existing bridge, make handler derivation through pre-bound attributes and groups behave like normal slog composition.

Attributes attached to a derived handler must be retained for every later record, while grouped attributes remain associated with the requested group and preserve their values, including errors and byte-oriented values. Deriving another handler

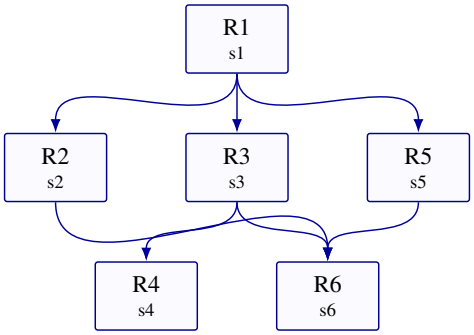
must not mutate its parent, so independently enriched or grouped loggers can safely be reused. Record delivery must continue to honor configured level filtering and hooks, and any failure encountered while handling a record should be returned rather than silently discarding the record's data.

For example, base.WithGroup("request").With("id", 42).Info("served") should emit the message with id=42 scoped beneath the request group, while records written through base remain ungrouped and do not inherit that attribute.

The final behavior should read naturally to a caller: clear result on success, established error on bad input, and no surprising side effect on an unrelated operation. When the repository has more than one adapter or format, run the example through both rather than validating only an internal helper.

H.1.11 JAVASCRIPT-03

6 stages, 6 requirements, 90 F2P tests, and 12 P2P tests.



Stage 1 instruction

A user-facing gap has been isolated in test-runner bail handling. The surrounding APIs are already present, so please make this first behavior dependable before we add more surface area.

Please add bail_on_test_failure to the test runner's configuration defaults. false keeps the current behavior, true means stop after the first failure, and a positive integer N means stop once N failures have been observed. Normalize the value during configuration so all runners receive the same representation, and reject negative or non-integer values with the existing configuration error path. The bail state must be visible to the app orchestration layer, reset for each run, and remain disabled when the option is omitted.

Please keep earlier public names and error conventions. Exercise the ordinary case and the nearest invalid or empty case, then run the focused checks so a failure is visible rather than masked. Begin with the smallest fixture a caller would write, then confirm the same result when the feature is used through the public entry point.

Stage 2 instruction

The previous slice of test-runner bail handling is usable in simple cases. We now need to handle the next scenario that appears in production, while keeping the earlier contract intact.

Please give BrowserTestRunner an idempotent asynchronous abort operation. When bail is reached it should send the abort-tests command once, resolve even if the browser is already stopping, and suppress result messages or exit errors that arrive after the abort. A second call must be harmless, and a browser that has not started should not receive a command. Keep the runner's normal cleanup and promise ownership intact so a non-bailing run behaves exactly as before.

When you verify this, include one repeated or reordered invocation; the result should stay deterministic and caller-owned state should remain untouched unless mutation is explicitly part of the API. Use a repeated invocation and a reordered input as a regression case; both should observe the same stable contract.

Stage 3 instruction

We have a concrete report from a project using test-runner bail handling: the edge case below is difficult to diagnose today. Please address it in the existing design instead of introducing a parallel API.

Please update the dot and TAP reporters to describe a bail stop clearly. They should record the configured threshold, the failure count that triggered it, how many tests were suppressed, and the original failure details. Keep valid dot/TAP syntax and existing summaries for runs that complete normally. Output must remain deterministic when several failures arrive in the same tick.

Do not change unrelated modules or add a fallback that hides malformed input. A focused regression check should demonstrate both the useful result and the behavior at the boundary. Include one malformed, empty, or missing-value case so the established error or no-op behavior is explicit.

Stage 4 instruction

The core path in test-runner bail handling now works, but callers still cross this boundary when they combine features. This step should make that composition predictable and observable.

Please make TeamCity and XUnit reporters emit parseable bail results. A stopped run should contain the triggering comparison failure, skipped or suppressed test counts, and the bail metadata without pretending that unstarted tests passed. Preserve the existing XML escaping, suite names, and normal success/failure layout for a run that reaches completion.

Preserve supported runtime versions and existing output for callers that do not use this feature. Check the observable return value, error text, and ordering before handing it over. Compare this path with the equivalent operation from the preceding Step and record any intentional difference in the focused test.

Stage 5 instruction

We are preparing test-runner bail handling for another release. Please close the behavior gap described below, including the failure path, without broad refactoring or silent compatibility changes.

Please propagate bail through the process runner and TAP process runner. Once the threshold fires, stop launching subsequent work, retain the tests that already started, and emit one bail event that reporters can consume. A child process that exits after the stop must not overwrite the recorded failure count. Keep signal handling and ordinary non-bailing process execution unchanged.

