CodeInFlight: Can LLMs Code Well under Changing User Intentions?

Anonymous authors
Paper under double-blind review

Abstract

LLM-based coding agents increasingly work alongside users who revise their requests before implementation is complete. Such revisions can invalidate an agent's current plan or edits, requiring it to redirect unfinished work while preserving progress that remains useful. However, existing repository-level benchmarks largely assume fixed specifications or introduce new requirements only between completed turns, and therefore do not directly evaluate adaptation to changes that arrive during execution. To address this gap, we introduce CodeInFlight, a dataset and repository-level benchmark for evaluating how coding agents adapt to changing user requests while they work. We curate 1,551 execution-validated source tasks spanning 534 repositories and ten programming languages, from which we derive a stratified evaluation set of 287 interaction instances covering requirement changes, cancellation with replacement, and additional tasks. For each instance, UPFRONT, SEQUENTIAL, and IN-FLIGHT reveal the final request before coding, after the initial attempt, or during execution, respectively. Comparing each of 19 configurations with itself, we find that all six configurations resolving at least half of the instances when a change follows their first attempt resolve fewer when it arrives during execution. They generate 11% fewer output tokens but resolve 3.9 points fewer instances, for both withdrawals and additions. In paired runs of eight open-weight models, thinking raises resolution under both deliveries by similar amounts. CodeInFlight thus indicates that these higher-performing configurations adapt less reliably to changes during execution than to changes after completion.

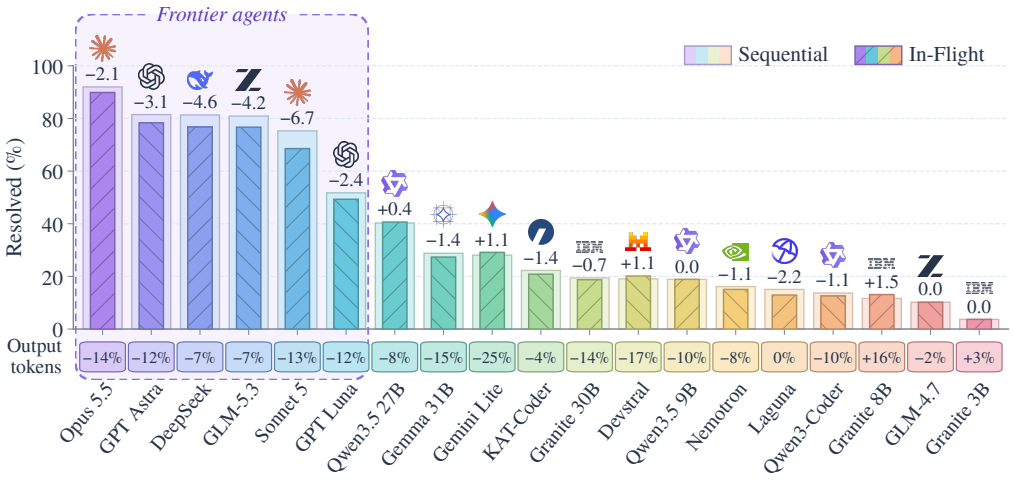


Figure 1: **Resolved rates and output-token changes under SEQUENTIAL and IN-FLIGHT delivery.** Pale and hatched bars show Resolved under SEQUENTIAL and IN-FLIGHT, respectively, with configurations ordered by SEQUENTIAL Resolved. Labels give the IN-FLIGHT-minus-SEQUENTIAL change in Resolved (percentage points) above each pair and in output tokens (percent) below. All six frontier configurations, which resolve at least 50% under SEQUENTIAL (outlined), resolve fewer instances under IN-FLIGHT while generating fewer tokens. Across all 19, Resolved falls for 12 and output tokens fall for 16.

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.

1 INTRODUCTION

LLM-based coding agents are moving from one-shot code generation to sustained collaboration: they implement features, resolve issues, and modify software repositories while a user watches and steers (Yang et al., 2024; Wang et al., 2025). In such sessions, requirements often continue to emerge after the first edit (Jiang et al., 2026). A revised request may invalidate an existing plan, make earlier edits unnecessary, or introduce new work without removing the original obligation. Whether an agent can redirect ongoing implementation while preserving the work that still matters is therefore central to successful collaboration, yet current repository-level evaluations do not measure it. Consider the gallery example in Fig. A.1 (Section A): a user requests a grid layout with upload and preview, then asks for a list while the agent is still implementing the grid. The agent should neither start over nor press on, but keep the finished upload, revise the unfinished layout, and still build the preview.

Existing benchmarks provide important foundations for studying this problem, yet each stops short of it. SWE-bench and FeatureBench evaluate issue resolution and feature development with specifications fixed before execution (Jimenez et al., 2024; Zhou et al., 2026). Interactive benchmarks extend evaluation to clarification, progressively revealed requirements, and feedback across turns (King & Flanigan, 2026; Raghavendra et al., 2026; Wu et al., 2026b), and SWE-Touch studies concurrent changes to the shared codebase (Tan et al., 2026). None of them compares the *same* request change delivered before coding, after a completed attempt, and during execution (Table 1). This limitation raises a critical question: *Can coding agents reliably adapt to changing user intentions during execution, and how does their performance depend on the type and timing of the change?*

To answer this question, we introduce CodeInFlight, a unified dataset and repository-level benchmark for coding under changing user intentions. We curate 1,551 execution-validated source tasks spanning 534 upstream repositories and ten programming languages, and derive from them a stratified core of 287 interaction instances covering *requirement changes*, *cancellation with replacement*, and *additional tasks* (Fig. 2). Each instance specifies an initial request, a subsequent change, and executable final requirements that distinguish what remains active from what has been withdrawn. We then evaluate the same final requirements under three delivery conditions (Fig. 3): UPFRONT provides the final request before coding begins, SEQUENTIAL delivers the change after the initial attempt, and IN-FLIGHT introduces it while execution is underway (Section 3.3). Holding the final requirements fixed lets us ask how success depends on when the change arrives.

Our experiments compare each of 19 configurations with itself (Section 4) and lead to one central finding. **Frontier agents do not yet handle a change during execution as well as a change after completion.** The six configurations that resolve at least 50% under SEQUENTIAL generate 11% fewer output tokens under IN-FLIGHT yet resolve 3.9 points fewer, whereas the other 13 change by -0.3 points (Fig. 1). Three further findings qualify this result. **First,** output shifts from before the change to after it, and the size of the token saving does not track the change in accuracy. **Second,** the loss takes a different form for each change type. Cancellations save the most tokens yet lose 5.3 points of Resolved, whereas additional tasks change token use little but lose both complete and partial correctness. **Third,** in paired runs of eight open-weight models, thinking raises IN-FLIGHT and SEQUENTIAL Resolved by similar amounts, and no model's gap changes significantly (Section 4.5).

In summary, our contributions are threefold:

- **Problem Identification.** We identify a gap in repository-level coding-agent evaluation: existing benchmarks do not assess how agents adapt when users change their requests during execution.

- **Unified Benchmark and Dataset.** We construct CodeInFlight from 1,551 execution-validated source tasks, with 287 evaluation instances covering three request-change types, and evaluate the same final requirements under UPFRONT, SEQUENTIAL, and IN-FLIGHT delivery.

- **Comprehensive Experiments and Empirical Insights.** Across 19 configurations, we show that frontier agents resolve fewer instances when a change arrives during execution despite spending fewer tokens, and that the loss takes a different form for each change type.

Table 1: **Evaluation of changing requests in agent benchmarks.** The final columns distinguish revised goals, intervention during execution, and controlled arrival. Marks indicate design support: ✓ present, ○ partial, and – absent.

Benchmark	Intervention	SETTING			REQUEST DYNAMICS		
		Repo-level	Exec. tests	User in the loop	Intent changes	Mid-execution	Controlled timing
FIXED-SPECIFICATION CODING							
SWE-bench † (2024)	None	✓	✓	–	–	–	–
FeatureBench (2026)	None	✓	✓	–	–	–	–
RoadmapBench (2026)	None	✓	✓	–	–	–	–
INTERACTIVE CODING							
Dialogue SWE-Bench (2026)	Clarification	✓	✓	✓	–	–	–
SWE-Interact (2026)	Feedback	✓	○	✓	○	–	–
SWE-chat (2026)	User requests	✓	–	✓	○	○	–
MULTI-STAGE CODING							
SWE-Chain (2026)	Chained tasks	✓	✓	–	–	–	–
EvoCode-Bench (2026)	Requirements	○	✓	○	✓	–	–
RUNTIME INTERVENTIONS AND GOAL REVISION							
SWE-Touch (2026)	Code	✓	✓	○	–	✓	○
Handoff Debt (2026)	Context handoff	✓	✓	–	–	✓	○
InterruptBench (2026)	Goal revision	–	–	✓	✓	✓	✓
HAS-Bench (2026a)	Goal revision	○	○	✓	✓	–	–
CodeInFlight *(ours)*	**Goal revision**	✓	✓	✓	✓	✓	✓

✓ holds by design ○ partial, or on a subset – not present

† Cells refer to the human-validated 500-instance VERIFIED subset (Chowdhury et al., 2024).

2 RELATED WORK

CodeInFlight connects coding evaluation with interactive agent benchmarks spanning general, web, and operating-system environments (Liu et al., 2024b; Zhou et al., 2024; Xie et al., 2024). We review three lines of work in turn, and Section B develops the broader context.

Repository-Level Coding. Executable evaluation extends from functions (Chen et al., 2021; Austin et al., 2021) to library use (Zhuo et al., 2025) and cross-file completion (Liu et al., 2024a; Ding et al., 2023). SWE-bench and its Verified subset evaluate issue resolution in real repositories (Jimenez et al., 2024; Chowdhury et al., 2024), and Multi-SWE-Bench, SWE-PolyBench, and SWE-rebench V2 broaden language and repository coverage (Zan et al., 2025; Rashid et al., 2025; Badertdinov et al., 2026). FeatureBench addresses feature implementation, while RoadmapBench, SWE-Evo, and SWE-Chain examine longer software changes (Zhou et al., 2026; Xu et al., 2026; Le et al., 2025; Lam et al., 2026). Agent loops and localization-and-repair pipelines are both evaluated in this fixed-specification setting (Xia et al., 2024; Zhang et al., 2024). CodeInFlight uses fixed-specification performance as its UPFRONT reference and varies when the revision arrives.

Interactive Coding. Interactive benchmarks then introduce information exchange into coding evaluation. Dialogue SWE-Bench and Ambig-SWE study clarification, while SWE-Interact's simulated user supplies further constraints and feedback (King & Flanigan, 2026; Vijayvargiya et al., 2026; Raghavendra et al., 2026). SWE-chat documents real coding conversations and SWE-Together reconstructs multi-turn sessions (Baumann et al., 2026; Wu et al., 2026b), and Slop-CodeBench and EvoCode-Bench let specifications evolve across rounds (Orlanski et al., 2026; Shen et al., 2026). CodeInFlight adds controlled delivery of changes during unfinished trajectories.

Mid-Execution Adaptation. Finally, a third line of work intervenes while the work is still underway. SWE-Touch introduces concurrent code changes, Handoff Debt studies transfers of unfinished work, and Gaia2 evaluates responses to asynchronous events (Tan et al., 2026; KC & Budathoki, 2026; Froger et al., 2026). Beyond coding, InterruptBench, DeskCraft, Full-Duplex-Bench-v3, and HAS-Bench examine interruptions and changing requests in web, desktop, voice, and mixed human–

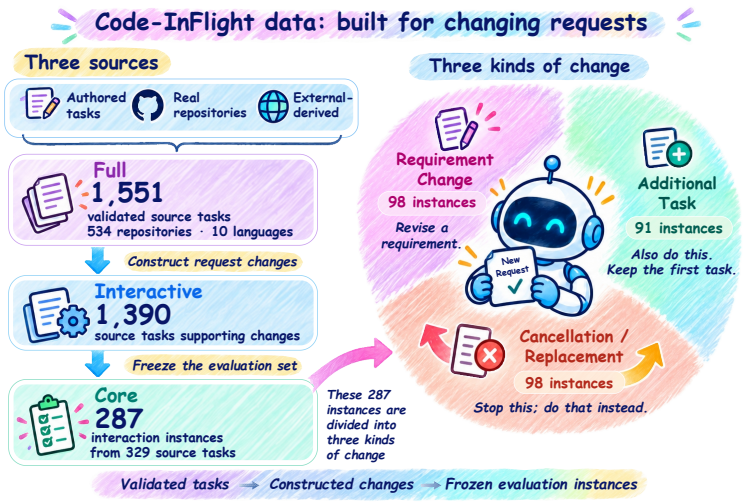


Figure 2: **The CodeInFlight dataset and its interaction coverage.** Three sources produce CodeInFlight-Full, eligibility defines CodeInFlight-Interactive, and stratified sampling fixes CodeInFlight-Core (287 interactions from 329 tasks). Repository counts exclude authored projects.

agent settings (Zou et al., 2026; Wang et al., 2026; Lin et al., 2026; Wu et al., 2026a). CodeInFlight brings controlled changes of user intent to persistent repository work, where the final implementation is checked against the same executable requirements under three delivery conditions.

3 Benchmark and Dataset

CodeInFlight connects execution-validated coding tasks to controlled changes in user requests. We first construct the dataset, then define the interactions and the three delivery conditions.

3.1 Dataset Construction

Figure 2 summarizes the dataset's sources and three released layers. *Controlled* tasks are authored mini-projects with individually tested targets. *Real-repository* tasks are derived from upstream commits, with requirements established by executing the associated tests. *External-derived* tasks are re-validated instances from public coding corpora (Badertdinov et al., 2026; Zan et al., 2025). These sources combine precise task structure with the code and test environments of existing repositories.

Validation then runs each task in its own environment with the reference patch withheld and applied. Tasks carry an evidence tier: *tier 1* requires every designated fail-to-pass test to be observed failing before the patch and passing afterward. In contrast, *tier 2* observes the post-patch pass but infers pre-patch failure from an aggregate upstream record or an unbuildable base state. CodeInFlight-Full contains 1,551 validated source tasks (1,101 tier 1 and 450 tier 2). CodeInFlight-Interactive contains the 1,390 tasks whose validated structure supports a request change. CodeInFlight-Core freezes 287 interaction instances over 329 source tasks, stratified by change type and source, with language coverage within each cell and a repository cap. An unrelated-task instance consumes two source tasks, explaining the distinction between tasks and instances. Section D gives validation and composition details.

3.2 Interaction Construction

To turn a validated task into an interaction, we specify what the user asks first, what changes, and what must finally hold. Each instance fixes an opening request, a subsequent change, and a final set of required behavior. The change is derived from the source task's acceptance targets and associated material, such as an issue, commit message, or authored subgoal. It takes one of three forms:

- **Requirement change:** add acceptance criteria to the current work, or replace a requirement while retaining the remaining obligations.

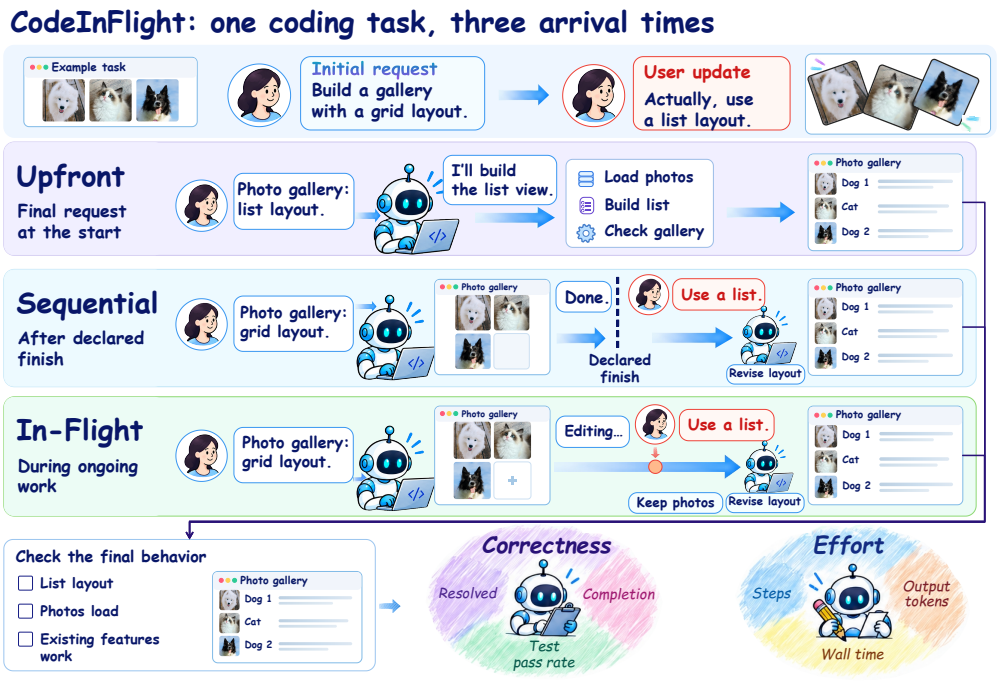


Figure 3: **Three delivery conditions for the same final request.** The revised request arrives before coding, after the initial attempt, or during ongoing work, and the same final tests assess correctness.

- **Cancellation/replacement:** withdraw the initial work and request another part of the same task.
- **Additional task:** add related work in the same repository, or an independent task in a second workspace. Both remain required.

In every case, the construction determines the final required tests and regression tests before any run. Withdrawn requirements never enter the pass-rate denominator. Every condition uses the same final requirements, task environments, and scoring procedure. Section E specifies the five construction families and shows representative messages.

3.3 Three Delivery Conditions

Having fixed the task and the change, we finally vary when the change is delivered. The conditions in Fig. 3 differ in when the agent receives the same final request:

- **Upfront:** the opening message states the final request. There is no later change.
- **Sequential:** the change is delivered after the first declared finish, with a fallback at 90% of the step budget if the agent has not declared completion.
- **In-Flight:** the change is delivered during execution. The arrival point is a benchmark parameter. Here it is half the first-finish step in the same model's frozen Sequential calibration run on that instance, rounded half up. When that calibration is unavailable, delivery occurs at half the step budget.

With a budget of 250 steps, the fallback points are 225 for Sequential and 125 for In-Flight. The run records distinguish calibrated and budget-based arrivals, including episodes that finish before the intended trigger. This makes delivery interpretable for models with different execution speeds while retaining fixed-budget episodes in the evaluation (Section F.2).

4 Experiments

We compare each configuration with itself and, after describing the setup, answer four questions.

- **RQ1: Accuracy.** How does a change during execution affect the tasks each agent resolves?
- **RQ2: Efficiency.** How does token use change, and does it track the change in accuracy?
- **RQ3: Change Types.** How does the timing effect differ across types of change?
- **RQ4: Reasoning.** Does extended thinking narrow the timing gap measured in RQ1?

4.1 Experimental Setup

We evaluate 19 configurations, comprising 12 locally served open-weight configurations and seven API configurations, on the frozen Core under all three delivery conditions. A shared harness provides repository inspection, editing, testing, and completion tools. Each configuration retains its serving and generation settings across conditions. The main runs allow 250 action steps with a 275-call guard. Model settings and arrival rules appear in Sections F.2 and F.3. Local configurations run without thinking where a switch exists, DeepSeek-V4-Flash runs with thinking OFF, and the other API configurations keep the reasoning settings listed in Table F.2.

Correctness and execution resources. Resolved requires all final required and valid regression tests to pass. Macro test pass rate (TPR) averages required-test pass rates equally across scored instances, excluding withdrawn requirements and regression tests from its denominator. These measures distinguish partial implementation from a complete solution, while steps and output tokens measure consumption (Appendix F).

Paired comparisons. The primary timing contrast is In-Flight minus Sequential on the same instances within each model, with each paired instance weighted equally. We report pointwise 95% source-task-cluster bootstrap intervals and retain model-caused failures as outcomes. Unknown verdicts are not replaced with zero. The output-token change is the ratio of mean output tokens under In-Flight to those under Sequential on the same instances, pooled as a geometric mean with configurations weighted equally. Budget and no-change-message controls examine the remaining

Table 2: **In-Flight (IF) minus Sequential (Seq) within each model.** Each model is compared on the 273 to 286 instances scored with a recorded trajectory under both conditions, and Up gives Upfront Resolved on the 270 to 286 shared with IF. Δ is IF minus Seq in points for Resolved and TPR, in percent for output tokens and in steps. Average weights the 19 models equally (geometric mean for tokens), and mint or rose fills mark where IF does better or worse.