Please compare the new path with the equivalent old path where one exists. The implementation is ready only when the repository checks cover success, failure, and cleanup without relying on private test selectors. Exercise two features together, because the main risk here is a state or metadata leak across the boundary described above.

Stage 6 instruction

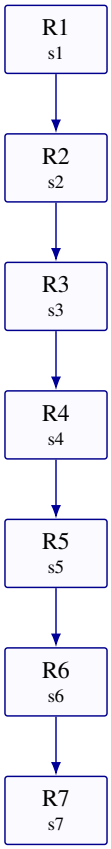
At this point the implementation needs a careful integration pass. In test-runner bail handling, the following behavior is what a real caller sees when state changes mid-operation.

Please wire the bail option through Reporter, Server, browser adapters, configuration, and the test-runner CLI. Every entry point should share one lifecycle state and should clean it up when a run ends, errors, or is cancelled. CLI values must use the same normalization as programmatic configuration, and adapters that do not support abort should report suppression without crashing. Existing configuration files and reporter APIs must remain source-compatible.

Keep the change local to the requested behavior and leave a reproducible check in the existing test entry point. If an operation is repeated, it must not accumulate stale state. Check cleanup and reuse after the operation completes or fails; the next caller should start from a clean state.

H.1.12 PYTHON-01

7 stages, 7 requirements, 94 F2P tests, and 19 P2P tests.



Stage 1 instruction

```
The name resolver is losing track of assignments when a statement is both a use and a
definition.  Please restore the control-flow filtering used by lookup/ilookup so that
an expression sees the bindings that are actually visible at its source position, not
every assignment that happens to occur in the surrounding block.  In particular, a
name used in an if condition must not see an assignment from the body that has not
run yet, while a later statement should see a binding that was definitely established
earlier.  Keep the existing treatment of branch-local and mutually exclusive
assignments:  an ambiguous branch should yield the established inference results
rather than silently picking one value.

Please make the behavior consistent for the node-level helpers that filter assignment
statements, including nested scopes and the \assignment after use" cases.  Do not
change public return types or exception conventions, and avoid solving this by
changing the test fixtures or by making lookup order-dependent.  A small example
worth checking is parsing x = 1\nif x:\n x = 2\nvalue = x:  the condition should
resolve to the first assignment, and the final expression should retain the normal
branch-aware inference semantics.  Run the focused builder/lookup checks and keep
unrelated AST behavior unchanged.
```

Stage 2 instruction

```
Please repair the inference and builder plumbing that callers rely on when
constructing a module from source.  The lookup mixin should expose the same ilookup
contract as before, including stable handling of globals, function arguments, and
augmented assignments; inference failures must continue to use the project's
dedicated exception types and preserve their useful context.  The builder also needs
to keep inferred values from leaking into the module's declared locals:  parsing or
inferring a name may consult a scope, but it must not mutate that scope merely as a
side effect of inspection.
```

Pay attention to signatures with positional-only, keyword-only, default, and variadic arguments. They should produce the same argument nodes and inferred values whether the function is built from text or inspected after construction. AugAssign must retain its target and value relationships, and an unsuccessful lookup should remain distinguishable from an empty successful result. Please exercise normal, missing-name, and repeated-inference paths while keeping the established node classes and exception messages compatible. The change should fit the existing builder/rebuilder design rather than adding a parallel lookup implementation.

Stage 3 instruction

The rebuilt AST is missing a few structural guarantees that make name inference and tree inspection reliable. Please restore the rebuilder's handling of pass and wrapped expressions, ensure every node reports the correct parent_of relationship, and keep get_children() ordering and node comparison stable for binary operations and their operands. These relationships are observable to visitors and to callers that walk a tree, so a shortcut that merely makes one inference assertion pass is not sufficient.

Control-flow cleanup must also remain precise. Assignments introduced by del, except ... as name, function parameters, and augmented assignments should be visible or hidden according to Python's block rules; deleting a name in one branch must not erase an unrelated binding in another branch. The builder should still handle empty blocks and nested try/except structures without manufacturing placeholder nodes. Please validate both the structural APIs (parents, children, comparison) and the resulting lookup behavior on ordinary, nested, and exceptional control flow. Preserve source order, node types, and the existing public exception behavior while making the smallest coherent change in the rebuilder and node classes.

Stage 4 instruction

Source locations and special-form inference need to survive a build/rebuild round trip. Please restore the builder and rebuilder behavior that assigns meaningful lineno, col_offset, end_lineno, and end_col_offset (or the project's equivalent tolineno/blockstart) to calls, classes, functions, loops, conditionals, try blocks, decorators, and docstring-bearing definitions. A node created from a generator expression or a subscript should keep the same scope and inference context as the source tree, including the project's typing and NumPy \looks like" helpers.

The goal is not to invent a new location convention. Match the conventions already used by neighboring node classes, preserve postinit and visitor behavior, and keep line numbers monotonic when a construct spans several lines. Invalid or unusual source should still produce the established error or a well-formed node, rather than a partially initialized object. Please check single-line and multi-line definitions, decorated definitions, for/while, if, with, and try/finally examples, then verify that tree traversal and subscript inference still see the same children and scope. Keep the patch focused on the location/rebuilder and special-form helpers.

Stage 5 instruction

Please bring class, function, and module scope inspection back in line with the public syntax-tree model. Decorated definitions should expose the same scope and inferred bases as undecorated ones, dataclass-like decorators must not erase class locals, and locals, bases, and related properties should remain usable for modules, classes, methods, static methods, and nested functions. When a source file declares an unknown encoding, retain the established parsing error instead of silently decoding it with a different default.

This is also where the builder's inference constraints and error objects meet. NameInferenceError and the related resolver exceptions should keep their expected initialization and context, while constraint matching must continue to distinguish a boolean condition from an equality condition. Do not eagerly evaluate decorators or mutate a scope merely to answer an inspection query. For example, asking for a class's bases and locals after parsing should return stable collections, and asking for a method's scope should not add inferred helper names to it. Preserve the existing public node types and return shapes, and cover decorated classes, methods, modules, encoding failures, and slot-based objects before considering the change complete.

Stage 6 instruction

Name lookup must remain useful across the scopes users actually inspect: builtins, modules, functions, methods, nested classes, lambdas, comprehensions, and loop variables. Please restore the manager, rebuilder, and lookup interactions so that lookup/ilookup find the nearest valid definition, respect an explicit __name__, and honor the configured inference limit without leaking names between sibling scopes. Nested list, set, and dictionary comprehensions should have their own target bindings while still resolving names from the surrounding function or module.

The same scope rules affect the small transforms used for I/O wrappers and the constraint objects used by inference. Keep their public match and transform results compatible, including the behavior for an object that cannot be transformed and for an empty or repeated lookup. A caller should be able to inspect a nested function's argument default, a class variable, or a builtin name and receive a deterministic node sequence, not a result that depends on traversal order. Please test normal nested scopes, deleted globals, comprehension targets, decorators, and the inference-limit boundary. Avoid broad changes to unrelated brain plugins; preserve established exceptions and node relationships while fixing the lookup path.

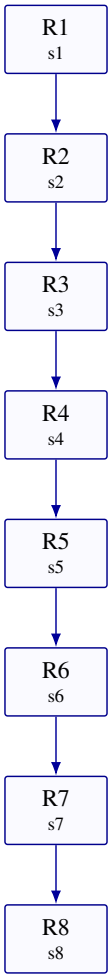
Stage 7 instruction

Please complete the comprehension nodes and their rebuilder hooks so dictionary, set, and nested comprehensions behave like the rest of the AST. A DictComp must retain its key/value and generator children in source order, expose them through get_children(), and carry the right frame and scope when postinit finishes. The rebuilder should construct the same node shape for a one-line comprehension, a nested comprehension, and a comprehension inside a decorated function; name inference must not accidentally bind a target in the enclosing scope.

There are a few parser edge cases in the same path: a module with no trailing newline, a module-level __file__ expression, and an inner decorator member lookup should all produce a complete, inspectable tree. Keep visit_dictcomp and the surrounding frame/decorator behavior compatible with existing visitors, including empty generators and invalid input handling. Please check that get_children() is repeatable, that inferred values from nested comprehensions remain stable across two calls, and that rebuilding does not change line locations or parent links. Keep the change confined to the comprehension/frame/rebuilder contracts instead of weakening inference globally.

H.1.13 PYTHON-04

8 stages, 8 requirements, 73 F2P tests, and 19 P2P tests.



Stage 1 instruction

Users need DNS message objects to behave consistently as they move between query and response workflows, including protocol validation, comparison, and section inspection. Restore the message semantics so response construction rejects an object that is already a response, response-oriented operations distinguish queries from malformed responses, and chaining only succeeds for structurally valid response messages. Message identity must account for meaningful flags and record contents rather than only the transaction ID, while a response indicating an error may still match its query when the protocol permits an empty question section. Text containing only comments should still produce a usable query message, and valid numeric section selectors should return the corresponding section while unknown selectors raise a clear value error. For example, dns.message.make_response(dns.message.make_response(dns.message.make_query("foo", "A"))) should raise dns.exception.FormError instead of creating a nested response.

Stage 2 instruction

Users need DNS messages to negotiate EDNS capabilities, carry options, and express DNSSEC-related preferences without losing ordinary message semantics. Building on the existing DNS message behavior, restore the EDNS-aware configuration and inspection that callers rely on.