Model	Resolved (%)				TPR (pp)	Tokens (%)	Steps
	Up	Seq	IF	Δ	Δ	Δ	Δ
Open-weight models							
Granite-4.2-3B	4.8	3.7	3.7	0.0	−1.3	+3	−0.2
GLM-4.7-Flash	10.6	10.2	10.2	0.0	−2.4	−2	+1.0
Granite-4.2-8B	13.7	11.6	13.1	+1.5	+0.9	+16	+0.9
Qwen3-Coder-30B	13.5	13.7	12.6	−1.1	−1.2	−10	−11.6
Laguna-XS	12.5	15.1	12.9	−2.2	+0.9	0	−0.2
Nemotron-3.5	18.7	16.1	15.1	−1.1	−0.4	−8	−10.4
Qwen3.5-9B	14.4	18.9	18.9	0.0	+0.2	−10	−0.1
Devstral-Small-2	23.8	19.1	20.1	+1.1	+1.2	−17	−39.9
Granite-4.2-30B	16.8	19.4	18.7	−0.7	−1.4	−14	−6.7
KAT-Coder	20.5	22.3	20.8	−1.4	−1.6	−4	+2.6
Gemma-4-31B	30.7	28.8	27.3	−1.4	+1.0	−15	−17.0
Qwen3.5-27B	39.7	40.3	40.6	+0.4	−0.2	−8	−7.2
API models							
Gemini-3.5-Flash-Lite	32.9	28.1	29.1	+1.1	−1.8	−25	−33.6
GPT-5.6-Luna	49.7	51.7	49.3	−2.4	−3.3	−12	−4.4
Claude-Sonnet-5	75.7	75.3	68.6	−6.7	−3.0	−13	−6.7
GLM-5.3-Flash	78.9	80.9	76.7	−4.2	−1.1	−7	−6.3
DeepSeek-V4-Flash	78.2	81.3	76.8	−4.6	−2.2	−7	−6.6
GPT-6-Astra	89.2	81.5	78.3	−3.1	−3.0	−12	−3.7
Claude-Opus-5.5	93.7	92.0	89.9	−2.1	+0.9	−14	−4.2
Average	37.8	37.4	35.9	−1.4	−0.9	−9	−8.1

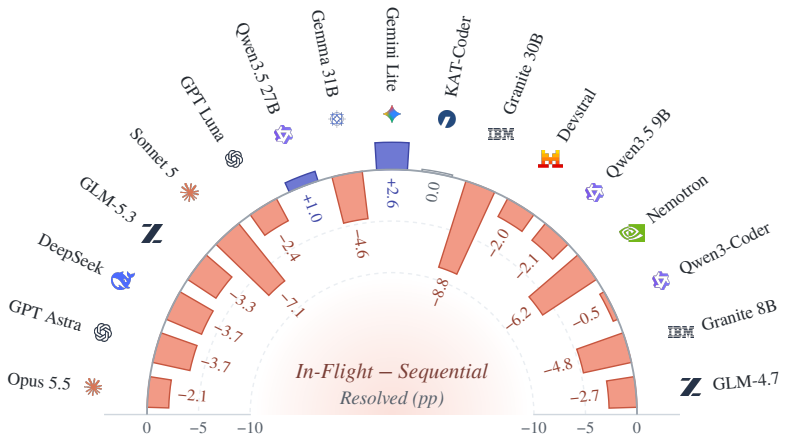


Figure 4: **Change in Resolved on instances with an Upfront completion declaration.** IN-FLIGHT minus SEQUENTIAL Resolved (percentage points) on the paired instances whose UPFRONT run ends with a declared finish, one bar per configuration in SEQUENTIAL Resolved order. Bars rest on 37 to 286 instances. Granite-4.2-3B and Laguna-XS declare completion on fewer than ten instances under UPFRONT and are not shown.

step budget and the extra user message separately, while a matched native-Codex comparison examines the execution framework (Appendices I and J).

4.2 ACCURACY UNDER MID-EXECUTION CHANGES (RQ1)

To answer RQ1, Fig. 1 and Table 2 compare each configuration with itself on paired instances. Twelve of the 19 configurations resolve fewer instances under IN-FLIGHT, and the pooled change is -1.4 percentage points, with a 95% interval from -2.4 to -0.5. The loss concentrates in the six frontier configurations that resolve at least 50% under SEQUENTIAL. All six resolve fewer instances, and together they lose 3.9 points, whereas the other 13 change by -0.3 points. The frontier change is thus 3.6 points lower, with a 95% interval for this difference from 1.5 to 5.7 points. Against UPFRONT, IN-FLIGHT resolves 1.9 points fewer on average, and 14 of the 19 configurations are lower.

The small average change of the other 13 does not mean that they adapt well. They often do not declare a finish, so 61% of their SEQUENTIAL changes arrive at the step-225 fallback, against 3% for the frontier configurations, and IN-FLIGHT then leaves more steps for the change. On instances that a configuration finishes under UPFRONT, IN-FLIGHT resolves 3.0 points fewer than SEQUENTIAL on average, and 14 of the 17 configurations with at least ten such instances are lower (Fig. 4).

Answer to RQ1. Frontier agents resolve fewer instances when a change arrives during execution than when it arrives after they finish. Weaker agents change little on average, and because they seldom declare a finish, their SEQUENTIAL change often arrives at the step-225 fallback.

4.3 TOKEN EFFICIENCY UNDER MID-EXECUTION CHANGES (RQ2)

To answer RQ2, Fig. 5 places each configuration by its changes in output tokens and Resolved. IN-FLIGHT delivery reduces generated output for 16 of 19 configurations, by 9% on average and by 11% for the frontier configurations, and shortens trajectories by 8.1 steps on average (Table 2). Behind these savings, output shifts from before the change to after it. Tokens generated before the change fall to 0.52 times their SEQUENTIAL level, close to the half that the midway arrival implies, whereas tokens generated after it rise to 3.09 times their SEQUENTIAL level (Fig. 5b).

However, lower output does not coincide with higher accuracy. Across the 19 configurations, the rank correlation between the change in tokens and the change in Resolved is 0.03, and 11 of them, including all six frontier ones, fall in the quadrant with fewer tokens and fewer solutions.

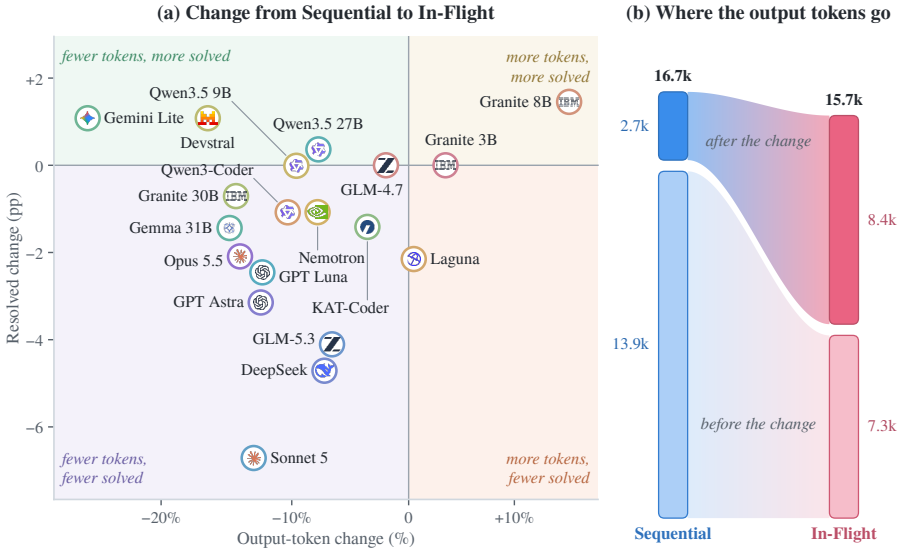


Figure 5: **Where the tokens go and what they buy.** (a) Each ring is one configuration, placed by its change in output tokens (log scale) and in Resolved from SEQUENTIAL to IN-FLIGHT. (b) Mean output tokens per instance before and after the change, averaged over the 19 configurations on instances whose output can be split at the change.

Answer to RQ2. IN-FLIGHT delivery saves about 9% of output tokens, 11% for frontier agents, and moves output from before the change to after it. The savings do not come with more complete solutions.

4.4 PATTERNS ACROSS CHANGE TYPES (RQ3)

To answer RQ3, we split the frontier comparison by change type (Table 3). Requirement changes revise the original work and cancellations withdraw it, whereas additional tasks keep the original obligation and add a new one. Cancellations save the most output, 20%, yet they also show the largest loss in Resolved, 5.3 points. Requirement changes save 9% and lose the least, 2.2 points. Additional tasks leave output nearly unchanged but lose 4.1 points of Resolved and 3.5 points of TPR, so for this type the later delivery costs partial as well as complete correctness. Resolved falls in 15 of the 18 combinations of frontier configuration and change type (Fig. 6), and Section H.2 extends the comparison to all 19 configurations and to the five event families.

Table 3: **Frontier agents by change type.** IN-FLIGHT minus SEQUENTIAL for the six frontier configurations, in points for Resolved and TPR and in percent for output tokens.

Change type	Resolved	TPR	Tokens
Requirement change	−2.2	−0.1	−9%
Cancellation	−5.3	−2.5	−20%
Additional task	−4.1	−3.5	−5%
All types	**−3.9**	**−2.0**	**−11%**

Answer to RQ3. Frontier agents save the most tokens on cancellations, yet they lose 5.3 points of Resolved there. Additional tasks cost both complete and partial correctness while leaving output nearly unchanged.

4.5 THINKING AND THE TIMING GAP (RQ4)

To answer RQ4, we compare thinking OFF and ON for the nine configurations run both ways (OpenAI, 2024; DeepSeek-AI, 2025; Yang et al., 2025). For the eight open-weight models, thinking raises IN-FLIGHT Resolved by 10.1 to 32.7 points (Fig. K.2) and SEQUENTIAL Resolved by a similar amount, with all eight gap-change intervals spanning zero (Fig. 7, Table K.6). DeepSeek-V4-Flash, whose two runs use separately calibrated arrivals, narrows its gap by 6.4 points, whereas the three frontier configurations that reason by default, Claude-Opus-5.5, GPT-6-Astra, and GLM-5.3-Flash, still resolve fewer instances under IN-FLIGHT (Table 2).

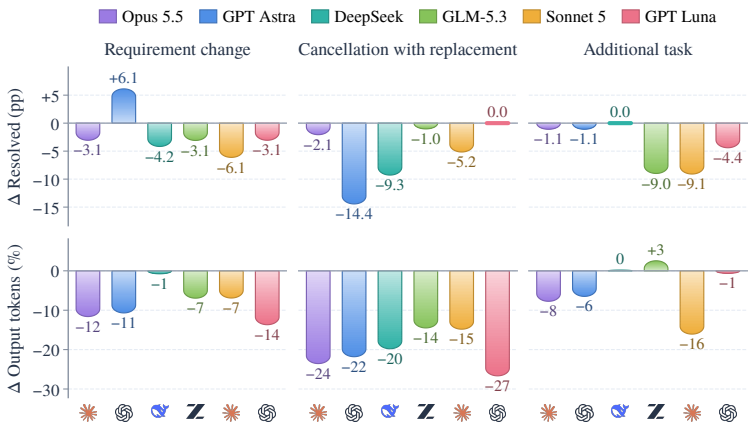


Figure 6: **Frontier agents by change type.** IN-FLIGHT minus SEQUENTIAL Resolved (top, points) and output tokens (bottom, percent) for the six frontier configurations, with all 19 in Fig. H.7.

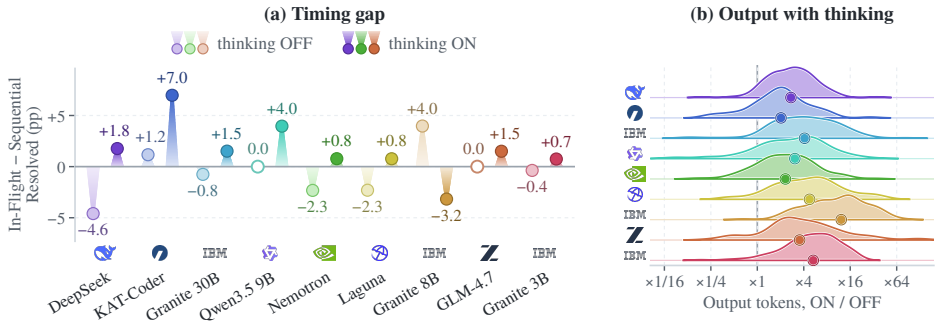


Figure 7: **Thinking and the timing gap.** (a) IN-FLIGHT minus SEQUENTIAL Resolved (percentage points) with thinking OFF (pale drops) and ON (saturated drops). (b) Per-instance ratio of IN-FLIGHT output tokens, ON to OFF, with medians marked. Both panels use the instances observed in all four runs of each configuration, so OFF gaps can differ from Table 2.

Answer to RQ4. Thinking raises accuracy under both deliveries by similar amounts, and frontier configurations that reason by default still lose accuracy under IN-FLIGHT.

5 CONCLUSION

In this paper, we introduced CodeInFlight, a benchmark that evaluates how coding agents adapt when a user changes the request during execution. Comparing each configuration with itself, we found that all six frontier configurations resolve fewer instances when a change arrives during execution than when it follows their first attempt. Their lower token use is not accompanied by higher accuracy, and additional test-time reasoning raises accuracy under both deliveries by similar amounts without reliably narrowing the timing gap.

Limitations. The controlled design of CodeInFlight leaves two aspects of real collaboration outside its current scope. First, each trajectory contains one change delivered as a text message at one controlled point, whereas real sessions can revise a request repeatedly. Second, success is judged by the final tests, which do not capture how an agent explains a change or whether it removes the code written for withdrawn work (Section C).

Future work. These limitations motivate two natural extensions. Delivering several changes within one trajectory at varied arrival points would relax the first and turn the three delivery conditions into a timing curve. Measuring residual withdrawn code and how agents acknowledge a change would address the second by evaluating a revision beyond its final behavior. Our findings further call for agents that track which work a change retains, withdraws, or adds.

Ethics Statement

CodeInFlight is built from public repositories with permissive licenses and from projects authored for the benchmark, and each task records its license and provenance (Section L). The study involves no human subjects, private repositories, or private communications, and requests are assembled from public issue or commit text, authored specifications, and fixed templates. Agents run only inside isolated task environments. The benchmark is intended to support more reliable coding agents, but improvements in following changing requests can also increase the capabilities and dual-use risks of coding agents. We limit evaluation-time risk through isolated environments and release only authored or permissively licensed public materials.

Reproducibility Statement

Section 3 and Section D describe how source tasks are validated and how Core interactions are built, and Sections E.2 and E.3 show the messages agents receive. Section F.2 specifies the harness, tools, step budget, and arrival rule, and Section F.3 lists the serving and generation settings of all 19 configurations. Section F defines the metrics, the pairing, and the source-task-cluster bootstrap with its fixed seeds. The dataset release contains task metadata, request text, test identifiers, reference changes, and environment recipes (Section L), and we will also release the harness and the scripts that produce every table and figure from the recorded runs.

AI Use Statement

During this work, we used LLM-based assistants to help with code development and language polishing. All reported numbers were produced with scripts from the recorded runs. The authors take full responsibility for the content of this paper.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. URL https://arxiv.org/abs/2108.07732.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench V2: Language-agnostic SWE task collection at scale. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2026. doi: 10.48550/arXiv.2602.23866. URL https://arxiv.org/abs/2602.23866.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025. URL https://arxiv.org/abs/2506.07982.

Joachim Baumann, Vishakh Padmakumar, Xiang Li, John Yang, Diyi Yang, and Sanmi Koyejo. SWE-chat: Coding agent interactions from real users in the wild. *arXiv preprint arXiv:2604.20779*, 2026. URL https://arxiv.org/abs/2604.20779.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench Verified. OpenAI, August 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. URL https://arxiv.org/abs/2501.12948.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2023. URL https://arxiv.org/abs/2310.11248.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, Kunal Malkan, Dheeraj Mekala, Pierre Ménard, Gerard Moreno-Torres Bertran, Ulyana Piterbarg, Mikhail Plekhanov, Mathieu Rita, Andrey Rusakov, Vladislav Vorotilov, Mengjue Wang, Ian Yu, Amine Benhalloum, Grégoire Mialon, and Thomas Scialom. Gaia2: Benchmarking LLM agents on dynamic and asynchronous environments. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2602.11964.

Chengqian Gao, Haonan Li, Liu Liu, Zeke Xie, Peilin Zhao, and Zhiqiang Xu. Principled data selection for alignment: The hidden risks of difficult examples. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2025. URL https://arxiv.org/abs/2502.09650.

Bowen Jiang, Haowei Cheng, Yuhong Fu, Anne Koziolek, Jialong Li, and Weixing Zhang. Requirements after the first edit: Mining late requirement emergence and rework in real-world coding-agent sessions. *arXiv preprint arXiv:2609.03028*, 2026. URL https://arxiv.org/abs/2609.03028.

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 (ICLR)*, 2024. URL https://arxiv.org/abs/2310.06770.

Dipesh KC and Anjila Budathoki. Handoff debt: The rediscovery cost when coding agents take over interrupted tasks. *arXiv preprint arXiv:2606.02875*, 2026. URL https://arxiv.org/abs/2606.02875.

Brendan King and Jeffrey Flanigan. Dialogue SWE-Bench: A benchmark for dialogue-driven coding agents. *arXiv preprint arXiv:2606.13995*, 2026. URL https://arxiv.org/abs/2606.13995.

Man Ho Lam, Chaozheng Wang, Hange Liu, Jingyu Xiao, Haau-sing Li, Jen-tse Huang, Terry Yue Zhuo, and Michael R. Lyu. SWE-Chain: Benchmarking coding agents on chained release-level package upgrades. *arXiv preprint arXiv:2605.14415*, 2026. URL https://arxiv.org/abs/2605.14415.

Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. SWE-EVO: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025. URL https://arxiv.org/abs/2512.18470.

Guan-Ting Lin, Chen Chen, Zhehuai Chen, and Hung-yi Lee. Full-Duplex-Bench-v3: Benchmarking tool use for full-duplex voice agents under real-world disfluency. *arXiv preprint arXiv:2604.04847*, 2026. URL https://arxiv.org/abs/2604.04847.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations (ICLR)*, 2024a. URL https://arxiv.org/abs/2306.03091.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations (ICLR)*, 2024b. URL https://arxiv.org/abs/2308.03688.

OpenAI. OpenAI o1 system card. *arXiv preprint arXiv:2412.16720*, 2024. URL https://arxiv.org/abs/2412.16720.

Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. SlopCodeBench: Benchmarking how coding agents degrade over long-horizon iterative tasks. *arXiv preprint arXiv:2603.24755*, 2026. URL https://arxiv.org/abs/2603.24755.

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. URL https://arxiv.org/abs/2606.30573.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. SWE-PolyBench: A multi-language benchmark for repository level evaluation of coding agents. *arXiv preprint arXiv:2504.08703*, 2025. URL https://arxiv.org/abs/2504.08703.

Haiyang Shen, Xuanzhong Chen, Wendong Xu, Yun Ma, Liang Chen, and Kuan Li. EvoCodeBench: Evaluating coding agents in multi-turn iterative interactions. *arXiv preprint arXiv:2605.24110*, 2026. URL https://arxiv.org/abs/2605.24110.

Yuqiao Tan, Jinxiang Meng, Fangyu Lei, Minzheng Wang, Shizhu He, Jun Zhao, and Kang Liu. SWE-Touch: Benchmarking coding agents when users touch the code. *arXiv preprint arXiv:2608.02499*, 2026. URL https://arxiv.org/abs/2608.02499.

Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. AmbigSWE: Interactive agents to overcome underspecificity in software engineering. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2502.13069.

Wenkai Wang, Tao Xiong, Jingchen Ni, Yunpeng Bao, Xiyun Li, Tianqi Liu, Hongcan Guo, Zilong Huang, and Shengyu Zhang. DeskCraft: Benchmarking desktop agents on professional workflows and human-in-the-loop collaboration. *arXiv preprint arXiv:2606.03103*, 2026. doi: 10.48550/arXiv.2606.03103. URL https://arxiv.org/abs/2606.03103.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2407.16741.

Yaozu Wu, Wei-Chieh Huang, Jizhou Guo, Dongyuan Li, Renhe Jiang, Henry Peng Zou, Chunyu Miao, Shanghao Li, Weizhi Zhang, WeiWei Ye, Yankai Chen, Meng Zhang, Xue Liu, and Philip S. Yu. HAS-Bench: Evaluating LLM-based human-agent systems under configurable human participation. *arXiv preprint arXiv:2607.04329*, 2026a. URL https://arxiv.org/abs/2607.04329.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, and Shengzhi Li. SWE-Together: Evaluating coding agents in interactive user sessions. *arXiv preprint arXiv:2606.29957*, 2026b. URL https://arxiv.org/abs/2606.29957.

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

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2024. URL https://arxiv.org/abs/2404.07972.

Xinbo Xu, Ruihan Yang, Haiyang Shen, Wendong Xu, Bofei Gao, Ruoyu Wu, Kean Shi, Weichu Xie, Xuanzhong Chen, Ming Wu, Jason Zeng, Michael Heinrich, Elvis Zhang, Liang Chen, Kuan Li, and Baobao Chang. RoadmapBench: Evaluating long-horizon agentic software development across version upgrades. *arXiv preprint arXiv:2605.15846*, 2026. URL https://arxiv.org/abs/2605.15846.

An Yang, Anfeng Li, Baosong Yang, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2405.15793.

Shuo Yang, Zeke Xie, Hanyu Peng, Min Xu, Mingming Sun, and Ping Li. Dataset pruning: Reducing training data by examining generalization influence. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://arxiv.org/abs/2205.09329.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2406.12045.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Ming Ding, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-2111. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/5afa9cb1e917b898ad418216dc726fbd-Abstract-Datasets_and_Benchmarks_Track.html. Datasets and Benchmarks Track.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 1592–1604, 2024. doi: 10.1145/3650212.3680384. URL https://arxiv.org/abs/2404.05427.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. FeatureBench: Benchmarking agentic coding for complex feature development. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2602.10975.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2307.13854.

Yifei Zhou, Song Jiang, Yuandong Tian, Jason Weston, Sergey Levine, Sainbayar Sukhbaatar, and Xian Li. SWEET-RL: Training multi-turn LLM agents on collaborative reasoning tasks. *arXiv preprint arXiv:2503.15478*, 2025. URL https://arxiv.org/abs/2503.15478.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2406.15877.

Henry Peng Zou, Chunyu Miao, Wei-Chieh Huang, Yankai Chen, Yue Zhou, Hanrong Zhang, Yaozu Wu, Liancheng Fang, Zhengyao Gu, Zhen Zhang, Kening Zheng, Fangxin Wang, Yi Nian, Shanghao Li, Wenzhe Fan, Langzhou He, Weizhi Zhang, Xue Liu, and Philip S. Yu. When users change their mind: Evaluating interruptible agents in long-horizon web navigation. *arXiv preprint arXiv:2604.00892*, 2026. URL https://arxiv.org/abs/2604.00892.