Enabling EDNS should encode the requested version and flags, use the standard default payload when no payload is supplied, retain request payload and options, and report a disabled state with zero payload and no options when EDNS is turned off. Boolean

EDNS arguments should retain their established meanings. The DNSSEC-desired toggle must enable EDNS when necessary and set or clear the DO bit appropriately. Extended response codes must round-trip through the message and EDNS fields while ordinary response codes should not introduce an unnecessary EDNS record. Structured EDNS options, including extended DNS errors, should remain available after message construction and response inspection.

Text message parsing should accept payload configuration and reject incomplete question lines or unknown classes and types with the appropriate DNS exceptions. For example, m = dns.message.make_query("foo", "A"); m.use_edns(); assert m.payload == 1232.

Stage 3 instruction

Users need to build DNS messages at the wire boundary and recover them without losing their meaning, including when size limits, EDNS records, or nonstandard opcodes are involved. Building on the existing message semantics and EDNS configuration, complete the renderer/parser behavior so callers can add questions and RRsets in section order, account for message size safely, reserve and release space, and obtain valid wire output with optional EDNS and TSIG data. Failed or oversized additions should not leave partial bytes or incorrect section counts behind. The parser should provide reliable bounded reads for integers, counted fields, and names, while temporary limits and seeking preserve cursor and furthest-read behavior and reject malformed input cleanly. Wire and text conversions should retain flags, records, options, and the appropriate message kind instead of normalizing every message to a query. For example, q1 = dns.message.Message(id=1); q1.set_opcode(dns.opcode.NOTIFY); q1.flags |= dns.flags.AA; q1.find_rrset(q1.question, dns.name.from_text("example"), dns.rdataclass.IN, dns.rdatatype.SOA, create=True, force_unique=True); assert dns.message.from_wire(q1.to_wire()) == q1 should round-trip as a generic message.

Stage 4 instruction

Building on the existing DNS message construction and wire support, make message transformations predictable for applications that turn queries into responses or exchange human-readable DNS text. Response creation must honor the requested copy policy: the default and question-only modes retain the question while leaving other sections empty, the everything mode preserves all sections, and the nothing mode preserves none. Copied content should remain valid message data without unexpectedly changing the source.

Text parsing should produce the appropriate message kind, ignore permitted header comments, preserve meaningful message equality, and reject malformed section records with the established DNS exceptions rather than silently accepting incomplete names, classes, types, or rdata. Public section access should stay consistent with the answer, authority, and additional collections, while parsed flags remain usable as DNS flag values and parsed wire data retains its original wire representation when available. For example, response = dns.message.make_response(query, copy_mode=dns.message.CopyMode.QUESTION) should contain the query question and no answer, authority, or additional records.

Stage 5 instruction

Applications that inspect or construct DNS traffic need message objects to remain trustworthy after parsing, mutation, and re-serialization. Complete the message behavior so wire output, comparisons, and dynamic-update bookkeeping stay consistent, while malformed special records are rejected rather than silently accepted.

Building on the existing DNS message parser and serializer, ensure parsed messages can retain and regenerate their original wire representation when appropriate, and that equality reflects meaningful message content rather than object identity. Dynamic update builders should place prerequisite and update records in the correct sections and report accurate counts as records are added, replaced, or deleted. Wire parsing must enforce the protocol's placement, uniqueness, owner-name, and class rules for EDNS and TSIG records; invalid layouts should raise the established DNS message errors. Preserve canonical padding behavior so requested message sizes do not produce unexpectedly short or oversized output, and keep basic message inspection stable and useful.

For example, a parsed message whose cached wire is cleared and then serialized again produces the same wire bytes as the original message.

Stage 6 instruction

Users need DNS text fixtures and zone-style data to round-trip predictably across messages, names, and RRsets, without losing the selected naming or encoding rules. Building on the existing message and wire support, make the text-facing interfaces agree on canonical formatting and accept the same public representations when inspecting sections and records.

Message text loaded from a string or file should produce the same meaningful structure and rendering as equivalent wire data. Name and rdata text conversion must honor an explicitly selected IDNA codec, while ordinary absolute names remain stable. RRset lookup should work with either a section collection or its public section selector. Query-chain resolution must terminate for incomplete or non-followable answers and report the current name and applicable minimum TTL rather than looping.

For example, `m = dns.message.from_text(text, idna_codec=dns.name.IDNA_2003); assert str(m) == expected_text`.

Keep established parsing and serialization error behavior intact, including malformed text and invalid name escapes; do not silently reinterpret invalid input.

Stage 7 instruction

Users need DNS messages to remain interoperable when encoded, parsed, inspected, and returned under real packet-size limits. Building on the existing message and text behavior, restore the remaining wire-level semantics so callers can preserve EDNS data, recognize incomplete responses, and work with sections consistently.