Appendix

Table of Contents

A Motivation **15**

B Extended Related Work **17**
B.1 Repository-Level Coding . . . 17
B.2 Interactive Coding . . . 17
B.3 Mid-Execution Adaptation . . . 17

C Limitations and Future Work **19**
C.1 Scope of the benchmark . . . 19
C.2 Future directions . . . 19

D Dataset Details **21**
D.1 Sources and validation . . . 21
D.2 Evaluation subset . . . 21
D.3 Task composition and test-suite scale . . . 22
D.4 Language coverage and interaction structure . . . 23

E Interaction Construction **26**
E.1 Targets and final requirements . . . 26
E.2 Requests and acceptance tests . . . 26
E.3 Examples from the dataset . . . 27

F Evaluation Details **30**
F.1 Correctness and request completion . . . 30
F.2 Arrival conditions and execution settings . . . 30
F.3 Model configurations . . . 31
F.4 Paired comparisons and uncertainty . . . 31

G Prompts **33**

H Additional Results **35**
H.1 Correctness and execution resources . . . 36
H.2 Variation across request-change types . . . 43
H.3 Variation across model families and codebases . . . 50
H.4 Paired changes and solved-instance overlap . . . 52
H.5 Taking up and completing an unrelated task . . . 55

I Budget and Message Controls **56**
I.1 Remaining budget and use of additional actions . . . 56
I.2 No-change message control . . . 57

J Comparison with Native Codex **59**

K Paired Thinking-Mode Experiments **60**
K.1 Frozen configurations and paired evaluation . . . 60
K.2 Correctness and cost across all delivery conditions . . . 64
K.3 Instance-level resource changes and trajectory endings . . . 66
K.4 Outcome transitions and remaining request-type difficulty . . . 66
K.5 Separating general gains from adaptation-specific gains . . . 69
K.6 Coverage, inclusion, and missingness sensitivities . . . 70

L Data and Licensing **73**

A MOTIVATION

This section explains the motivation behind the design of CodeInFlight. It argues that a change arriving while a coding agent is still working poses a problem of its own, that the timing of the change therefore belongs in the evaluation, and that studying it required a dataset that did not yet exist.

Coding agents increasingly work alongside their users rather than in isolation, implementing features and repairing repositories while the user follows the progress (Yang et al., 2024; Wang et al., 2025). In this setting, users often refine what they want as the work unfolds, and studies of real coding sessions find that requirements continue to emerge after the agent's first edit (Jiang et al., 2026; Baumann et al., 2026). A change therefore rarely meets a blank slate. Instead, it reaches a repository that the agent has already begun to transform, in which some work is finished, some is half done, and some has not yet started. Whether the agent can carry such a change through to a complete implementation is thus a question about collaboration as much as about coding.

Figure A.1 makes this situation concrete with a small gallery application. The user first asks for a grid layout with image upload and preview, and while the agent is building the grid, the user asks for a list instead. At that moment, upload already works, the grid is partly built, and preview has not been started, so a correct response keeps the working upload, redirects the layout edits toward the list, and still delivers the preview. However, each obvious shortcut fails in its own way. Continuing the original plan delivers the wrong layout, rebuilding from scratch endangers the upload that already works, and following only the latest message forgets the preview that is still owed. The difficulty therefore lies less in writing new code than in reconciling the conversation with the current state of the code, that is, in deciding which obligations survive, which edits must change, and which work is still owed. Moreover, the kind of change determines which of these decisions matters most. A requirement change keeps the current work but alters what it must satisfy, a cancellation withdraws work that is still in progress, and an additional task keeps the original obligation while adding another. These three relations define the change types of CodeInFlight (Section E).

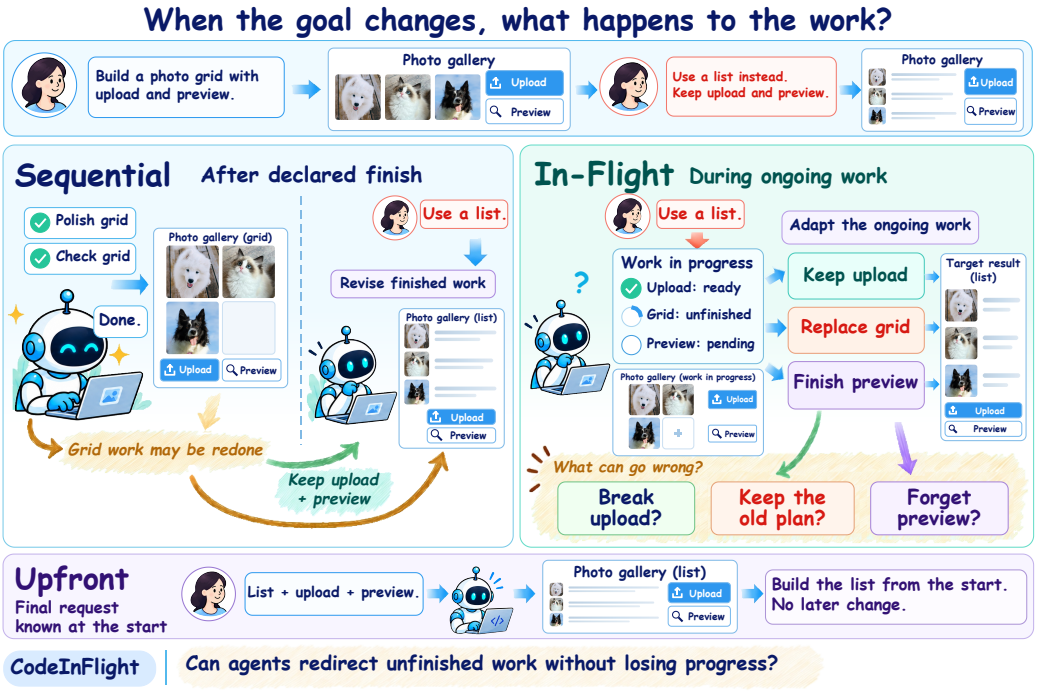


Figure A.1: **Following a changed request while implementation is unfinished.** In this illustrative gallery example, replacing a grid with a list requires retaining upload support, redirecting the layout, and completing the pending preview. UPFRONT, SEQUENTIAL, and IN-FLIGHT expose the revised request before implementation, after the initial attempt, or during ongoing work. The precise delivery and fallback rules appear in Section 3.3.

Meanwhile, the same final request poses a different problem depending on when the agent learns it. Told at the start, the agent faces an ordinary implementation task. Told after it has finished the grid, it must revise a completed attempt, and told in the middle, it must redirect work that is still in motion. Existing repository-level benchmarks do not separate these situations, because they either fix the specification before execution or reveal information across turns without controlling when a change arrives relative to the work (Section B). Scoring the final code under a single presentation would hide the effect of timing, and comparing different tasks would confound timing with task content. CodeInFlight therefore holds the final requirements of each instance fixed and varies only when the change arrives, before coding under UPFRONT, after the first attempt under SEQUENTIAL, and during that attempt under IN-FLIGHT. Because the same final tests score all three conditions and withdrawn requirements never count, a difference in outcome reflects when the agent learned the final request rather than what it had to build. Timing also changes how effort is spent, since an agent that learns of a change earlier can abandon superseded work sooner and generate less. A shorter trajectory is not necessarily a better one, however, so the evaluation reports complete resolution, partial test progress, and output tokens side by side.

Finally, such a controlled comparison needs data that did not exist. Issue trackers and benchmark tasks record a final specification, but they rarely record what an agent was first asked to do, how the request later changed, and which parts of the original work the change withdrew. We therefore built a dataset in which every instance pairs an initial request with a subsequent change and with executable final requirements that separate the work that remains active from the work that has been withdrawn. Drawing on authored projects, mined repository changes, and tasks derived from public benchmarks, CodeInFlight provides 1,551 execution-validated source tasks and a stratified core of 287 interaction instances (Section D). Together, these data turn the timing of a change from an accident of how a task was written into a variable that can be measured.

B Extended Related Work

This appendix expands the three strands of related work summarized in Section 2. Each strand closes with what it leaves open for CodeInFlight, and Table B.1 summarizes the benchmark designs discussed here.

B.1 Repository-Level Coding

Executable evaluation began by asking whether a generated program satisfies a stated specification. HumanEval and MBPP check function-level correctness with unit tests, and BigCodeBench adds complex instructions and combinations of library calls (Chen et al., 2021; Austin et al., 2021; Zhuo et al., 2025). RepoBench and CrossCodeEval add repository and cross-file context to code completion (Liu et al., 2024a; Ding et al., 2023). SWE-bench evaluates issue resolution in existing repositories, with human screening of tasks in its Verified subset (Jimenez et al., 2024; Chowdhury et al., 2024), while Multi-SWE-Bench, SWE-PolyBench, and SWE-rebench V2 broaden language coverage and task collection (Zan et al., 2025; Rashid et al., 2025; Badertdinov et al., 2026). These benchmarks share validated environments with acceptance tests, so a repository change is judged by executable behavior rather than by similarity to a reference patch.

Beyond isolated fixes, the difficulty shifts to coherence across larger changes. FeatureBench studies feature development across commits and pull requests, RoadmapBench evaluates coordinated version upgrades, and SWE-EVO uses release-level requirements (Zhou et al., 2026; Xu et al., 2026; Le et al., 2025). SlopCodeBench and SWE-Chain carry agent-produced code through successive specifications or upgrades and expose the accumulated consequences for functionality and maintainability (Orlanski et al., 2026; Lam et al., 2026). These settings show how earlier implementation choices constrain later work, but each stage begins with its scope known. CodeInFlight keeps this executable foundation and examines the same dependence within a single attempt, when a revision can arrive before the existing implementation is complete.

B.2 Interactive Coding

Larger tasks also raise the importance of information exchanged with the user. ColBench, introduced with SWEET-RL, studies collaborative backend programming and frontend design with simulated users (Zhou et al., 2025). Ambig-SWE examines clarification of underspecified requests, and Dialogue SWE-Bench evaluates repository work together with dialogue quality through a persona-grounded simulator (Vijayvargiya et al., 2026; King & Flanigan, 2026). SWE-Interact extends the interaction to progressive requirements and feedback from a simulator that can inspect the workspace (Raghavendra et al., 2026). Its comparison of fully specified and interactive conditions on the same tasks and final verifiers is a close precedent for separating task content from its presentation. In all of these benchmarks, however, the simulator responds when the agent chooses to communicate, so none tests a change the agent did not initiate.

Recorded sessions show when such changes occur in practice. SWE-chat documents direct interruptions and feedback after agent turns (Baumann et al., 2026), and SWE-Together reconstructs evaluations from these patterns with an anchored simulator that checks trajectory-dependent conditions after completed turns (Wu et al., 2026b). Jiang et al. (2026) associate late requirements in real sessions with replacement or deletion of earlier agent edits, and their controlled comparison of upfront and later-round disclosure concerns disclosure between rounds rather than during one. EvoCode-Bench supplies iterative requirements that extend or overturn earlier assumptions, with cumulative tests for the obligations that remain active, but issues the next instruction only after the preceding round passes (Shen et al., 2026). CodeInFlight instead delivers the same constructed revision before coding, after an initial attempt, or during unfinished execution, where retained functionality, withdrawn work, and pending obligations must be reconciled within the current trajectory.

B.3 Mid-Execution Adaptation

Intervening during execution introduces dependencies that completed-turn interaction does not capture. SWE-Touch injects task-relevant code edits at explicit triggers while preserving the requested repair, and Handoff Debt pauses a predecessor at specified checkpoints and varies the context sup-

Table B.1: **Comparison of benchmark settings.** Revised scope includes later additions or changes to requirements. During execution indicates an intervention before the current attempt ends. A runtime trigger is an explicit execution-relative checkpoint. A new turn alone does not meet this criterion. Partial marks indicate mixed or observation-only support. Marks in the columns shared with Table 1 follow that table.

Benchmark	SETTING		REQUEST DYNAMICS		
	Repository coding	Executable tests	Revised scope	During execution	Runtime trigger
FIXED-SPECIFICATION CODING					
SWE-bench Verified (Chowdhury et al., 2024)	✓	✓	–	–	–
Multi-SWE-bench (Zan et al., 2025)	✓	✓	–	–	–
SWE-PolyBench (Rashid et al., 2025)	✓	✓	–	–	–
FeatureBench (Zhou et al., 2026)	✓	✓	–	–	–
RoadmapBench (Xu et al., 2026)	✓	✓	–	–	–
SWE-EVO (Le et al., 2025)	✓	✓	–	–	–
INTERACTIVE CODING					
Ambig-SWE (Vijayvargiya et al., 2026)	✓	✓	–	–	–
Dialogue SWE-Bench (King & Flanigan, 2026)	✓	✓	–	–	–
SWE-Interact (Raghavendra et al., 2026)	✓	○	○	–	–
SWE-Together (Wu et al., 2026b)	✓	○	○	–	–
SWE-chat (Baumann et al., 2026)	✓	–	○	○	–
MULTI-STAGE CODING					
SlopCodeBench (Orlanski et al., 2026)	○	✓	✓	–	–
SWE-Chain (Lam et al., 2026)	✓	✓	–	–	–
EvoCode-Bench (Shen et al., 2026)	○	✓	✓	–	–
RUNTIME INTERVENTIONS AND GOAL REVISION					
SWE-Touch (Tan et al., 2026)	✓	✓	–	✓	○
Handoff Debt (KC & Budathoki, 2026)	✓	✓	–	✓	○
HAS-Bench (Wu et al., 2026a)	○	○	✓	–	–
InterruptBench (web) (Zou et al., 2026)	–	–	✓	✓	✓
CodeInFlight *(ours)*	✓	✓	✓	✓	✓

✓ present by design, ○ partial or subset, – absent. Partial marks cover mixed-domain programs, mixed test and rubric checks, feedback within an anchored task, naturally occurring rather than controlled changes, and triggers that change the workspace or context rather than the goal.

plied to a successor (Tan et al., 2026; KC & Budathoki, 2026). Both change the workspace or the context rather than the goal. InterruptBench changes the user's goal through additions, revisions, and retractions during web navigation, with timing controlled relative to a baseline trajectory (Zou et al., 2026), and is the closest analogue for studying goal updates and arrival timing together.

Other evaluations broaden the forms of coordination. τ-bench studies conversational tool use and policy compliance, and τ^2-Bench gives users and agents a shared environment in which both can act (Yao et al., 2025; Barres et al., 2025). Gaia2 adds asynchronous events and temporal constraints, DeskCraft studies evolving user input in desktop workflows, Full-Duplex-Bench-v3 examines spoken tool use under self-correction and turn-taking constraints, and HAS-Bench varies human participation roles and channels (Froger et al., 2026; Wang et al., 2026; Lin et al., 2026; Wu et al., 2026a). Together they show that an evaluation must specify what changes, when it changes, and how success is checked. The outcome also depends on the receiving system. Agent harnesses such as SWE-agent and OpenHands differ in how they inspect, edit, and test code (Yang et al., 2024; Wang et al., 2025). CodeInFlight brings goal updates into repository coding, where executable checks of revised requirements and existing behavior expose the consequences of redirection, and it holds the harness fixed while treating the reasoning mode as a separate factor.

C Limitations and Future Work

CodeInFlight measures whether a coding agent completes a revised request when a change reaches it at a controlled point of execution, with success judged by the executable tests of the final request. What lies beyond this measurement is of two kinds, the settings that the benchmark does not yet cover and the questions that its findings raise about how agents should handle a change.

C.1 Scope of the benchmark

Each trajectory in CodeInFlight contains one change, delivered as a text message at one point of execution. This design gives every delivery condition the same final requirements, yet real sessions can contain several revisions in succession, each arriving at its own moment and sometimes expressed through direct edits or review comments rather than a new message.

Because success is defined by the final required and regression tests, the benchmark evaluates what the revised software does rather than how the agent arrived at it. The tests therefore do not capture how an agent acknowledges or explains a change, nor whether it removes the code it wrote for work that the user withdrew, although both shape how a collaborator experiences the revision.

The execution setting is likewise held fixed. All configurations run in one shared harness, so that differences between delivery conditions arise from the model and the timing of the change rather than from the tooling. Native agent frameworks may manage context and interruption in their own ways, which Section J examines for one model.

The delivery protocol also couples arrival timing to the opportunity to respond. In particular, when a configuration does not declare completion, the SEQUENTIAL fallback occurs later than the IN-FLIGHT fallback and therefore leaves fewer post-change actions. The budget and message controls probe these components, but the primary contrast estimates the full delivery protocol rather than timing with the post-change action allowance held fixed.

The task pool bounds the benchmark in a similar way. Its tasks are drawn from three sources and span ten programming languages, so changes in other languages or in other kinds of software work, such as data analysis or infrastructure configuration, lie outside the present scope.

C.2 Future directions

Each of these boundaries suggests an extension of the benchmark, while the findings of Section 4 suggest how agents themselves could improve. Figure C.1 connects both to the directions below.

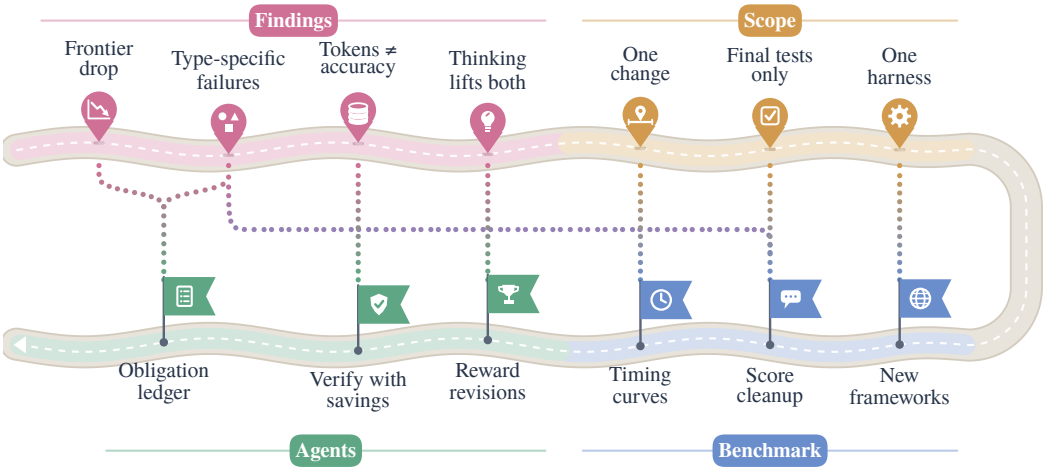


Figure C.1: **From findings and scope to future directions.** Pins on the upper road mark the findings of Section 4 and the scope of the benchmark, and flags on the lower road mark the directions they lead to, for agent design and for the benchmark itself.

A natural next step is to deliver several changes within one trajectory and to vary the arrival point from the first action to the last. Such a sweep would turn the comparison of three delivery conditions into a timing curve, revealing where along a trajectory a change costs the most and whether the loss of frontier agents grows with the amount of unfinished work. Running the same protocol in native frameworks and on tasks from further languages and domains would, in the same way, show how far the timing gap depends on the harness and the task pool.

A second extension looks beyond final correctness. How an agent treats withdrawn work deserves its own measure, because cancellations cost frontier agents the most complete solutions (Section 4.4) and final tests alone cannot tell whether the code written for a withdrawn request was removed, left dormant, or left to interfere with its replacement. Measures of residual withdrawn code and of how clearly an agent acknowledges a change would make both behaviors visible.

Beyond the benchmark itself, the central finding that frontier agents resolve fewer instances when a change arrives during execution calls for agents that manage unfinished work deliberately. Because each change type fails in its own way, with cancellations losing the most complete solutions and additional tasks costing partial correctness, agents appear to mishandle both what a change withdraws and what it keeps. An agent that maintains an explicit ledger of which obligations a change keeps, withdraws, or adds, and that checks its final code against this ledger before finishing, targets both failures directly.

The token results point to a complementary opportunity. Agents generate fewer output tokens when the change arrives during execution, yet the saved effort does not become accuracy (Section 4.3). Spending part of this effort on verification, for example by rerunning the tests of the features that a change retains, offers a direct way to turn the savings into correct revisions.

Finally, thinking raises accuracy under both deliveries by similar amounts, and frontier configurations that reason by default still resolve fewer instances under IN-FLIGHT (Section 4.5), which suggests that stronger general reasoning alone does not make an agent robust to changes during execution. Training signals that reward completing the revised request, on trajectories in which changes arrive at varied points, offer a more targeted route, and the executable final requirements of CodeInFlight can supply such rewards directly.

D Dataset Details

This appendix describes the dataset in four parts, i.e., sources and validation, the evaluation subset, task composition and test-suite scale, and language coverage. CodeInFlight combines authored projects, mined repository changes, and tasks derived from public coding benchmarks. Every task is validated before release, and a stratified subset of them is used for the three-condition evaluation. The dataset contains 1,551 validated source tasks. 1,390 support an interaction, and 287 interaction instances form the evaluated Core. One of these instances is excluded before evaluation, so 286 are run.

D.1 Sources and validation

We first define a source task, then list the checks it must pass before release. A source task comprises a repository state, a request, a reference change, and an executable verifier. Its acceptance targets identify the behaviors to implement, with a set of deciding tests for each target. Controlled tasks are authored projects with independently tested targets. Real-repository tasks derive their targets from mined commits or replayed upstream changes. External-derived tasks are drawn from public issue-resolution corpora and re-executed in their own environments (Badertdinov et al., 2026; Zan et al., 2025).