Wire parsing should retain supported EDNS metadata and unknown options rather than discarding them, while malformed or incomplete input must fail with the appropriate DNS exception; when truncation is explicitly requested, a packet marked truncated must raise dns.message.Truncated. Message section helpers should accept the established section representations, including enum or textual forms, and return consistent RRset results for equivalent lookups. Serialization must enforce the requested maximum size, raise dns.exception.TooBig when ordinary output cannot fit, and support preferred truncation with DNS-correct TC signaling depending on which section is cut. EDNS padding should produce the requested wire size and remain compatible with parse-and-compare round trips; section counts must include automatically managed EDNS and TSIG records.

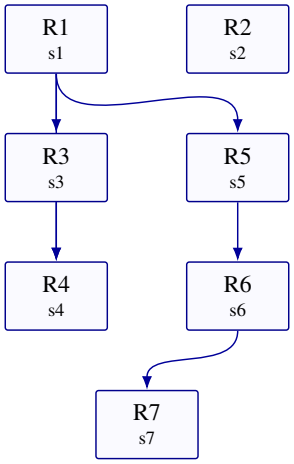
For example, `m2 = dns.message.from_wire(dns.message.make_query("foo", "A").to_wire())` should produce a message equivalent to the original query.

Stage 8 instruction

DNS clients need wire handling that is both diagnosable when an upstream packet is damaged and lossless when a valid packet is authenticated or padded. Building on the existing DNS message and wire support, make tolerant parsing retain all records that can still be decoded, expose ordered parsing errors with useful exceptions, and reject malformed boundaries rather than silently fabricating data. In particular, invalid AAAA RDATA should report that IPv6 addresses must be 16 bytes long, while later valid content remains available and the message is marked malformed. Signed queries must also survive serialization without losing their TSIG state, EDNS options, or wire padding, and parsing the resulting packet with the matching keyring should reproduce the original message semantics. For example, `q = dns.message.make_query( "www.example", "a", use_edns=0, options=[dns.edns.ECSOption("1.2.3.0", 24)], pad=128); q.use_tsig(keyring); assert dns.message.from_wire(q.to_wire(), keyring=keyring) == q` should hold, including the padded wire length and authenticated result.

H.1.14 PYTHON-07

7 stages, 7 requirements, 19 F2P tests, and 10 P2P tests.



Stage 1 instruction

A user-facing gap has been isolated in exception-group matching. The surrounding APIs are already present, so please make this first behavior dependable before we add more surface area.

Please extend RaisesGroup matching with flatten_subgroups=True and allow_unwrapped=True. Flatten nested exception groups recursively before counting and matching when requested. With allow_unwrapped=True, allow a single bare exception to match the only non-group matcher, but reject ambiguous multi-matcher cases. Keep the default strict behavior, context-manager semantics, and supported Python-version fallbacks unchanged.

Please keep earlier public names and error conventions. Exercise the ordinary case and the nearest invalid or empty case, then run the focused checks so a failure is visible rather than masked. Begin with the smallest fixture a caller would write, then confirm the same result when the feature is used through the public entry point.

Stage 2 instruction

The previous slice of exception-group matching is usable in simple cases. We now need to handle the next scenario that appears in production, while keeping the earlier contract intact.

Please add use_library_deprecation_warning: bool = False to warn_deprecated and deprecated. The default must emit the standard DeprecationWarning; only an explicit True selects the library's warning class. Preserve stacklevel, message formatting, decorator behavior, and the existing compatibility path for Python versions that do not expose the newer warning helpers.

When you verify this, include one repeated or reordered invocation; the result should stay deterministic and caller-owned state should remain untouched unless mutation is explicitly part of the API. Use a repeated invocation and a reordered input as a regression case; both should observe the same stable contract.

Stage 3 instruction

We have a concrete report from a project using exception-group matching: the edge case below is difficult to diagnose today. Please address it in the existing design instead of introducing a parallel API.

Please make the static typing contract for Matcher default its type parameter to BaseException when callers omit a generic argument, rather than falling back to Any. Runtime matching must stay unchanged, and the annotations must remain valid on every supported Python version. Check nested RaisesGroup matchers and ordinary context-manager use so the improvement does not break inference.

Do not change unrelated modules or add a fallback that hides malformed input. A focused regression check should demonstrate both the useful result and the behavior at the boundary. Include one malformed, empty, or missing-value case so the established error or no-op behavior is explicit.