Validation checks provenance and licensing, environment setup, a failing initial state, a passing reference change, a reference-withheld negative control, verifier repeatability, per-test scoring, isolated execution, and duplicate identity. A task enters the release after all required checks pass. Table D.1 splits the 1,551 validated source tasks by source and evidence tier. The reference patch is used to validate the task and is excluded from the agent's context. The same validated environment and verifier are retained when constructing its interaction variants. Selecting a smaller subset from a larger validated pool also arises in training-data curation (Yang et al., 2023; Gao et al., 2025). Here that choice is a fixed stratified draw rather than a learned criterion.

Table D.1: **Validated source tasks.** Counts are grouped by validation evidence, with the tiers defined below. Repository counts exclude authored projects. One upstream repository occurs in two source categories.

Source	Tasks	Tier 1	Tier 2	Repositories	Languages
Controlled	231	231	0	–	4
Real repository	634	402	232	55	6
External-derived	686	468	218	480	9
Total	**1,551**	**1,101**	**450**	**534**	**10**

Evidence tiers. We first record what evidence each task carries about its initial state. On upstream-derived tier-1 tasks, every required test is observed failing at the base state and passing after the reference change. For tier-2 tasks, post-patch success is observed for every required test, while base-state failure is inferred from an aggregate upstream record or an unbuildable base state. Authored tasks use constructed negative controls. These tiers distinguish the available evidence for the task's initial state. All released tasks have a validated reference solution. The tier is retained in the released metadata.

Duplicate control. We then remove repeated tasks. Canonicalization matches repository and gold commit, repository and parent commit, upstream instance identifier, reference-patch digest, or normalized commit subject within a repository. Similarity signals trigger review but do not automatically merge tasks. The evaluation additionally includes at most one member of any group with identical required-test sets, reducing repeated measurements of the same behavior.

D.2 Evaluation subset

We now describe how the frozen evaluation set is drawn from the validated pool. Core is selected before model evaluation by change type and source, with language coverage within each cell and

a cap of 2 tasks per repository key. Authored projects use archetype keys for this cap. Each selected source task occurs in one interaction instance. Choosing an unrelated-task pair consumes both source tasks. Thus 287 instances contain 329 source tasks. Table D.2 gives the resulting number of instances in each source and change-type cell. The evaluation covers Python, Go, TypeScript, JavaScript, and Rust, while the full release retains ten languages.

Table D.2: **Core composition by source and change type.** Counts are interaction instances. Cancellation is constructed from repository-derived tasks. Real-repository cells reflect the available eligible tasks. The remaining cells meet the planned per-cell floor.

Source	Requirement change	Cancellation	Additional task
Controlled	32	–	30
Real repository	17	15	16
External-derived	49	83	45

Within Core, the three change types contain 74 constraint updates and 24 goal revisions, 98 cancellations, and 49 related and 42 unrelated additional tasks. Selection uses source metadata, target structure, and verifier evidence, without reading model outcomes. The ordered subset also supports smaller evaluations through its stratified prefixes. The full release, the interaction-eligible pool, and Core have different purposes: release breadth, construction coverage, and a balanced multi-model comparison, respectively.

D.3 Task composition and test-suite scale

To characterize what the release contains, we examine task composition and then test-suite scale. The three data sources contribute different kinds of software work to the release. Figure D.1 summarizes the task composition of all 1,551 validated source tasks. The 231 controlled tasks emphasize bug fixing and feature implementation. The 634 real-repository tasks contribute a larger feature-development share, while the 686 external-derived tasks contain substantial bug-fixing and multi-component work. Every category is labeled, including the absence of multi-component tasks in the controlled source.

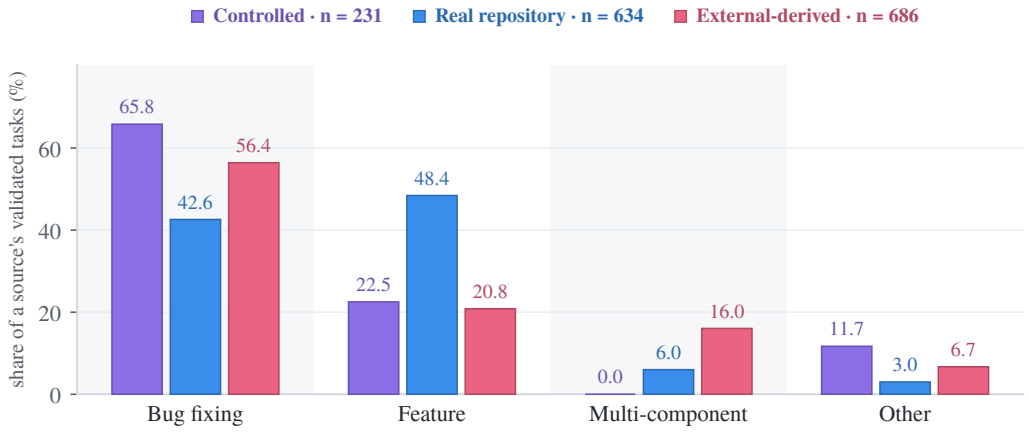


Figure D.1: **The three sources contribute different kinds of software work.** Bars report the within-source percentage of validated source tasks in each category, with every value labeled and source counts in the legend. Violet, blue, and pink denote controlled, real-repository, and external-derived tasks. Other combines API/behavior changes, refactoring, dependency upgrades, build/configuration work, and test/tooling tasks.

Test-suite size and acceptance-target count add two further dimensions to this diversity (Figure D.2). The median task contains 19 tests in the controlled source, 38.5 in the real-repository source, and 144 in the external-derived source. The real-repository source spans the widest range, including a long tail of large suites. The corresponding median numbers of acceptance targets are 5, 1, and 3. A target can contain multiple tests, so these quantities describe different aspects of the task. An

instance-averaged score consequently describes performance across tasks, while pooling individual tests would give larger suites more weight.

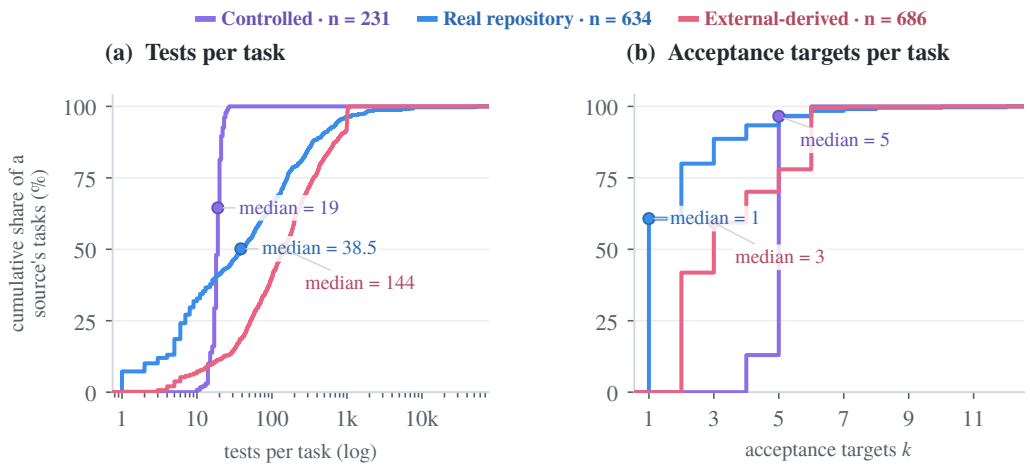


Figure D.2: **Test-suite breadth and target multiplicity have distinct distributions.** Empirical cumulative distributions over all validated source tasks of (a) tests per task, on a logarithmic axis, and (b) acceptance targets per task. Each curve is one source. The dot marks the source's median, printed beside it.

D.4 Language coverage and interaction structure

We next report language coverage, first for the full release and then for the interactions in Core. The release spans ten programming languages and 534 upstream repositories. From this pool, Core fixes 287 interaction instances drawn from 329 source tasks, including the two-source construction used for unrelated additions. Table D.3 separates these source-task and interaction counts so that the scale of the release and the composition of the evaluation set can be read directly.

Beyond these counts, Figure D.3 exposes the source-to-language hierarchy together with all language-by-source counts and task-level test-suite distributions. The distributions show how strongly suite size varies within a language. Language counts alone do not characterize executable coverage. These source-task summaries describe the validated release before constructing the smaller fixed Core evaluation set.

Within Core, every language contains requirement changes, cancellations with replacement, and additional tasks. Figure D.4 further decomposes these interactions into the five event families and their data sources. Python supplies the largest interaction group, followed by Go, TypeScript, JavaScript, and Rust. Their source mixtures differ: JavaScript draws predominantly from external-derived tasks, whereas Python, Go, and Rust combine all three sources. This coverage supports examining request structure within a language and checking whether a cross-model pattern persists across different codebases.

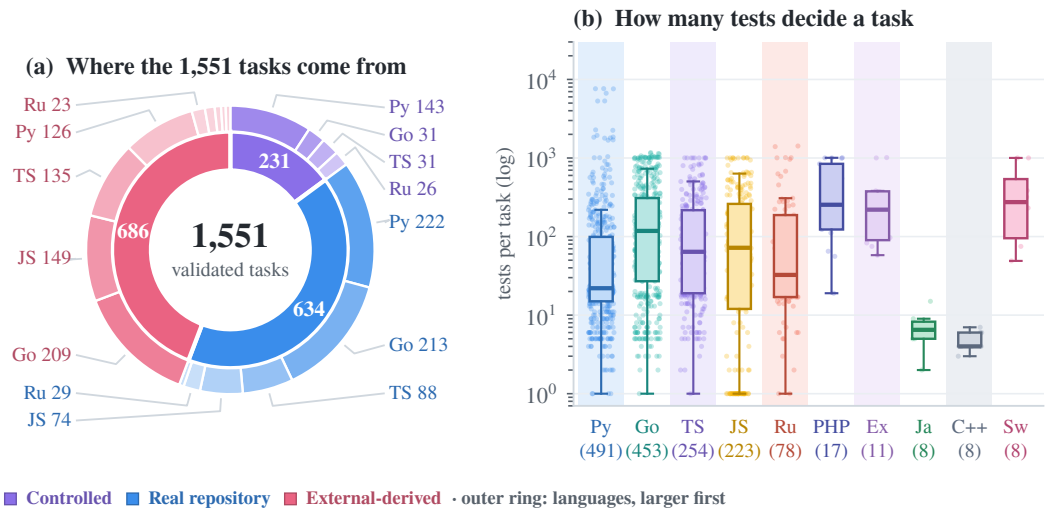


Figure D.3: **Source composition, language coverage, and executable scale of the release.** (a) The inner ring partitions the validated source tasks by source and the outer ring splits each source by language, larger languages first, with every count of at least 20 labeled outside the ring. (b) Test counts for every task grouped by language on a logarithmic axis, with boxes spanning the interquartile range, center lines at the median, whiskers to the most extreme values within 1.5 IQR, and language totals in parentheses. All panels describe source tasks before interaction construction.

Table D.3: **Language coverage of the release and Core.** Full counts validated source tasks. Core counts interaction instances. The final three columns partition each Core language by change type. A dash denotes a language outside Core.

Language	Full tasks	Core instances	Change	Replace	Add
Python	491	91	35	29	27
Go	453	72	22	23	27
TypeScript	254	56	18	21	17
JavaScript	223	48	17	20	11
Rust	78	20	6	5	9
PHP	17	–	–	–	–
Elixir	11	–	–	–	–
Java	8	–	–	–	–
C++	8	–	–	–	–
Swift	8	–	–	–	–
Total	**1,551**	**287**	**98**	**98**	**91**

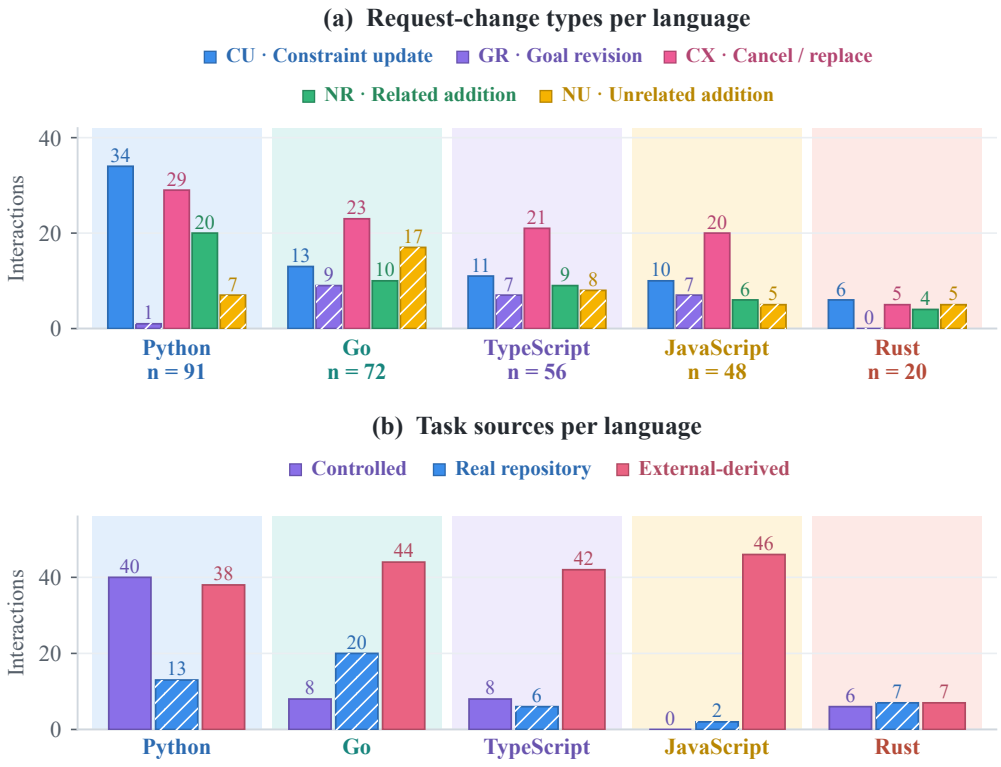


Figure D.4: **Core covers three request-change types across five languages.** (a) Exact interaction counts for constraint updates (CU), goal revisions (GR), cancellation/replacement (CX), related additions (NR), and unrelated additions (NU). CU and GR form requirement change. NR and NU form additional task. (b) Exact source counts within each language. Violet, blue, and pink denote controlled, real-repository, and external-derived tasks. Bars are solid, with a diagonal hatch on GR, NU, and the real-repository source to separate them from their neighbours. Every bar, including zero, is labeled. The counts under panel (a) give language totals. Counts include all 287 frozen Core interactions and are independent of model outcomes.

E INTERACTION CONSTRUCTION

This appendix specifies how interactions are built, in three parts, i.e., targets and final requirements, request and acceptance text, and worked examples from the dataset. The three change types are implemented through five event families. Each family changes the scope of a validated task while preserving its executable verifier. The following definitions and examples specify what the agent is asked to retain, add, or withdraw.

E.1 TARGETS AND FINAL REQUIREMENTS

We first fix the notation and the rule that determines what remains required. Let A denote the initially requested targets, B the arriving targets, and $T(X)$ the deciding tests of target set X. Each instance has a final required-test set T_{req} and a common regression set T_{reg}. All three conditions are scored against $R^* = T_{\text{req}} \cup T_{\text{reg}}$. Withdrawn tests, denoted W, are tracked separately and never enter final correctness. Table E.1 gives T_{req} and W for each of the five event families.

A target's behavioral unit is identified from its test file, class, package, or corresponding language-specific structure. Constraint updates add acceptance targets on the same unit. Goal revisions replace a target on that unit. Related additions use distinct units where these are known. For cancellation and related additions, targets are grouped by file when possible, then split at the boundary that minimizes imbalance in target count, followed by test count. When file units are unavailable, the construction uses target boundaries. Unrelated additions pair whole validated tasks from different repositories in the same language, with separate checkouts and verifiers.

Table E.1: **Event families and final scoring scope.** CU and GR are requirement changes. CX is cancellation/replacement. NR and NU are additional tasks. Regression tests are retained across conditions. In GR, a_r is the withdrawn target and $A' = A \setminus \{a_r\}$.

Family	Requested change	T_{req}	W
CU	Add a constraint on the same work	$T(A) \cup T(B)$	$\emptyset$
GR	Replace one requirement on the same unit	$T(A') \cup T(B)$	$T(a_r)$
CX	Stop the initial work and implement its replacement	$T(B)$	$T(A)$
NR	Add related work in the same repository	$T(A) \cup T(B)$	$\emptyset$
NU	Add an independent task in another repository	$T(A) \cup T(B)$	$\emptyset$

The families also differ in the task structure they require. CU requires at least two targets on one known unit. GR additionally requires a third target so that some initial work remains after withdrawal, together with base-state evidence that both the withdrawn and arriving tests can be collected. CX requires at least two targets and is constructed on repository-derived tasks. NR requires multiple known units or a target-boundary fallback. NU requires a compatible task from another repository. These structural conditions are evaluated before model runs.

E.2 REQUESTS AND ACCEPTANCE TESTS

We next describe the text the agent actually receives and the tests that decide the outcome. The initial request combines the task statement with the targets in scope. The update combines a family-specific framing sentence with the arriving targets and, when applicable, the withdrawal list. Statements come from the upstream issue or commit text, or from an authored specification. The scope names the acceptance tests. Their hidden assertions are applied during scoring. The three conditions use the same final test identities and verifier.

Under UPFRONT, all final targets appear in the opening request. Under SEQUENTIAL and IN-FLIGHT, the same initial request and update are delivered at different stages of execution (Section F.2). For cancellation, the UPFRONT request contains the replacement alone. For goal revision, it omits the withdrawn target. The comparison with UPFRONT therefore also captures the value of knowing the final scope before investing in work that will be withdrawn.

E.3 EXAMPLES FROM THE DATASET

Five Core examples, one per event family, connect the change rules to concrete software behavior, with messages abbreviated from the task records and an execution stage that is schematic rather than a reported model trajectory. Figure E.1 covers work that is retained, revised, or replaced: a constraint update in `bacpop/ska.rust` adds merge and delete checks to the repeat-masking work, a goal revision in `jlongster/prettier` keeps the class case but changes which function-expression case is required, and a cancellation in `tidwall/sjson` replaces deleting keys that contain escaped dots with setting them. Figure E.2 covers added work: a related addition in `unjs/ufo` adds `withQuery` to the work on `getQuery`, and an unrelated addition activates `spf13/afero` next to a parser fix in `itchyny/gojq`, so success requires both repairs. In every example the existing regression tests remain active, so withdrawing the sjson deletion target does not remove the library's ordinary deletion test from the final suite.

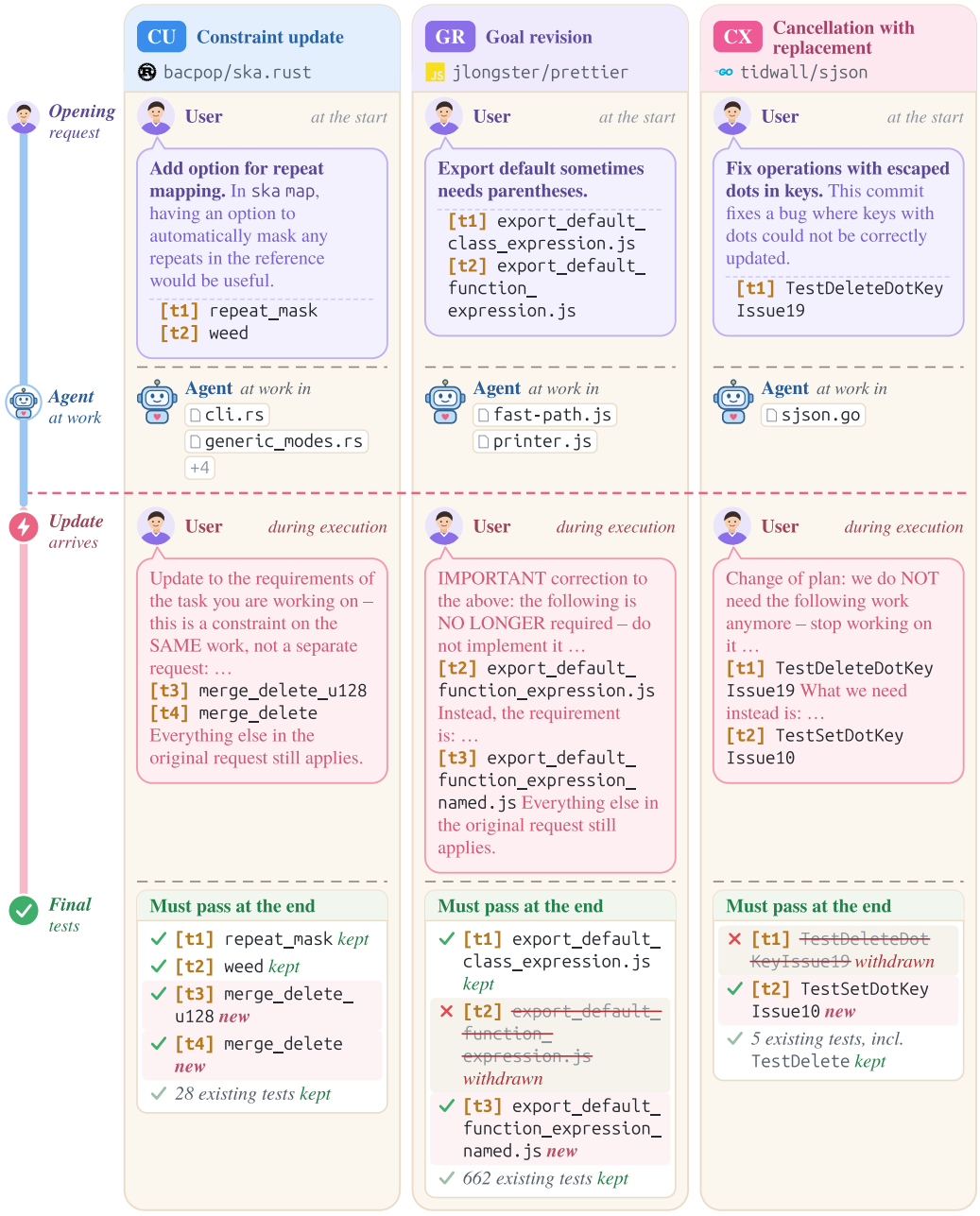


Figure E.1: **Retain, revise, or replace work already underway.** Each column follows one Core instance down a shared timeline, from the opening request through the update that arrives during execution (dashed pink line) to the tests that must pass at the end. Checks mark required targets, crosses mark withdrawn ones, and messages are abbreviated from the task records.