Stage 4 instruction

```
The core path in exception-group matching now works, but callers still cross this
boundary when they combine features.  This step should make that composition
predictable and observable.

Please revise the generics and overloads for Matcher and RaisesGroup.  The context
manager, matches() TypeGuard, nested groups, and check callback should all preserve
the exception type they actually inspect.  Remove the obsolete strictness parameter
from the public overloads while keeping runtime compatibility for existing call sites
and type-checkers.

Preserve supported runtime versions and existing output for callers that do not use
this feature.  Check the observable return value, error text, and ordering before
handing it over.  Compare this path with the equivalent operation from the preceding
Step and record any intentional difference in the focused test.
```

Stage 5 instruction

```
We are preparing exception-group matching for another release.  Please close the
behavior gap described below, including the failure path, without broad refactoring
or silent compatibility changes.

Please give Matcher.matches() and RaisesGroup.matches() useful failure diagnostics.
Clear fail_reason at the start of every call, then record a deterministic English
explanation for type mismatches, missing or unexpected exceptions, and failed
callbacks.  The context-manager error should include the same reason without leaking
object addresses, and a successful call must leave no stale failure text behind.

Please compare the new path with the equivalent old path where one exists.  The
implementation is ready only when the repository checks cover success, failure, and
cleanup without relying on private test selectors.  Exercise two features together,
because the main risk here is a state or metadata leak across the boundary described
above.
```

Stage 6 instruction

```
At this point the implementation needs a careful integration pass.  In
exception-group matching, the following behavior is what a real caller sees when
state changes mid-operation.

Please complete the multi-failure report.  When matches have already been paired,
start the report with a newline, <n> matched exception(s)., and another newline, then
add Too few exceptions raised!  or Unexpected exception(s)!  as appropriate.  Include
expected and raised identities, preserve indentation for nested groups, and keep the
original exception object available through the context manager.

Keep the change local to the requested behavior and leave a reproducible check in the
existing test entry point.  If an operation is repeated, it must not accumulate stale
state.  Check cleanup and reuse after the operation completes or fails; the next
caller should start from a clean state.
```

Stage 7 instruction

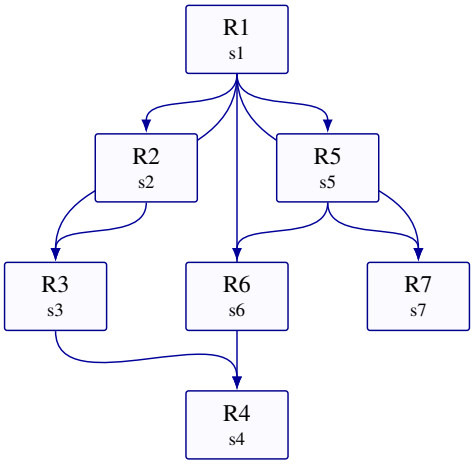
```
One last compatibility issue remains in exception-group matching.  Treat this as a
request from an application maintainer:  solve the concrete problem and leave
unrelated behavior alone.

Please finish the advanced diagnostics and representations.  Greedy pairing may
explain a line such as It matches <item> which was paired with <item>.  When an
exhaustive match exists, append the stable suggestion text; make str() and repr()
deterministic and evaluable where promised.  Keep diagnostics useful for nested
groups without changing matching order or exception propagation.

Before delivery, run the focused repository checks and one integration example.
Report real failures instead of making the feature silently permissive.  Keep the
example small enough to run locally, but make its expected output precise enough for
a reviewer to diagnose a regression.
```

H.1.15 RUST-01

7 stages, 7 requirements, 40 F2P tests, and 26 P2P tests.



Stage 1 instruction

A user-facing gap has been isolated in recipe caching. The surrounding APIs are already present, so please make this first behavior dependable before we add more surface area.

Please add an experimental [cache] recipe attribute behind the existing unstable feature gate. It is valid only on script recipes. The first execution should create a JSON cache entry under .taskcache, a matching invocation should reuse it, and concurrent callers must not corrupt the entry. Recipes without the attribute and builds without the feature gate must behave exactly as before.

Please keep earlier public names and error conventions. Exercise the ordinary case and the nearest invalid or empty case, then run the focused checks so a failure is visible rather than masked. Begin with the smallest fixture a caller would write, then confirm the same result when the feature is used through the public entry point.

Stage 2 instruction

The previous slice of recipe caching is usable in simple cases. We now need to handle the next scenario that appears in production, while keeping the earlier contract intact.

Please extend the current implementation with this next requirement, which revises R1's cache identity without changing its hit and write behavior. Make the cache key include every input that can change recipe behavior: recipe body and name, module path, arguments, interpreter and extension settings, working directory, and exported environment. Keep the key deterministic across processes so a cache entry never crosses unrelated recipes.

When you verify this, include one repeated or reordered invocation; the result should stay deterministic and caller-owned state should remain untouched unless mutation is explicitly part of the API. Use a repeated invocation and a reordered input as a regression case; both should observe the same stable contract.

Stage 3 instruction

We have a concrete report from a project using recipe caching: the edge case below is difficult to diagnose today. Please address it in the existing design instead of introducing a parallel API.

Please add [cache] extra and inputs arguments. Both accept runtime expressions; inputs may be one path or a list resolved relative to the recipe working directory. Include BLAKE3 hashes of regular files, handle symlinks and missing paths explicitly, and let dry-run report what would be read without writing a cache entry.

Do not change unrelated modules or add a fallback that hides malformed input. A focused regression check should demonstrate both the useful result and the behavior at the boundary. Include one malformed, empty, or missing-value case so the established error or no-op behavior is explicit.

Stage 4 instruction

```
The core path in recipe caching now works, but callers still cross this boundary when
they combine features.  This step should make that composition predictable and
observable.

Please add [cache] outputs as one path or a list relative to the recipe working
directory.  A hit is valid only when every declared output still exists; missing or
dangling outputs must force execution.  Output directories should be checked without
hashing their contents, and dry-run must not delete or create anything.

Preserve supported runtime versions and existing output for callers that do not use
this feature.  Check the observable return value, error text, and ordering before
handing it over.  Compare this path with the equivalent operation from the preceding
Step and record any intentional difference in the focused test.
```

Stage 5 instruction

```
We are preparing recipe caching for another release.  Please close the behavior gap
described below, including the failure path, without broad refactoring or silent
compatibility changes.

Please add task_runner --clean.  With no cache directory it should succeed with a
clear message; with one present it should remove only valid cache entries, preserve
unknown files, and report the number removed.  Keep quiet mode, singular/plural
wording, and the exit status useful for scripts.

Please compare the new path with the equivalent old path where one exists.  The
implementation is ready only when the repository checks cover success, failure, and
cleanup without relying on private test selectors.  Exercise two features together,
because the main risk here is a state or metadata leak across the boundary described
above.
```

Stage 6 instruction

```
At this point the implementation needs a careful integration pass.  In recipe caching,
the following behavior is what a real caller sees when state changes mid-operation.

Please let task_runner --clean [RECIPE_PATH] select a recipe or module path using
existing module-path semantics.  Cleaning a recipe removes only its entries; cleaning
a module removes that subtree; an empty selection is a successful no-op.  Invalid
paths should explain what was not found without deleting unrelated cache data.

Keep the change local to the requested behavior and leave a reproducible check in the
existing test entry point.  If an operation is repeated, it must not accumulate stale
state.  Check cleanup and reuse after the operation completes or fails; the next
caller should start from a clean state.
```

Stage 7 instruction

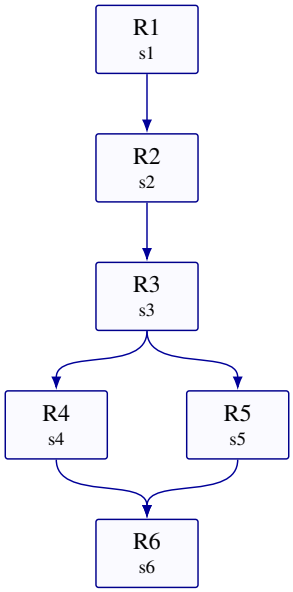
```
One last compatibility issue remains in recipe caching.  Treat this as a request from
an application maintainer:  solve the concrete problem and leave unrelated behavior
alone.

Please extend the current implementation with this next requirement, which revises
R1's cache entry point while keeping the default behavior unchanged.  Add the cache
bypass controls --no-cache and TASK_RUNNER_NO_CACHE. By default the normal cache
behavior remains active; when either switch is enabled, skip both reads and writes
and execute the script.  Verbose mode should say when a cache hit is used, while a
bypass should be distinguishable from a miss without exposing internal hashes.

Before delivery, run the focused repository checks and one integration example.
Report real failures instead of making the feature silently permissive.  Keep the
example small enough to run locally, but make its expected output precise enough for
a reviewer to diagnose a regression.
```

H.1.16 TYPESCRIPT-01

6 stages, 6 requirements, 82 F2P tests, and 643 P2P tests.



Stage 1 instruction

A user-facing gap has been isolated in terminal autocomplete. The surrounding APIs are already present, so please make this first behavior dependable before we add more surface area.

The AutocompletePrompt class currently supports static arrays and synchronous option functions, but it cannot resolve options asynchronously for search-as-you-type workflows.

Extend autocomplete options to support asynchronous resolvers without changing the existing behavior of static arrays or synchronous functions. For this first increment, pass the current search string to the async resolver, expose a loading state while the request is in flight, update the filtered options when the result resolves, and preserve normal navigation, selection, and submission.

Further lifecycle and advanced behavior will be specified in later turns.

Please keep earlier public names and error conventions. Exercise the ordinary case and the nearest invalid or empty case, then run the focused checks so a failure is visible rather than masked. Begin with the smallest fixture a caller would write, then confirm the same result when the feature is used through the public entry point.

Stage 2 instruction

The previous slice of terminal autocomplete is usable in simple cases. We now need to handle the next scenario that appears in production, while keeping the earlier contract intact.