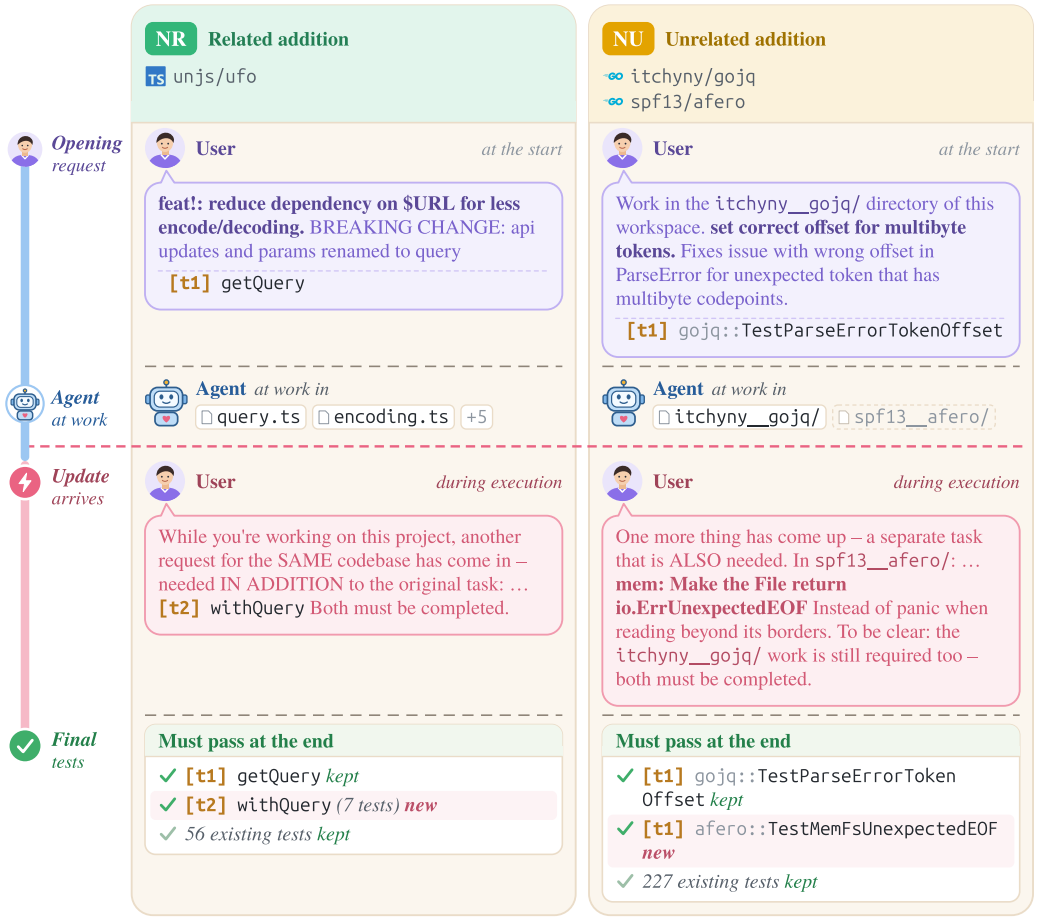


Figure E.2: **Adding work preserves the original obligation.** In a related addition (`unjs/ufo`) and an unrelated addition across two repositories (`itchyny/gojq` and `spf13/afero`), the update that arrives during execution adds a target while the original target and the existing tests stay required. Messages are abbreviated from the task records, and the agent stage is schematic.

F Evaluation Details

The evaluation connects a revised request to its final tests and to the trajectory that produced the implementation. We first define correctness and completion, then describe request delivery and model settings. Finally, we explain how paired comparisons account for shared tasks and uncertainty.

F.1 Correctness and request completion

For instance i, let $T_{\mathrm{req}}(i)$ contain the tests of the final requested targets and $T_{\mathrm{reg}}(i)$ the valid regression tests. A regression test enters this set if the verifier collects it and it passes at the reference state. With $\mathrm{pass}_i(t)$ denoting the final verdict,

$$\mathrm{Resolved}(i) = \mathbf{1}[\forall t \in T_{\mathrm{req}}(i) \cup T_{\mathrm{reg}}(i) : \mathrm{pass}_i(t)], \tag{1}$$

$$\mathrm{TPR}(i) = \frac{\sum_{t \in T_{\mathrm{req}}(i)} \mathbf{1}[\mathrm{pass}_i(t)]}{|T_{\mathrm{req}}(i)|}, \tag{2}$$

$$\mathrm{Completion}(i) = \frac{\#\{\text{final targets whose deciding tests all pass}\}}{k_i}, \quad k_i \geq 2. \tag{3}$$

Here k_i is the number of final targets. Resolved measures complete success, TPR captures partial progress at the test level, and Completion measures progress across multi-target tasks. Withdrawn tests enter none of these denominators. On repository tasks with observed initial verdicts, TPR corresponds to the fail-to-pass rate.

These per-instance quantities are then aggregated in two ways. For a set of episodes S, Macro TPR averages $\mathrm{TPR}(i)$ over instances, whereas Micro TPR divides the total number of passing required tests by the total number of required tests. The former gives each instance equal weight. The latter gives each test equal weight. Their difference reveals how test-suite size is associated with task outcomes. Under In-Flight, every configuration has a higher TPR than Resolved rate (Table H.3), because Resolved additionally requires every required and regression test to pass.

Arriving and withdrawn work. We then score the work that arrives and the work that is withdrawn. Event completion is the indicator that all tests of the arriving targets pass. For an unrelated addition, both repository tasks must resolve for the instance to resolve, and primary TPR pools the required tests of both tasks. Passing tests of withdrawn work do not by themselves establish continued effort on that work: a shared implementation can satisfy both old and new tests. Activity in a repository and successful completion of its tests are therefore distinct measurements.

F.2 Arrival conditions and execution settings

We now specify what is held fixed across the three conditions and what varies with delivery timing. All conditions begin from the same repository state and use the same system prompt, tools, final verifier, and within-configuration budget. The agent can inspect files, search, edit, run tests, and declare completion. Acceptance tests are protected from agent writes. Repository tasks execute in their validated environments. An unrelated-task instance has two independent checkouts and containers. The initial request and arriving update are defined in Section E.

Upfront supplies the final scope at the start. Sequential delivers the update at the first completion declaration, or at step 225 if the first part has not been declared complete. In-Flight delivers it at the first action boundary at the specified arrival step:

$$n_{\mathrm{arrival}}(a,i) = \begin{cases} \mathrm{round}(0.5 L_{a,i}), & \text{if a calibrated first-finish length } L_{a,i} \geq 6 \text{ exists,} \\ 125, & \text{otherwise.} \end{cases}$$

Rounding is half up. The first rule is model-adaptive. The second uses half the step budget. If an agent finishes before the trigger, it receives the update at that finish. A wall-clock stop can end an episode before delivery. The run records identify the delivery step and trigger.

Arrival thresholds are carried forward from each configuration's frozen Sequential calibration. Ten of the twelve local configurations use seed 1234, and Qwen3.5-27B and GLM-4.7-Flash use seed

4567. These ten configurations keep their original calibration lengths rather than recalibrating repaired trajectories. Final paired comparisons use the accepted Sequential and In-Flight outcomes of the same instance.

Additionally, the main benchmark configurations share 250 agent steps and a 275-call guard. The ten local configurations other than Qwen3.5-27B and GLM-4.7-Flash use an 8,192-token per-call output ceiling and a 65,536-token context setting. Qwen3.5-27B and GLM-4.7-Flash use a 16,384-token output ceiling and a 73,728-token context setting. Both local groups use a 57,344-token agent context budget, temperature 0.2, a 600-second test timeout, and a 1,800-second setup timeout. The request limits and generation controls of the API configurations are recorded separately in Table F.2. Container-based local records specify a 5,400-second episode limit. Records without an explicit wall-time field are not assigned that limit retrospectively. Steps and generated output are accumulated over the episode. These settings define the evaluated configurations, and each is held fixed across its three delivery conditions.

F.3 Model configurations

We next record exactly which checkpoints and endpoints were served. The main evaluation covers twelve local open-weight configurations and seven API-served configurations. The local main runs use the non-reasoning setting where a switch is supported. Table F.1 lists the served checkpoint and numerical format of each open-weight configuration. The API group uses the documented main deployment settings in Table F.2. The main table includes DeepSeek-V4-Flash with reasoning disabled. These deployment groups describe serving mode rather than model licensing. Tool compatibility is checked before evaluation, and identifiers distinguish checkpoints that share a family name.

Table F.1: **Open-weight checkpoints and numerical formats.** Checkpoints identify the weights actually served. API request identifiers and generation settings appear in Table F.2.

Model	Served checkpoint	Format
Granite-4.2-3B	`ibm-granite/granite-4.2-3b-fp8`	FP8
GLM-4.7-Flash	`QuantTrio/GLM-4.7-Flash-AWQ`	AWQ
Qwen3-Coder-30B	`QuantTrio/Qwen3-Coder-30B-A3B-Instruct-AWQ`	AWQ
Laguna-XS	`poolside/Laguna-XS-2.1-FP8`	FP8
Granite-4.2-8B	`ibm-granite/granite-4.2-8b-fp8`	FP8
Nemotron-3.5	`nvidia/NVIDIA-Nemotron-3.5-Lightning-30B-A3B-NVFP4`	NVFP4
Granite-4.2-30B	`ibm-granite/granite-4.2-30b-fp8`	FP8
Qwen3.5-9B	`QuantTrio/Qwen3.5-9B-AWQ`	AWQ
Devstral-Small-2	`mistralai/Devstral-Small-2-24B-Instruct-2512`	FP8
KAT-Coder	`cyankiwi/KAT-Coder-V2.5-Dev-AWQ-INT4`	AWQ INT4
Gemma-4-31B	`google/gemma-4-31B-it-qat-w4a16-ct`	QAT INT4
Qwen3.5-27B	`QuantTrio/Qwen3.5-27B-AWQ`	AWQ

For Gemini-3.5-Flash-Lite, GPT-5.6-Luna, Claude-Sonnet-5, GLM-5.3-Flash, and DeepSeek-V4-Flash, the harness records a seed of 2345 and a limit of 250 execution steps and 275 model calls. The recorded harness temperature is 0.2. The GLM gateway configuration overrides this to 1.0 with `top_p`=0.95 and retains thinking across tool turns (`clear_thinking=false`). Gemini uses the provider's default thinking configuration, recorded as minimal, and is not treated as reasoning OFF. The retained records do not expose a server context-window limit or establish that each provider enforces the supplied seed. These settings were checked against one original record per configuration and condition, together with its contemporaneous execution receipt. The audit does not assert that all implementation details are identical across endpoints.

F.4 Paired comparisons and uncertainty

Finally, we describe the estimand, the evidence rule for accepting an outcome, and the resampling procedure. The primary contrast is the paired difference between In-Flight and Sequential

Table F.2: **Recorded API configurations.** Exact endpoint model identifiers and per-request output ceilings. The seven main endpoints use the indicated reasoning settings.

Endpoint model identifier	Recorded reasoning setting	Output ceiling
`gemini-3.5-flash-lite`	Provider default (minimal)	16,384
`gpt-5.6-luna`	Effort `none`	8,192
`claude-sonnet-5`	OFF	8,192
`glm-5.3-flash`	Enabled, effort `max`	16,384
`deepseek-flash`	OFF	8,192
`gpt-6-astra`	Effort `high`	Provider default
`claude-opus-5-5`	Adaptive thinking, effort `high`	64,000

for the same model and interaction instance, with each paired instance weighted equally. Source-balanced estimates, which first average within each source category and then weight the categories equally, serve as a sensitivity analysis and are reported in the paired tables of Section H.4. Marginal condition scores and paired differences can use different numbers of instances, so tables report their respective sample sizes.

Correctness follows a single evidence-based rule applied independently of model identity and condition. An execution with a functioning verifier remains an observed outcome when the agent makes no useful change, fails to implement a required module, or introduces a compilation or import failure. Its recorded partial test passes are retained. Independent evidence of an absent environment, missing runner or dependencies, or a scoring-infrastructure failure excludes the affected episode from correctness. Successful writes to protected verifier files require trajectory evidence before exclusion. Ambiguous collection failures, unattributed verifier timeouts, and unverified labels remain unknown. They contribute neither a successful nor a failed verdict until their evidence is resolved.

Applying that rule, one of the 287 constructed Core instances is excluded before evaluation, so 286 instances are run, giving 286 scheduled cells per condition and 858 per configuration. Additional cell-level exclusions and unknown verdicts are reported in the centralized coverage table. The ten local configurations other than Qwen3.5-27B and GLM-4.7-Flash account for 8,580 planned cells: 8,438 scored outcomes, 81 unknown verdicts, 42 runs whose environment failed, and 19 excluded cells. These counts describe coverage. The primary correctness denominator contains only accepted, observed outcomes. Each paired estimate uses the intersection of its two required populations, and no alternate attempt is selected because it has a better score.

Intervals in the main text use 4,000 cluster-bootstrap draws with seed 20260923, and the paired tables of Section H.4 use 10,000 draws with seed 20260918. The cluster is the canonical source task. An unrelated-task construction is identified by its source-task pair. All episodes of a cluster within a contrast are resampled together. In the source-balanced tables, overall point estimates average the three source-category differences equally, while absolute condition scores are instance-equal means. Request-type contrasts and direct differences between those contrasts are described with their corresponding estimands. Intervals are pointwise and unadjusted for multiple comparisons. Frozen controls retain the bootstrap procedure documented in their own section.

Completion strata. The progress-based and budget-based analyses preserve each configuration's frozen arrival rule. This stratification describes how the update was scheduled. A completion declaration is separately recorded as a trajectory event and is not treated as evidence of passing the final tests. The budget and message controls provide complementary comparisons under deliberately varied protocols.

Output-token totals count generated text and exclude repeatedly transmitted prompts. A capped run contributes the resources consumed before its terminal state, so the reported means characterize the evaluated deployment budget.

G PROMPTS

This section reproduces the messages that the CodeInFlight harness sends to the agent, where colored fields stand for text that varies across instances. All three conditions share the system prompt, the tool interface, and the form of the opening user turn, and they differ only in where the change message appears: UPFRONT merges it into the opening request, SEQUENTIAL delivers it when the agent first declares completion, and IN-FLIGHT delivers it after the action at the arrival step.

System prompt

♠ **System message, identical in all conditions**

```
You are a coding agent collaborating with a user. The user gave you an initial
task description and may send more messages while you work (corrections, new
constraints, additional information, or answers to your questions). New user
messages appear as [USER @t=...] lines in the tool feedback; when they reach you
depends on when the user can get through to you.

You interact with the repository by calling the provided tools.
Call exactly ONE tool per turn. "write" replaces the whole file, so pass the
FULL new content.
Tests are run with the run_tests tool. Keep any accompanying text to one short
sentence.
In a real repository, start with tree and search rather than reading files at
random.
Change existing files with edit (one exact, unique occurrence); write replaces a
whole file.
Use "finish" when the task is complete and the tests pass.
In a real repository, start with "tree" and "search" rather than reading files
at random.

"ask" sends the user one specific question; the user replies when they are
available.
When you believe the task is complete and tests pass, use "finish".
```

Tool interface

♠ **Tool names, arguments, and descriptions**

```
tree(path?, depth?)  Show the repository's directory structure. Start here in an
    unfamiliar repository.
search(query, path?, offset?)  Search file CONTENTS across the repository for
    text or a regular expression. Returns path:line: matches.
find(name)  Locate a file by NAME or glob (e.g. "*_test.go").
list()  List every file path. Large repositories are truncated; prefer
    tree/find.
read(path, offset?, limit?)  Read a window of one file. Output is line-numbered;
    a truncated read reports the line to continue at.
edit(path, old, new)  Change existing code: replace one exact, unique occurrence
    of `old` with `new`. Use this rather than write for any file you did not
    create.
write(path, content)  Replace a file with new content. You must supply the FULL
    new file content.
revert(path)  Restore a file to its original content.
run_tests(path?)  Run the repository test suite, or only one directory's tests.
say(text)  Send the user a short message.
ask(question)  Ask the user one specific question. Non-blocking.
wait()  Wait for the user. Only if you are completely blocked; never twice in a
    row.
finish()  Declare the task complete.
```

Arguments marked with ? are optional.

Opening user turn

♥ **First user message, all conditions**

```
[USER @t=0] {opening_request}

[REPO]
{repository_file_list}
```

♦ **Opening request of a single-repository task**

```
{task_statement}

Scope of this request (all of the following acceptance tests must pass):
[{target_id}] {target_title} - tests:
    {test_id}
These identifiers name the acceptance tests that will be run against your final
state; they may not exist in the checked-out tree yet - implement the behaviour
they describe. Existing tests must keep passing.
```

♦ **Opening request of an unrelated addition**

```
Work in the `{dir_A}/` directory of this {workspace|repository}. {task_A} The
{workspace|repository} also contains another project directory that is not part
of this request - its tests are not your responsibility.
```

UPFRONT opening request

♦ **Constraint update**

```
{initial_request} ADDITIONALLY, from the start: {new_scope}
```

♦ **Related addition**

```
{initial_request} Additionally, on the same codebase: {added_request}
```

♦ **Unrelated addition**

```
Two independent tasks in this {workspace|repository}, both required.
1) In `{dir_A}/`: {task_A}
2) In `{dir_B}/`: {task_B}
```

♦ **Goal revision and cancellation**

The opening request of a single-repository task, with a scope that lists only the final targets. The withdrawn target is not named.

Change message

♥ **Constraint update**

```
Update to the requirements of the task you are working on - this is a constraint
on the SAME work, not a separate request: {new_scope} Everything else in the
original request still applies.
```

♥ **Goal revision**

```
IMPORTANT correction to the above: the following is NO LONGER required - do not
implement it and do not spend any more effort on it:
{withdrawn_target}
Instead, the requirement is: {new_scope}
Everything else in the original request still applies.
```

♥ **Cancellation**

```
Change of plan: we do NOT need the following work anymore - stop working on it
and do not spend any more effort there:
{withdrawn_targets}
What we need instead is: {new_scope}
```

♥ **Related addition**

```
While you're working on this project, another request for the SAME codebase has
come in - needed IN ADDITION to the original task: {added_request} Both must be
completed.
```

♥ **Unrelated addition**

One more thing has come up - a separate task that is ALSO needed. In `{dir_B}/`: {task_B} To be clear: the {dir_A}/ work is still required too - both must be completed.

♦ **Form of the scope block**

Scope of this request (all of the following acceptance tests must pass):
[{target_id}] {target_title} - tests:
    {test_id}

Withdrawn targets are listed as [{target_id}] {target_title}, one per line.

SEQUENTIAL delivery

♣ **Result of the first finish call**

[TOOL @t={t}] The user has more to say - the work is NOT over. Address the new message.

♥ **Next user turn**

[USER @t={t}] {change_text}

Without a completion declaration by step 225, the message follows the result of that step instead.

IN-FLIGHT delivery

♣ **Result of the action at the arrival step**

[TOOL @t={t}] {tool_output}

♥ **Next user turn**

[USER @t={t}] {change_text}

An agent that declares completion before the arrival step receives the SEQUENTIAL *form.*

No-change check-in

♣ **Result of the action at the arrival step**

[TOOL @t={t}] {tool_output}

♥ **Next user turn**

[USER @t={t}] Just checking in; no changes from my side, carry on.

Other harness replies

♣ **Result of an ask call**

[TOOL @t={t}] The user replies: everything you need is in the task descriptions and the repository - use your best judgment.

♣ **Result of a wait call**

[TOOL @t={t}] (waited; nothing new arrived. Do NOT wait again - continue working: read/write/run_tests.)

♥ **Reply without a tool call**

[SYSTEM] Your reply contained no tool call. Call exactly one tool now; if the work is complete or you cannot proceed, call `finish`.

♥ **Context notice after elided turns**

[SYSTEM] {n} earlier turns were elided to fit the context window. Re-read any file you need; nothing in the repository changed.
Every message the user sent, in order:
{user_messages}

No message states the step budget or the number of remaining steps.

H ADDITIONAL RESULTS

We examine the 19 main configurations in three stages. First, we connect complete solutions and partial correctness to the resources used. We then examine variation across request types, model

families, and codebases. Finally, paired comparisons show how request timing changes the set of solved instances, and the added-task analysis separates taking up work from completing it. All tables use the result set of the main comparison, with rows ordered by In-Flight Resolved within the open-weight and API groups.

H.1 Correctness and execution resources

This subsection reports the full condition-level results and reads them in three steps. We first compare correctness across the three delivery conditions, then relate correctness to the resources spent obtaining it, and finally ask how stable these resource profiles are across trajectories and conditions. Table H.3 reports test pass rate, complete resolution, and target completion under all three conditions, and Table H.4 the execution steps, input and output tokens, and elapsed time. Reading the two tables together separates partial progress from satisfying the whole request, and keeps the number of actions separate from the volume of generated text.

Under In-Flight, Resolved spans 3.5% to 89.9%. Claude-Opus-5.5 reaches 89.9%, while Qwen3.5-27B records the highest locally hosted score at 40.5%. Mean generated output per instance ranges from 1.78k tokens for GPT-5.6-Luna to 43.71k for Claude-Sonnet-5, so agents reach different levels of correctness with very different mixes of generated text and tool-mediated execution. The remainder of this subsection examines these quantities jointly.

Table H.1 condenses correctness under the three conditions and In-Flight resource use for all 19 configurations.

Table H.1: **Main results on CodeInFlight.** Resolved, test pass rate (TPR), and target completion (Comp.) are percentages (↑), and steps and output tokens (k) are means (↓). Resolved bars are proportional to the value in the colour of the condition (violet Upfront, blue Sequential, pink In-Flight). Within each model group, bold marks the best value and underlining the second best. Rows increase in In-Flight Resolved. Entries average each condition's scored instances, so they can differ by a few tenths from the paired values in Table 2.

Model	Upfront	Sequential	In-Flight				
	Resolved ↑	Resolved ↑	Resolved ↑	TPR ↑	Comp. ↑	Steps ↓	Tokens ↓
Open-weight models							
Granite-4.2-3B	4.6	3.5	3.5	5.5	7.5	238.4	16.1
GLM-4.7-Flash	10.4	10.0	10.1	23.6	23.6	211.8	8.5
Qwen3-Coder-30B	13.4	13.5	12.5	33.7	29.4	82.7	18.6
Laguna-XS	12.5	14.9	12.8	23.3	24.7	243.9	9.8
Granite-4.2-8B	13.7	11.5	12.9	25.0	23.1	211.8	14.4
Nemotron-3.5	18.6	16.0	14.9	29.2	29.7	203.7	12.8
Granite-4.2-30B	16.7	19.4	18.6	34.5	27.9	192.8	14.5
Qwen3.5-9B	14.3	18.7	18.9	36.3	38.6	200.4	16.2
Devstral-Small-2	23.5	18.9	20.1	40.7	39.6	149.7	9.5
KAT-Coder	20.4	22.2	21.1	32.5	33.8	212.9	11.4
Gemma-4-31B	30.5	28.7	27.1	49.7	45.9	**74.4**	**6.0**
Qwen3.5-27B	**39.8**	**40.1**	**40.5**	**60.3**	**55.7**	165.1	13.2
API models							
Gemini-3.5-Flash-Lite	32.7	27.9	28.8	41.0	28.8	99.4	4.2
GPT-5.6-Luna	49.7	51.7	49.3	67.8	62.6	26.5	**1.8**
Claude-Sonnet-5	75.3	75.2	68.6	86.2	88.2	69.8	43.7
GLM-5.3-Flash	78.9	81.0	76.1	89.0	87.9	87.1	43.6
DeepSeek-V4-Flash	78.2	81.3	76.2	90.4	92.4	69.0	10.5
GPT-6-Astra	89.2	81.5	78.3	86.3	88.1	36.6	3.4
Claude-Opus-5.5	**93.7**	**92.0**	**89.9**	**97.9**	**99.1**	**22.5**	6.6

Observation coverage. Before comparing these quantities, we record their observation requirements in Table H.2. Correctness, execution telemetry, and multi-target completion can use different

subsets. Completion eligibility is explicitly recorded for 158 multi-target instances (127 external-derived and 31 real-repository). Missing scores reduce the observed counts, and eligibility is not inferred when legacy records omit it. Absolute scores use instance means within the indicated cohort, while paired contrasts use shared instances and equal source weighting.

Table H.2: **Analysis coverage.** Each model is scheduled on 286 instances per condition. Score counts scored instances. Trace counts recorded trajectories. Comp. counts explicitly eligible and scored multi-target instances. Pairs counts jointly scored instances for IF versus the indicated condition. NU counts the common In-Flight cohort with action attribution and both source-task outcomes for unrelated additions. Missing outcomes are not imputed.

Model	Score			Trace			Comp.			Pairs		NU
	Up.	Seq.	IF	Up.	Seq.	IF	Up.	Seq.	IF	Seq.	Up.	IF
Open-weight models												
Granite-4.2-3B	282	283	282	284	284	275	156	156	156	281	279	41
GLM-4.7-Flash	279	279	277	285	285	285	157	157	157	275	273	42
Qwen3-Coder-30B	284	281	281	286	285	286	158	157	158	278	281	40
Laguna-XS	281	281	282	283	283	283	156	155	156	279	281	41
Granite-4.2-8B	278	279	280	284	283	285	157	156	157	275	277	42
Nemotron-3.5	280	281	281	284	284	283	156	156	155	279	278	41
Granite-4.2-30B	281	283	285	283	284	286	155	156	158	283	280	42
Qwen3.5-9B	280	284	281	284	285	284	156	157	156	281	278	41
Devstral-Small-2	281	281	279	284	284	283	156	156	155	278	277	41
KAT-Coder	285	284	284	285	285	285	157	157	157	283	283	42
Gemma-4-31B	275	279	280	279	283	284	155	156	156	278	274	42
Qwen3.5-27B	284	284	284	286	285	286	158	158	158	283	282	41
API models												
Gemini-3.5-Flash-Lite	281	280	281	285	285	286	155	153	154	279	278	38
GPT-5.6-Luna	286	286	286	286	286	286	158	158	158	286	286	42
Claude-Sonnet-5	283	286	283	286	286	286	158	158	157	283	280	39
GLM-5.3-Flash	285	284	285	286	286	286	158	158	157	283	284	41
DeepSeek-V4-Flash	284	284	286	286	286	286	158	158	158	284	284	42
GPT-6-Astra	286	286	286	286	286	286	158	158	158	286	286	42
Claude-Opus-5.5	286	286	286	286	286	286	158	158	158	286	286	42

Correctness across delivery conditions. Table H.3 shows that Resolved differs far less between conditions than between models: the Upfront minus In-Flight gap ranges from -4.6 points for Qwen3.5-9B to $+10.8$ points for GPT-6-Astra. In-Flight is below both other conditions for nine of the 19 configurations, sits between them for eight, and exceeds both for the two Qwen3.5 configurations. Consequently, the timing of the request rarely reorders the field, and the paired analyses in Section H.4 are needed to see whether it changes which instances a given model solves.

Correctness against resource use. We next ask whether higher correctness is bought with more generation or more actions. Figure H.1 places the 19 configurations by mean output tokens and In-Flight Resolved and divides the plane at the field medians of 11.4k tokens and 21.1%. Seven configurations resolve at least the median share while generating at most the median output, including DeepSeek-V4-Flash and GPT-5.6-Luna, whereas six generate more and resolve less, among them all three Granite-4.2 configurations. The two configurations with the largest output, GLM-5.3-Flash and Claude-Sonnet-5, are nevertheless among the five most accurate, so a large generation volume accompanies some of the best results as well as, in the lower-right quadrant, some of the weakest.

Configurations that resolve more instances tend to use fewer execution steps. On the common score-and-telemetry cohort, mean steps and Resolved have a Spearman correlation of -0.79 across the 19 configurations, and the six highest-scoring configurations all use fewer than the median 149

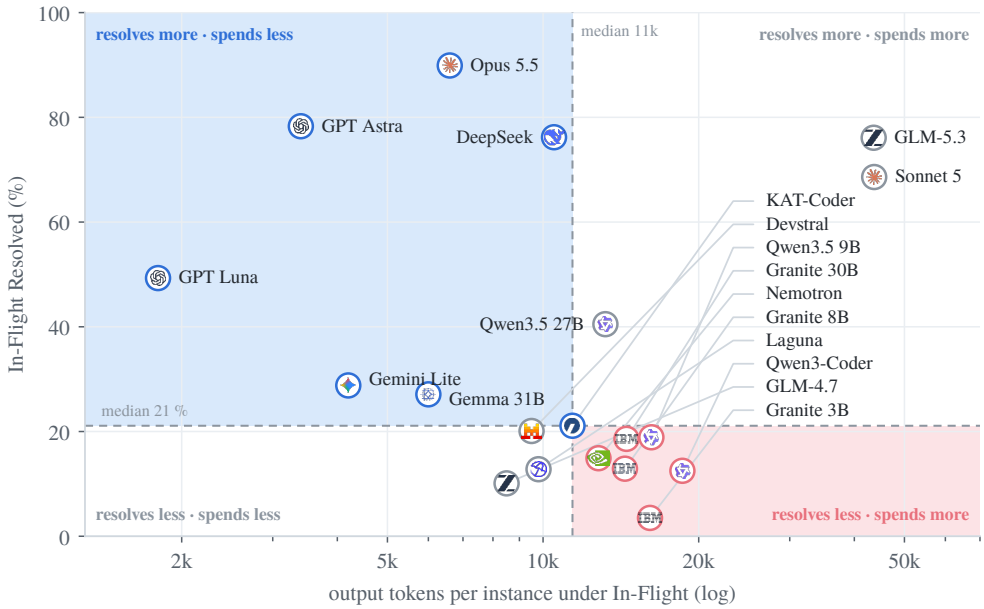


Figure H.1: **Output tokens against Resolved under In-Flight.** Each point is one configuration at the instance means of Tables H.3 and H.4. The horizontal axis is logarithmic. Dashed lines are the medians over the 19 configurations, so the quadrants describe position relative to the field: the upper-left quadrant (blue) holds configurations above the median Resolved and below the median output, and the lower-right quadrant (pink) the opposite. Names in the dense region are attached by leader lines. Output is bounded by the generation and action budgets, so the plane describes measured profiles rather than an efficiency ranking.

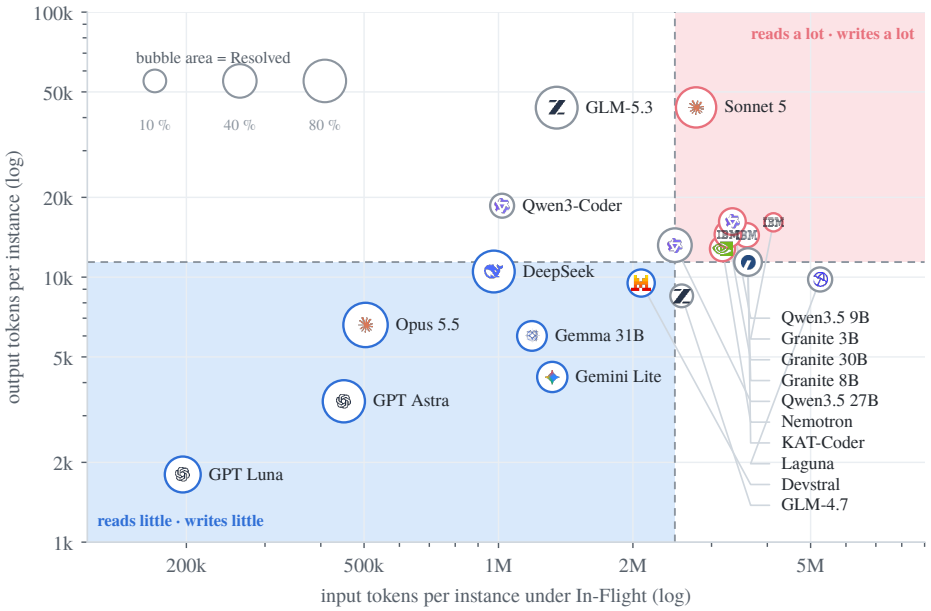


Figure H.2: **Input against output tokens under In-Flight.** Both axes are logarithmic instance means over recorded trajectories, and marker area is proportional to Resolved. Dashed lines are medians over the 19 configurations. The lower-left quadrant (blue) holds configurations below both medians and the upper-right quadrant (pink) those above both. Names in the dense region are attached by leader lines.

Table H.3: **Complete correctness results under the three delivery conditions.** TPR, Resolved, and Comp. are percentages. Entries are instance means, and Comp. covers eligible multi-target instances. Within each model group, bold marks the best value in a column and underlining the second best. The row with the highest In-Flight Resolved in each group is tinted. The Sequential block carries a pale band so that the three conditions can be read apart.

Model	Upfront			Sequential			In-Flight		
	TPR ↑	Res. ↑	Comp. ↑	TPR ↑	Res. ↑	Comp. ↑	TPR ↑	Res. ↑	Comp. ↑
Open-weight models									
Granite-4.2-3B	7.7	4.6	10.1	6.8	3.5	9.4	5.5	3.5	7.5
GLM-4.7-Flash	23.4	10.4	23.1	25.5	10.0	26.0	23.6	10.1	23.6
Qwen3-Coder-30B	34.9	13.4	32.5	35.2	13.5	31.7	33.7	12.5	29.4
Laguna-XS	18.8	12.5	20.7	22.2	14.9	24.9	23.3	12.8	24.7
Granite-4.2-8B	27.3	13.7	29.0	24.3	11.5	23.7	25.0	12.9	23.1
Nemotron-3.5	32.2	18.6	32.2	29.8	16.0	31.3	29.2	14.9	29.7
Granite-4.2-30B	34.8	16.7	29.6	36.1	19.4	29.6	34.5	18.6	27.9
Qwen3.5-9B	29.7	14.3	31.4	36.2	18.7	35.1	36.3	18.9	38.6
Devstral-Small-2	43.6	23.5	39.1	39.2	18.9	37.3	40.7	20.1	39.6
KAT-Coder	30.5	20.4	31.5	33.7	22.2	33.0	32.5	21.1	33.8
Gemma-4-31B	52.1	30.5	47.4	48.9	28.7	45.7	49.7	27.1	45.9
Qwen3.5-27B	**59.7**	**39.8**	**55.7**	**60.5**	**40.1**	**56.3**	**60.3**	**40.5**	**55.7**
API models									
Gemini-3.5-Flash-Lite	44.9	32.7	36.6	42.4	27.9	34.6	41.0	28.8	28.8
GPT-5.6-Luna	67.1	49.7	63.7	71.1	51.7	68.2	67.8	49.3	62.6
Claude-Sonnet-5	88.3	75.3	92.2	89.2	75.2	90.7	86.2	68.6	88.2
GLM-5.3-Flash	87.6	78.9	89.3	90.2	81.0	91.6	89.0	76.1	87.9
DeepSeek-V4-Flash	92.3	78.2	94.6	92.9	81.3	93.4	90.4	76.2	92.4
GPT-6-Astra	96.3	89.2	97.4	89.3	81.5	89.6	86.3	78.3	88.1
Claude-Opus-5.5	**98.1**	**93.7**	**99.3**	**97.0**	**92.0**	**99.3**	**97.9**	**89.9**	**99.1**

steps. Steps are capped by the action budget and tokens by the generation budget, so these relations describe where the measured configurations lie rather than an efficiency ranking.

Meanwhile, Figure H.2 separates what a configuration reads from what it writes. Input volume spans more than an order of magnitude, from 196k tokens for GPT-5.6-Luna to 5.25M for Laguna-XS, and the configurations that read the most are those that run to the step cap. Four of the seven configurations below both medians, Claude-Opus-5.5, GPT-6-Astra, DeepSeek-V4-Flash, and GPT-5.6-Luna, are among the six most accurate, so reading and writing less is compatible with high correctness.

Consistency of resource use and how trajectories end. Means can hide how consistently a configuration consumes its budget, and for some configurations they are not summaries at all. Figures H.3 and H.4 show the recorded In-Flight distributions with the individual trajectories overlaid. Six open-weight configurations reach the 250-step action cap on more than 70% of their trajectories, from Qwen3.5-9B at 72.2% to Laguna-XS at 96.5%, so their mean step counts are censored at the cap and their medians equal it. Every API configuration, in contrast, is capped on fewer than 8% of trajectories, and Claude-Opus-5.5 is the most consistent of all with an interquartile range of 14 to 26 steps. Output tokens show the opposite kind of spread at the top of the field: GLM-5.3-Flash and Claude-Sonnet-5 have interquartile ranges of roughly 14k to 57k and 9k to 51k tokens, so their large means arise from long tails rather than from uniformly long generations.

Figure H.5 makes the same point from the perspective of how trajectories stop. The share of trajectories that end with a declared finish spans 3.2% for Laguna-XS to 100.0% for Claude-Opus-5.5 and tracks Resolved closely across the 19 configurations (Spearman correlation 0.85). However, a declared finish is an agent action, and correctness is determined by the final tests: Qwen3-Coder-30B declares completion on 79.7% of its trajectories yet resolves 12.5% of instances, whereas Qwen3.5-

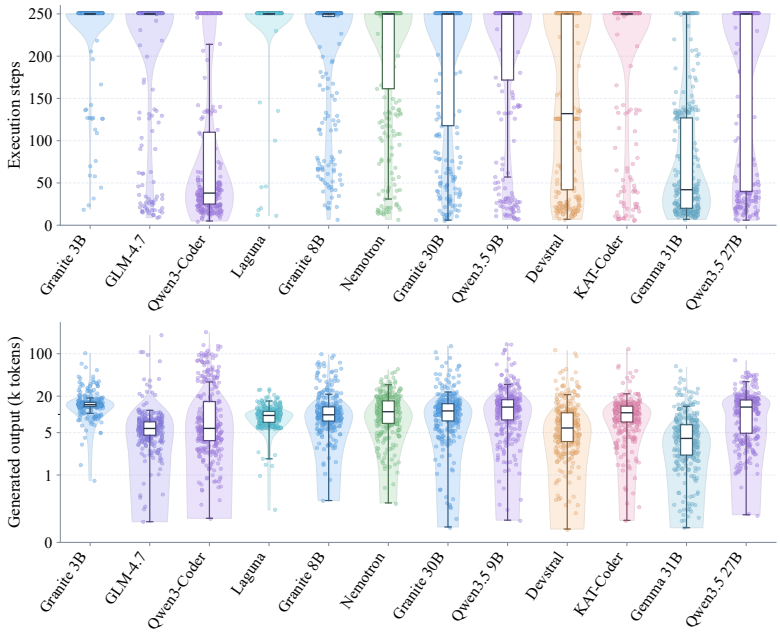


Figure H.3: **Resource-use distributions for open-weight models.** Recorded In-Flight trajectories are shown as jittered points with a density envelope and a box plot (median and interquartile range, with whiskers following the 1.5-IQR rule). The top panel reports execution steps. The bottom panel reports generated output tokens in thousands on a symmetric-log axis, linear near zero.

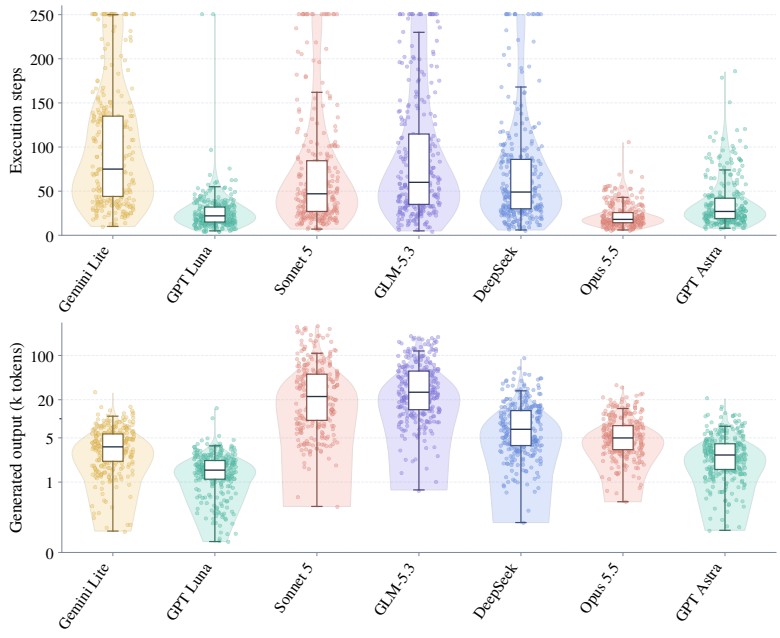


Figure H.4: **Resource-use distributions for API models.** The construction and units match Figure H.3. Distributions use available execution records independently of whether the final tests yielded a score.

Table H.4: **Complete resource results under the three delivery conditions.** Steps are mean execution steps. In and Out are input and output tokens in thousands. Min is elapsed minutes. Lower is better in every column, and entries are instance means over recorded trajectories. Marking follows Table H.3.

Model	Upfront (lower is better)				Sequential (lower is better)				In-Flight (lower is better)			
	Steps	In (k)	Out (k)	Min	Steps	In (k)	Out (k)	Min	Steps	In (k)	Out (k)	Min
Open-weight models												
Granite-4.2-3B	241.2	4270	15.5	9.9	239.4	4244	15.5	11.5	238.4	4128	16.1	10.4
GLM-4.7-Flash	216.1	2646	8.0	**9.4**	209.7	2536	8.3	11.8	211.8	2572	8.5	11.1
Qwen3-Coder-30B	89.1	**1137**	19.7	11.6	93.7	**1198**	20.0	13.7	82.7	**1019**	18.6	11.3
Laguna-XS	242.8	5222	9.6	9.5	244.1	5206	9.8	**10.1**	243.9	5252	9.8	**9.8**
Granite-4.2-8B	210.4	3601	13.3	21.6	211.9	3602	12.8	21.3	211.8	3599	14.4	22.1
Nemotron-3.5	205.7	3268	14.1	19.5	213.4	3326	13.9	21.3	203.7	3174	12.8	21.7
Granite-4.2-30B	199.9	3426	16.2	33.8	199.3	3382	17.0	29.8	192.8	3264	14.5	27.5
Qwen3.5-9B	208.3	3451	20.5	17.2	200.5	3347	17.8	17.3	200.4	3340	16.2	16.1
Devstral-Small-2	171.5	2459	10.3	14.9	188.9	2638	11.2	15.7	149.7	2087	9.5	14.4
KAT-Coder	214.4	3782	11.4	24.7	210.3	3610	11.8	23.0	212.9	3622	11.4	23.4
Gemma-4-31B	**80.2**	1429	**5.4**	45.5	**92.3**	1547	**7.3**	35.2	**74.4**	1188	**6.0**	38.8
Qwen3.5-27B	172.3	2693	14.0	31.2	172.0	2650	14.3	31.6	165.1	2483	13.2	30.3
API models												
Gemini-3.5-Flash-Lite	146.4	2166	5.6	17.5	133.2	1870	5.7	15.8	99.4	1320	4.2	12.9
GPT-5.6-Luna	25.0	**185**	**1.8**	**6.3**	30.9	**236**	**2.0**	**7.8**	26.5	**196**	**1.8**	**7.2**
Claude-Sonnet-5	69.0	2995	45.3	17.1	76.4	2426	49.9	18.8	69.8	2770	43.7	17.7
GLM-5.3-Flash	91.4	1422	42.9	27.6	93.2	1446	46.9	29.5	87.1	1350	43.6	27.4
DeepSeek-V4-Flash	72.1	1029	10.8	10.4	75.6	1059	11.3	10.9	69.0	976	10.5	9.5
GPT-6-Astra	38.8	486	3.7	30.9	40.3	483	3.9	48.0	36.6	450	3.4	39.0
Claude-Opus-5.5	**23.9**	572	7.2	12.7	**26.6**	649	7.6	8.2	**22.5**	504	6.6	8.7

27B declares completion on only 40.9% and resolves 40.5%. Time limits never dominate, accounting for at most 11.3% of endings for any configuration.