Detect asynchronous option functions by behavior, not by their constructor, prototype, or declared parameter count: invoke the function and treat its return value as asynchronous when it is thenable, meaning it has a .then method. This must also work for zero-parameter async functions. The detection invocation is the first fetch and must not be discarded or repeated solely for detection.

Call the resolver with the current search string and a second argument shaped as { signal: AbortSignal }. Keep loading true while the fetch is in flight. Re-render when async state changes only after the prompt is active; construction itself must not trigger renders.

When you verify this, include one repeated or reordered invocation; the result should stay deterministic and caller-owned state should remain untouched unless mutation is explicitly part of the API. Use a repeated invocation and a reordered input as a regression case; both should observe the same stable contract.

Stage 3 instruction

We have a concrete report from a project using terminal autocomplete: the edge case below is difficult to diagnose today. Please address it in the existing design instead of introducing a parallel API.

Add robust request lifecycle handling. Starting a fetch must abort the previous fetch's signal, and only the latest fetch may update prompt state. A non-SWR cache hit or entering the searchTooShort state must also abort any in-flight signal and invalidate its pending result.

Silently ignore errors whose name is AbortError: set loading to false, return, and do not set loadError. Convert other failures to a string in loadError; a later successful fetch must clear that error.

Debounce fetches using configurable debounceMs. When omitted, use a sensible default between 100 and 300 milliseconds. Rapid input changes should result in only the latest debounced fetch.

Do not change unrelated modules or add a fallback that hides malformed input. A focused regression check should demonstrate both the useful result and the behavior at the boundary. Include one malformed, empty, or missing-value case so the established error or no-op behavior is explicit.

Stage 4 instruction

The core path in terminal autocomplete now works, but callers still cross this boundary when they combine features. This step should make that composition predictable and observable.

Add optional result caching. When cacheResults is enabled, successful results should avoid redundant fetches. Support maxCacheSize with oldest-entry eviction and expose clearCache() to empty the cache. A normal, non-SWR cache hit should use the cached result without refetching and must invalidate any older in-flight request.

Support staleWhileRevalidate, which requires cacheResults: serve cached results immediately, then refetch in the background. Keep loading true during background revalidation, and update both the cache and visible options when the current revalidation completes.

Support minSearchLength. For non-empty input shorter than the threshold, do not fetch, clear filteredOptions, set searchTooShort to true, and invalidate any in-flight result. Empty input must always fetch regardless of the threshold. Clear searchTooShort once the input reaches the threshold.

Preserve supported runtime versions and existing output for callers that do not use this feature. Check the observable return value, error text, and ordering before handing it over. Compare this path with the equivalent operation from the preceding Step and record any intentional difference in the focused test.

Stage 5 instruction

We are preparing terminal autocomplete for another release. Please close the behavior gap described below, including the failure path, without broad refactoring or silent compatibility changes.

Add retry support using maxRetries and retryDelay. Keep loading true between attempts and expose the current attempt through retryCount. A new fetch must cancel any pending retry. Support retryBackoff: default to linear, which uses the same delay for every retry; exponential doubles the base delay for each successive retry.

After all retries are exhausted, set loadError. If fallbackOptions is provided, show those options in filteredOptions; otherwise leave filteredOptions empty. Do not show fallback options after a successful fetch.

Add loadingMinDuration, defaulting to 0. Measure it from the start of the fetch, keep loading true, and defer applying a fast result until that duration has elapsed. Do not add further delay when the fetch itself takes longer. A new fetch must cancel any pending minimum-duration timer.

Please compare the new path with the equivalent old path where one exists. The implementation is ready only when the repository checks cover success, failure, and cleanup without relying on private test selectors. Exercise two features together, because the main risk here is a state or metadata leak across the boundary described above.

Stage 6 instruction

```
At this point the implementation needs a careful integration pass.  In terminal
autocomplete, the following behavior is what a real caller sees when state changes
mid-operation.

Complete the lifecycle cleanup and public wrappers.  On submit, cancel, or close,
abort in-flight fetches, clear debounce, retry, and minimum-duration timers, and
reset all transient async state:  loading, loadError, searchTooShort, and retryCount.

Both autocomplete and autocompleteMultiselect must pass all async options through to
the core prompt:  debounceMs, cacheResults, maxCacheSize, minSearchLength, maxRetries,
retryDelay, retryBackoff, staleWhileRevalidate, fallbackOptions, and
loadingMinDuration.

When the search is too short, display the equivalent of Type at least N characters.
Honor custom loadingMessage and noResultsMessage values, while preserving existing
static, synchronous, validation, cancellation, and multi-select behavior.

Keep the change local to the requested behavior and leave a reproducible check in the
existing test entry point.  If an operation is repeated, it must not accumulate stale
state.  Check cleanup and reuse after the operation completes or fails; the next
caller should start from a clean state.
```