Resource profiles across delivery conditions. Finally, we ask whether these profiles depend on when the request arrives. Figure H.6 places the 19 configurations by mean steps and output under each condition, with marker colour showing Resolved on a shared scale. The three panels are nearly identical. The step extremes are held by Claude-Opus-5.5 and Laguna-XS in every panel, and 16 of the 19 configurations move by fewer than 12 mean steps between conditions. Under the In-Flight median split of Figure H.1, the favorable upper-left quadrant holds seven configurations under Upfront, six under Sequential, and seven under In-Flight, and only Devstral-Small-2 and KAT-Coder change quadrant, each of them close to a median line. The plane therefore moves far less between conditions than between configurations: request timing changes the outcome of individual instances, examined in the following subsections, more than it changes a model's overall resource profile.

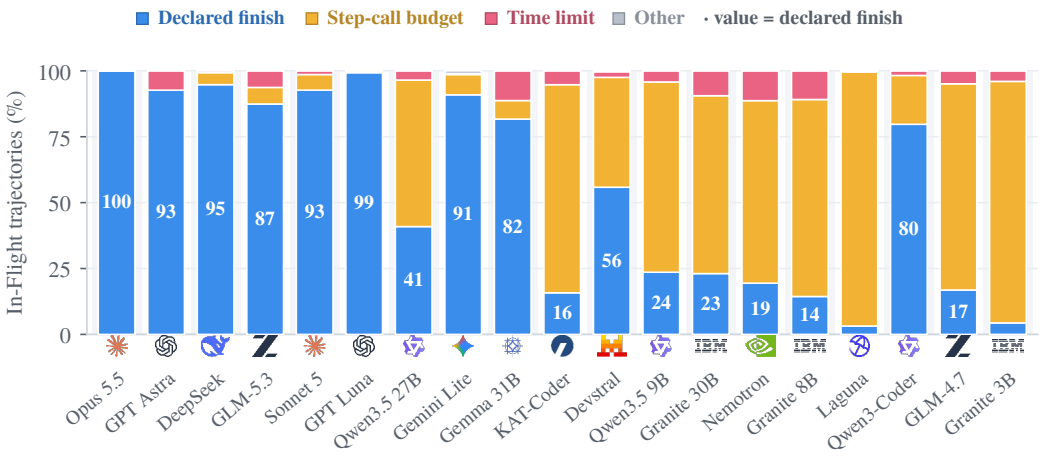


Figure H.5: **How In-Flight trajectories end.** Stacked vertical bars partition the recorded trajectories of each configuration, ordered by In-Flight Resolved, into declared finish, step or call-budget exhaustion, time limit, and other endings. The declared-finish share is printed inside its segment when it is large enough. A declared finish is an agent action and is distinct from test-based resolution.

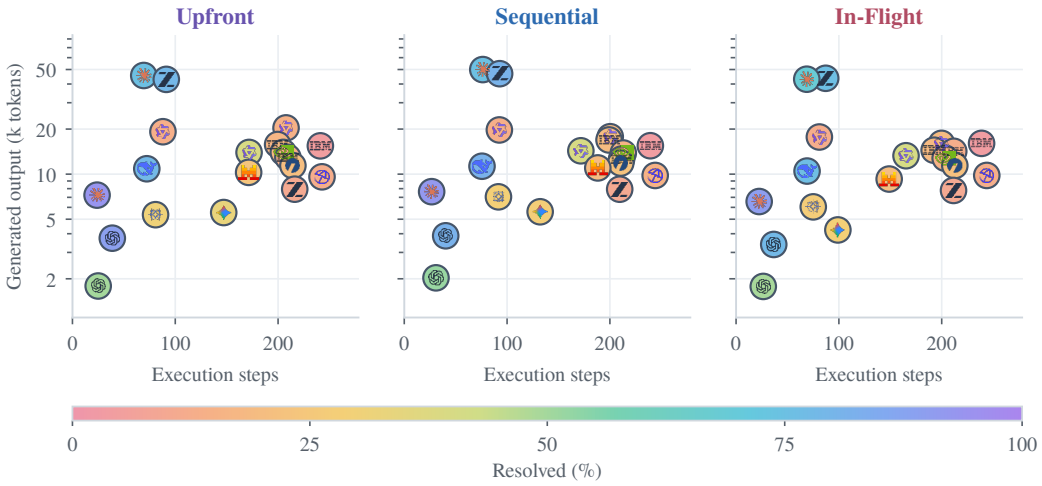


Figure H.6: **Correctness adds a third dimension to resource profiles.** Each panel places the 19 configurations by their mean execution steps and generated output (k tokens, logarithmic axis). Vendor icons identify configurations and marker colour shows Resolved on the shared 0 to 100% scale below. All three quantities use the same score-and-telemetry cohort within each model and condition. These observed subsets can differ across conditions. The panels describe the measured configurations and do not define a causal relationship between resources and correctness.

H.2 VARIATION ACROSS REQUEST-CHANGE TYPES

We next separate the three request-change types, first in correctness and then in the paired effect of delivery timing. A single overall score combines three different adaptation objectives. Requirement changes revise the current work, cancellation replaces its target, and additional tasks preserve the original obligation while introducing another. Tables H.5 to H.7 therefore separate the three types under each condition, with individual columns for partial test success and complete resolution.

Across the three tables, additional tasks have the strictly lowest Resolved rate in 52 of the 57 model–condition combinations. The pattern also holds for 18 of the 19 configurations under UPFRONT, where no change arrives during work, so it reflects the difficulty of completing a combined request more than the timing of the change. At the same time, the per-type TPR columns show how much of that work is satisfied even when the entire request is not resolved. Reading the two measures together distinguishes degrees of partial completion.

Figure H.7 separates the timing effect by configuration and change type. Output tokens fall under IN-FLIGHT in 43 of the 57 cells, and the reduction is largest for cancellations with replacement, where 15 of the 19 configurations generate fewer tokens and the median change is -15%, against -7% for requirement changes and -4% for additional tasks. The Resolved losses concentrate in the six frontier configurations, which lose in 15 of their 18 cells, most sharply for GPT-6-Astra on cancellations (14.4 points), DeepSeek-V4-Flash on cancellations (9.3 points), and Claude-Sonnet-5 on additional tasks (9.1 points). The other 13 configurations change by -5.2 to $+7.0$ points, with both gains and losses within every change type. Output tokens rise in 14 of the 57 cells, and the largest increase is $+30\%$ for Granite-4.2-8B on additional tasks.

Separating the change types into their five event families locates the frontier loss (Figure H.8). Within requirement changes, the frontier configurations lose 3.0 points on constraint updates, whereas goal revisions, which contribute only 144 frontier pairs, show no change in Resolved. Cancellation with replacement produces both the largest frontier loss (5.3 points) and the largest token reduction (20%), and both additional-task families also lose, by 3.7 points for related and 4.6 points for unrelated tasks. In these two families the frontier token reduction (2% and 8%) is smaller than that of the other 13 configurations (5% and 9%). The other 13 configurations stay within 1.3 points of zero in every family while generating 1% to 10% fewer output tokens.

Figures H.9 and H.10 extend this view to every configuration and every event family under all three delivery conditions. Across the 19 configurations, the two additional-task families are the hardest, with a mean SEQUENTIAL Resolved of 24.9% for unrelated and 29.1% for related additional tasks against 46.6% for goal revisions. IN-FLIGHT delivery falls below SEQUENTIAL delivery most often for cancellations with replacement, in thirteen of the 19 configurations, and for unrelated additional tasks, in eleven. The loss is concentrated in the six strongest configurations, where IN-FLIGHT is lower in 21 of their 30 family results. The other configurations resolve few additional tasks under any condition, and IN-FLIGHT is lower in only 30 of their 65 family results.

The effect of delivery timing can also differ across these objectives. Figure H.11 pairs correctness and generated-output changes within each request type. Both panels use the same set of instances with scores and telemetry under both Sequential and In-Flight. These comparisons distinguish a change in the amount of generated work from a change in how much of the final request passes its tests.

Figure H.12 places each configuration by its change in output tokens and in Resolved separately for the three request types.

Table H.5: **Upfront correctness by request-change type.** TPR and Resolved are percentages (↑). Additional-task Resolved requires satisfying the combined final request. Each Resolved value sits on a bar proportional to it. Bold marks the best value in a column within each model group and underlining the second best.

Model	Requirement change		Cancel / replace		Additional task	
	TPR ↑	Resolved ↑	TPR ↑	Resolved ↑	TPR ↑	Resolved ↑
Open-weight models						
Granite-4.2-3B	10.7	6.1	7.3	7.3	4.7	0.0
GLM-4.7-Flash	32.5	14.6	20.0	13.5	17.3	2.3
Qwen3-Coder-30B	46.2	21.9	28.4	15.5	30.0	2.2
Laguna-XS	19.1	16.3	21.7	16.8	15.4	3.4
Granite-4.2-8B	35.3	17.9	23.8	17.7	22.4	4.6
Nemotron-3.5	36.8	24.0	33.9	25.0	25.2	5.7
Granite-4.2-30B	42.4	17.0	31.8	22.9	30.1	9.9
Qwen3.5-9B	38.3	20.2	22.5	16.5	28.4	5.6
Devstral-Small-2	50.1	24.7	38.7	34.4	41.8	10.2
KAT-Coder	35.4	23.5	34.2	29.9	21.1	6.7
Gemma-4-31B	58.6	37.9	51.2	38.7	46.0	13.8
Qwen3.5-27B	**68.0**	**46.4**	**56.3**	**45.4**	**54.3**	**26.7**
API models						
Gemini-3.5-Flash-Lite	54.2	44.2	34.4	28.9	46.6	24.7
GPT-5.6-Luna	74.1	57.1	65.4	55.7	61.4	35.2
Claude-Sonnet-5	86.6	69.1	91.8	87.6	86.4	68.5
GLM-5.3-Flash	90.4	82.5	85.6	83.5	87.0	70.3
DeepSeek-V4-Flash	89.8	73.5	94.8	89.7	92.4	70.8
GPT-6-Astra	96.1	**90.8**	96.9	93.8	95.7	82.4
Claude-Opus-5.5	**96.2**	90.8	**99.0**	**97.9**	**99.2**	**92.3**

Table H.6: **Sequential correctness by request-change type.** TPR and Resolved are percentages (↑). Additional-task Resolved requires satisfying the combined final request. Each Resolved value sits on a bar proportional to it. Bold marks the best value in a column within each model group and underlining the second best.

Model	Requirement change		Cancel / replace		Additional task	
	TPR ↑	Resolved ↑	TPR ↑	Resolved ↑	TPR ↑	Resolved ↑
Open-weight models						
Granite-4.2-3B	11.4	7.2	3.6	3.1	5.1	0.0
GLM-4.7-Flash	30.2	11.7	24.9	15.5	21.2	2.3
Qwen3-Coder-30B	44.4	20.8	26.1	15.5	35.1	3.4
Laguna-XS	25.0	21.6	19.8	16.8	21.7	5.6
Granite-4.2-8B	32.5	13.4	18.9	16.7	21.2	3.5
Nemotron-3.5	33.7	16.5	33.5	26.0	21.5	4.5
Granite-4.2-30B	41.9	23.7	33.0	26.8	33.1	6.7
Qwen3.5-9B	42.2	23.7	34.9	23.7	31.2	7.8
Devstral-Small-2	50.7	25.8	33.2	24.0	33.2	5.7
KAT-Coder	38.1	27.8	37.0	29.9	25.6	7.8
Gemma-4-31B	62.1	36.8	40.4	36.1	44.1	11.5
Qwen3.5-27B	**66.0**	**46.9**	**59.0**	**51.0**	**56.1**	**21.1**
API models						
Gemini-3.5-Flash-Lite	53.1	39.6	36.8	30.9	36.8	11.5
GPT-5.6-Luna	79.2	61.2	64.2	55.7	69.7	37.4
Claude-Sonnet-5	86.6	69.4	89.8	84.5	91.3	71.4
GLM-5.3-Flash	90.6	81.4	85.9	85.6	94.4	75.6
DeepSeek-V4-Flash	90.7	81.2	94.2	91.8	93.8	70.3
GPT-6-Astra	90.3	82.7	87.6	82.5	89.9	79.1
Claude-Opus-5.5	**96.2**	**90.8**	**95.9**	**93.8**	**99.2**	**91.2**

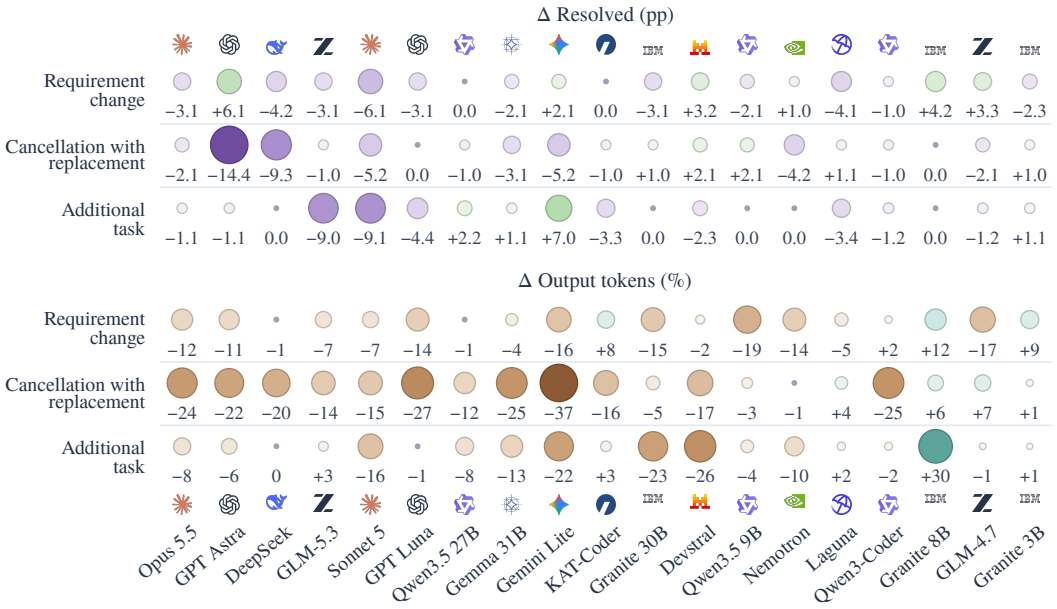


Figure H.7: **Timing effects by request type for all 19 configurations.** Each circle shows the change from SEQUENTIAL to IN-FLIGHT in Resolved (top, points) or output tokens (bottom, percent) for one configuration and change type, with area proportional to the size of the change and configurations ordered by SEQUENTIAL Resolved. Output tokens fall in 43 of the 57 cells, most of all for cancellations with replacement.

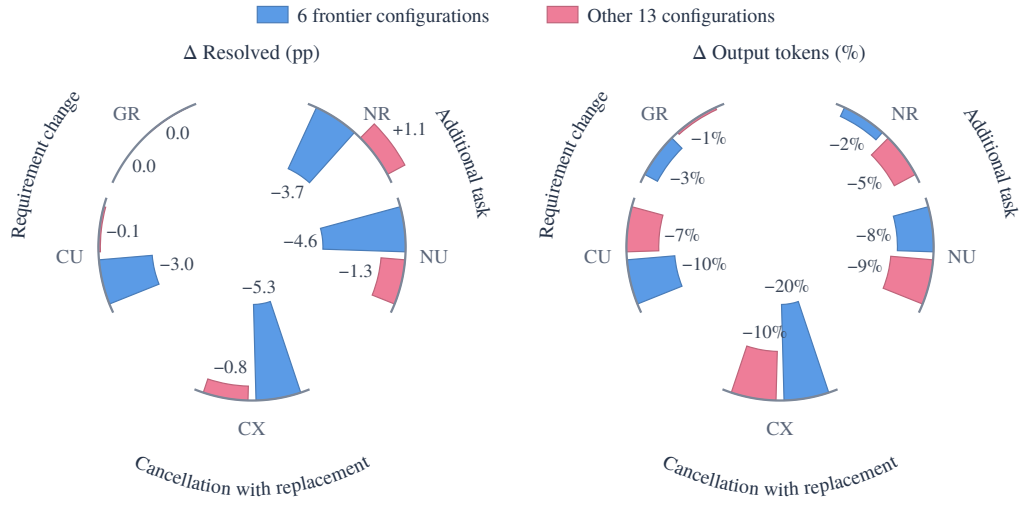
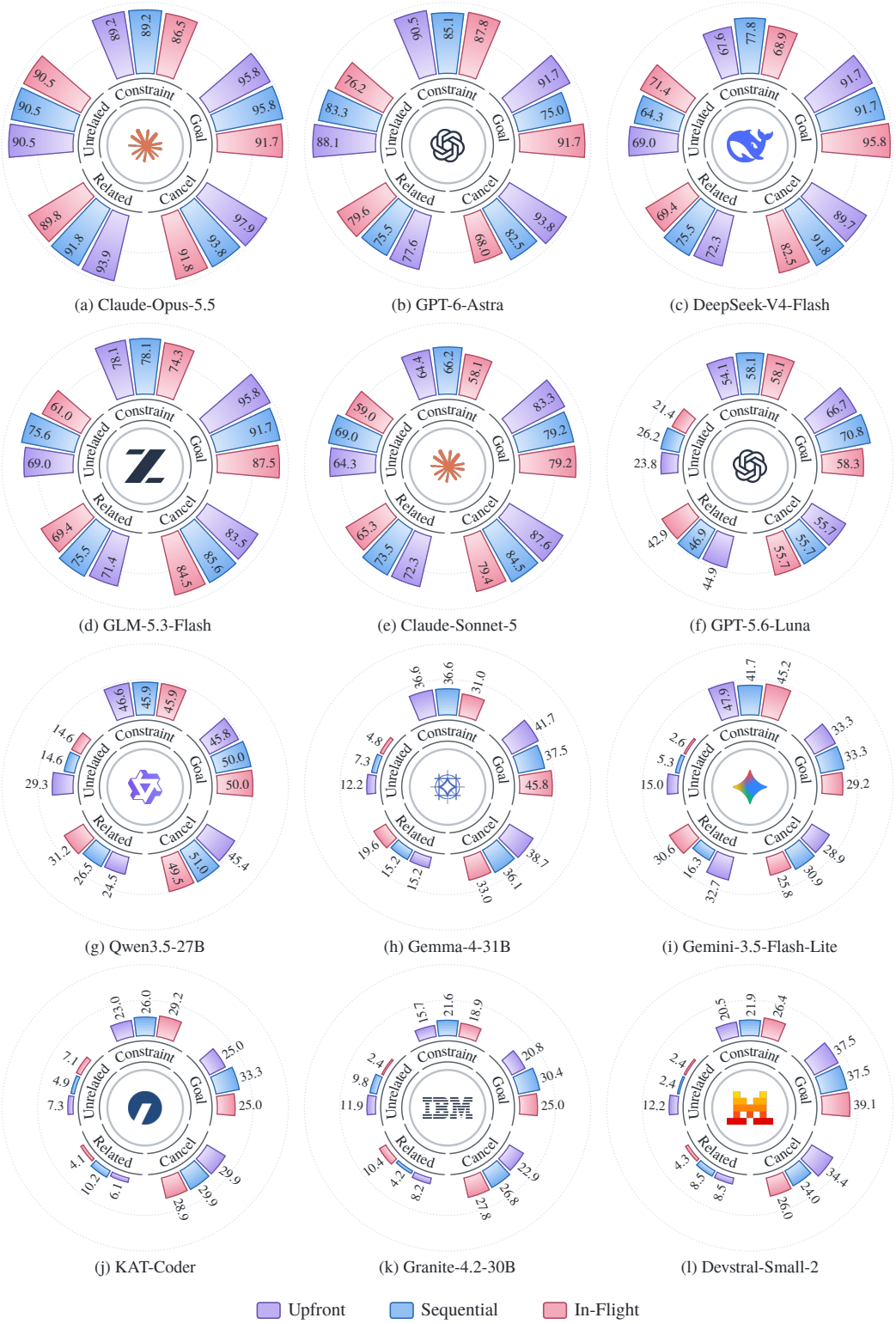


Figure H.8: **Timing effects by event family for the six frontier configurations and the other 13.** Each bar shows the pooled change from SEQUENTIAL to IN-FLIGHT in Resolved (left, points) or output tokens (right, percent), with the five families (CU, GR, CX, NR, NU) grouped into the three change types. The frontier configurations lose 3.0 to 5.3 points in every family except goal revisions, whereas the other 13 stay within 1.3 points of zero.



(a) Claude-Opus-5.5 (b) GPT-6-Astra (c) DeepSeek-V4-Flash

(d) GLM-5.3-Flash (e) Claude-Sonnet-5 (f) GPT-5.6-Luna

(g) Qwen3.5-27B (h) Gemma-4-31B (i) Gemini-3.5-Flash-Lite

(j) KAT-Coder (k) Granite-4.2-30B (l) Devstral-Small-2

Figure H.9: **Resolved rate by event family for every configuration.** Each panel shows one configuration with its UPFRONT, SEQUENTIAL, and IN-FLIGHT Resolved rates in percent on a common radial scale from 0 to 100, and panels follow SEQUENTIAL Resolved from the strongest configuration. Constraint, Goal, Cancel, Related, and Unrelated stand for constraint updates, goal revisions, cancellations with replacement, and related and unrelated additional tasks. Panels (m) to (s) continue in Fig. H.10.

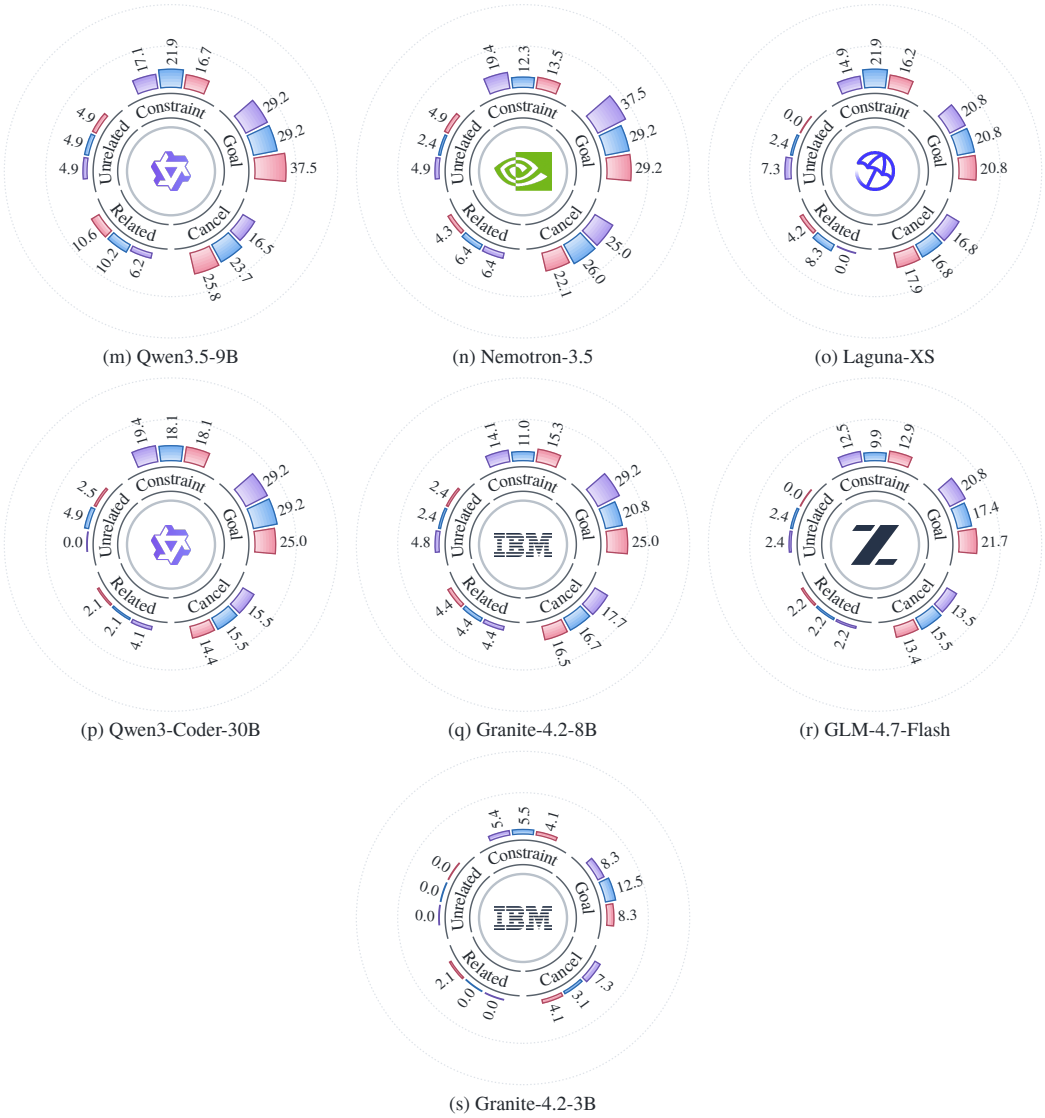


Figure H.10: **Resolved rate by event family for every configuration, continued.** Panels (m) to (s) complete Fig. H.9 with the same scale and colors.

Table H.7: **In-Flight correctness by request-change type.** TPR and Resolved are percentages (↑). Additional-task Resolved requires satisfying the combined final request. Each Resolved value sits on a bar proportional to it. Bold marks the best value in a column within each model group and underlining the second best.

Model	Requirement change		Cancel / replace		Additional task	
	TPR ↑	Resolved ↑	TPR ↑	Resolved ↑	TPR ↑	Resolved ↑
Open-weight models						
Granite-4.2-3B	7.3	5.2	4.1	4.1	5.0	1.1
GLM-4.7-Flash	31.8	15.1	17.7	13.4	21.5	1.1
Qwen3-Coder-30B	41.8	19.8	29.4	14.4	29.7	2.3
Laguna-XS	25.9	17.3	24.6	17.9	19.0	2.2
Granite-4.2-8B	35.7	17.7	20.8	16.5	18.0	3.4
Nemotron-3.5	33.3	17.3	31.9	22.1	21.6	4.5
Granite-4.2-30B	38.7	20.4	34.7	27.8	29.8	6.7
Qwen3.5-9B	42.2	21.9	33.2	25.8	33.4	8.0
Devstral-Small-2	55.4	29.5	32.5	26.0	33.6	3.4
KAT-Coder	40.4	28.1	35.9	28.9	20.4	5.5
Gemma-4-31B	61.8	34.7	45.2	33.0	41.6	12.5
Qwen3.5-27B	**66.5**	**46.9**	**57.3**	**49.5**	**56.7**	**23.6**
API models						
Gemini-3.5-Flash-Lite	55.9	41.2	27.1	25.8	39.9	18.4
GPT-5.6-Luna	76.2	58.2	64.7	55.7	62.1	33.0
Claude-Sonnet-5	86.6	63.3	85.6	79.4	86.5	62.5
GLM-5.3-Flash	89.2	77.6	91.2	84.5	86.5	65.6
DeepSeek-V4-Flash	89.6	75.5	89.5	82.5	92.1	70.3
GPT-6-Astra	95.2	**88.8**	72.8	68.0	91.0	78.0
Claude-Opus-5.5	**95.5**	87.8	**99.0**	**91.8**	**99.2**	**90.1**

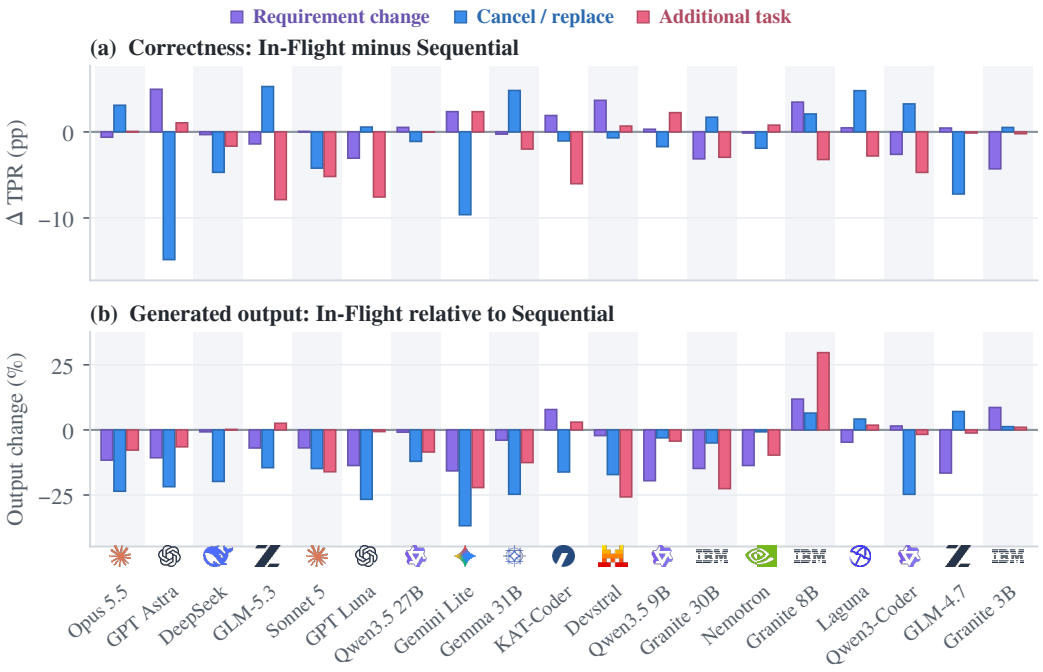


Figure H.11: **Paired correctness and generation changes by request type.** (a) Mean within-instance TPR change, In-Flight minus Sequential, in percentage points. (b) Percentage change in mean generated output on the same pairs. Each group of bars is one configuration, ordered by In-Flight Resolved. Violet, blue, and pink denote requirement change, cancellation with replacement, and additional task. These descriptive means are instance-weighted. Overall source-balanced contrasts and uncertainty are reported in Tables H.9 and H.10.

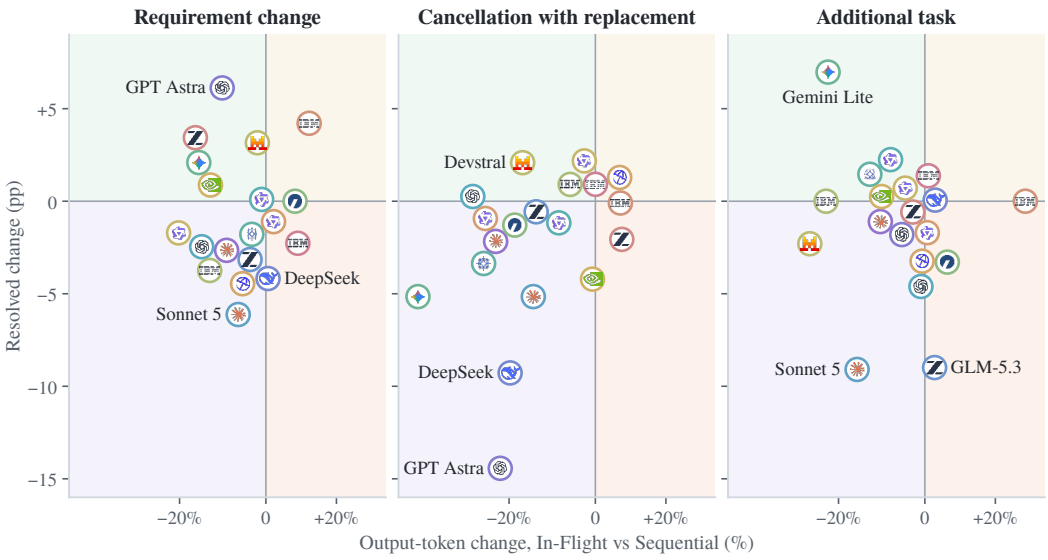


Figure H.12: **Token change against Resolved change by request type.** Each ring is one configuration, with the same axes as Fig. 5a. Cancellation with replacement saves the most tokens and contains the largest single drop in Resolved.

H.3 Variation across model families and codebases

The request-type analysis describes which changes are difficult. We next examine whether the overall correctness pattern is consistent within model families and across the codebases on which they work. Figure H.13 compares the evaluated Granite-4.2 and Qwen3.5 configurations across three correctness measures. All three measures increase from 3B to 8B to 30B within Granite-4.2. Within Qwen3.5, the 27B configuration also improves on the 9B configuration in TPR, Resolved, and target completion. These observations connect partial and complete success within each family, although the configurations do not constitute a controlled scaling experiment.

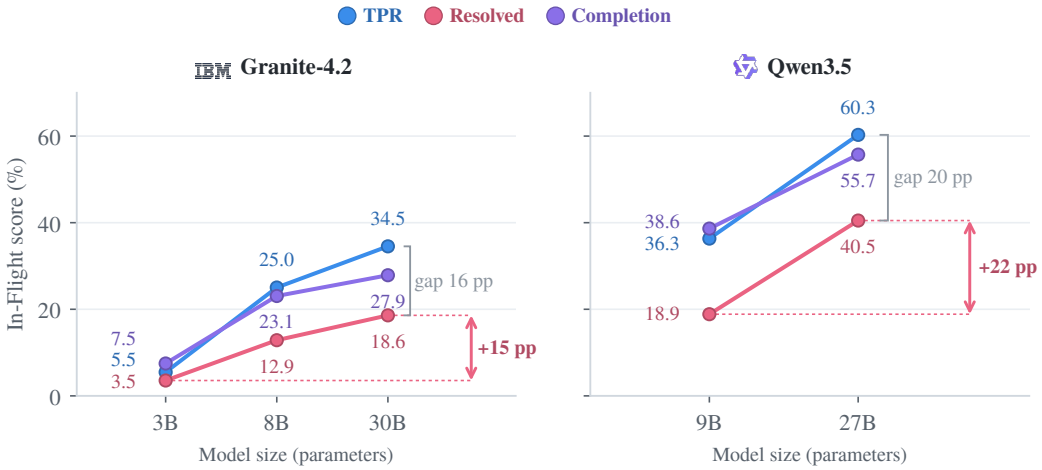


Figure H.13: **Within-family correctness profiles.** Lines connect In-Flight scores at the evaluated parameter sizes within Granite-4.2 and Qwen3.5. Markers show the actual observations. Connecting segments guide the eye and do not estimate performance at unmeasured sizes. TPR and Resolved use scored instances. Target completion uses the explicitly eligible, observed subset reported in Table H.2. Values are printed beside each marker. The pink arrow marks the rise in Resolved from the smallest to the largest size and the grey bracket the TPR–Resolved gap at the largest size.

Language and source profiles. To examine the data behind these model-level patterns, Table H.8 breaks down In-Flight TPR across the five Core languages and the three sources. Each column describes its own constructed subset, whose codebase and task composition can differ. Reading these profiles together shows whether strong aggregate performance extends across the evaluated codebases.

Qwen3.5-27B has the highest locally hosted TPR in all five language columns and all three source columns. Among API models, the highest source-level TPR belongs to GPT-6-Astra on controlled tasks and to Claude-Opus-5.5 on real-repository and external-derived tasks.

Finally, Figure H.14 provides a compact view of how language profiles compare across delivery conditions. We show the highest In-Flight Resolved configuration in each model group, using the same five language subsets and percentage range in both panels. The profiles summarize the observed language subsets. They do not replace the paired timing comparisons below.

Table H.8: **In-Flight test pass rate by language and by source.** TPR is in percent (↑). Language and source columns retain their own task subsets. Light blue bars use the same 0 to 100% scale. Bold marks the best score within each model group and column.

Model	*Language*					*Source*		
	Python	Go	TypeScript	JavaScript	Rust	Controlled	Real repository	External-derived
Open-weight models								
Granite-4.2-3B	3.8	5.1	6.4	8.3	5.0	3.1	3.0	7.0
GLM-4.7-Flash	28.6	19.5	23.9	25.0	10.3	34.5	15.6	22.5
Qwen3-Coder-30B	43.7	21.8	36.2	33.1	23.9	53.7	23.5	30.0
Laguna-XS	25.7	17.2	24.8	30.9	10.5	23.7	13.2	25.8
Granite-4.2-8B	35.0	19.0	22.2	23.0	11.8	37.2	16.4	23.4
Nemotron-3.5	36.1	23.3	26.0	34.1	15.2	31.8	11.5	33.0
Granite-4.2-30B	51.4	24.6	25.9	33.8	18.6	47.2	28.4	31.8
Qwen3.5-9B	41.4	27.7	44.8	37.1	16.9	40.6	25.0	37.9
Devstral-Small-2	55.1	34.2	38.2	29.8	28.1	60.3	45.2	32.9
KAT-Coder	41.7	24.9	29.9	38.3	10.0	32.2	24.9	34.6
Gemma-4-31B	62.4	39.9	48.6	50.5	24.2	64.4	41.5	47.1
Qwen3.5-27B	**71.4**	**49.1**	**66.8**	**56.1**	**40.4**	**83.7**	**47.2**	**55.9**
API models								
Gemini-3.5-Flash-Lite	58.3	31.0	37.8	30.6	29.2	77.9	30.6	30.8
GPT-5.6-Luna	82.3	59.3	66.9	62.1	48.7	82.0	69.8	62.3
Claude-Sonnet-5	90.6	84.8	83.0	88.5	73.4	82.1	96.3	85.0
GLM-5.3-Flash	91.6	84.8	91.3	95.2	70.1	88.6	88.8	89.2
DeepSeek-V4-Flash	92.4	93.3	88.1	91.3	74.5	85.0	95.8	90.8
GPT-6-Astra	91.5	85.8	81.2	86.1	78.4	**94.3**	92.7	81.7
Claude-Opus-5.5	**99.2**	**98.3**	**95.2**	**100.0**	**92.6**	93.9	**100.0**	**98.7**

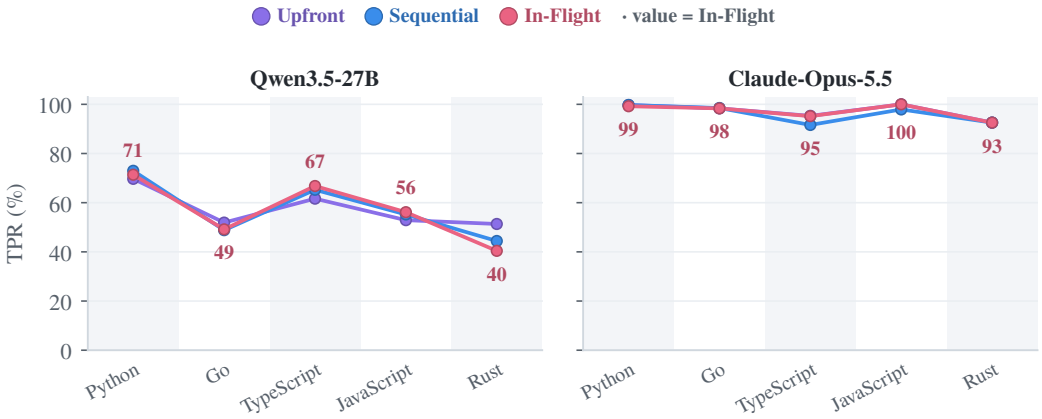


Figure H.14: **Language profiles across delivery conditions.** Each panel reports TPR on the five Core language subsets for one configuration, with one line per delivery condition (violet Upfront, blue Sequential, pink In-Flight) and the In-Flight value printed. Qwen3.5-27B and Claude-Opus-5.5 are the highest In-Flight Resolved configurations in their model groups. Scores use each condition's observed instances.

H.4 Paired changes and solved-instance overlap

The preceding breakdowns describe performance on different task subsets. To examine timing on the same tasks, we next match interaction instances within each model. Tables H.9 and H.10 report source-balanced mean changes with lower and upper confidence limits. Matching controls which tasks enter each comparison, while equal weighting of the three sources prevents the largest source from dominating the average contrast. We then examine solved-instance overlap and compare the timing contrasts across request types.

For In-Flight versus Sequential, the Resolved point estimate is positive for 5 models, negative for 12, and 0.0 after rounding for 2. The pointwise 95% interval excludes zero only for Claude-Sonnet-5, whose contrast is negative. However, a small average contrast can conceal a change in which requests are solved. All 19 models have both Sequential-only and In-Flight-only successes. Table H.11 therefore complements the mean contrasts by reporting solved-instance overlap and retention of Sequential successes.

Table H.9: **Paired In-Flight minus Sequential contrasts.** Estimates and pointwise 95% confidence limits are in percentage points. Positive favors In-Flight. Contrasts match the same instances, weight the three source-specific mean changes equally, and use 10,000 source-task-cluster bootstrap resamples. Estimate cells are tinted rose for negative and mint for positive changes, with intensity proportional to the magnitude up to 8 pp.

Model	Resolved difference (pp)			TPR difference (pp)		
	Estimate	Lower	Upper	Estimate	Lower	Upper
Open-weight models						
Granite-4.2-3B	+0.0	-1.1	+1.0	-0.6	-2.9	+1.5
GLM-4.7-Flash	+1.0	-1.0	+3.4	-1.5	-6.0	+2.7
Qwen3-Coder-30B	+0.0	-3.8	+4.0	-0.8	-5.1	+3.5
Laguna-XS	-2.9	-6.5	+0.3	+1.1	-2.6	+4.8
Granite-4.2-8B	+1.8	-1.3	+5.1	+1.0	-3.3	+5.5
Nemotron-3.5	-0.1	-2.2	+2.3	+0.4	-3.3	+4.1
Granite-4.2-30B	-0.9	-5.6	+3.7	-1.3	-5.9	+3.3
Qwen3.5-9B	-2.0	-6.7	+2.6	-1.1	-5.9	+3.6
Devstral-Small-2	+1.2	-4.3	+6.7	+2.7	-2.7	+8.1
KAT-Coder	-2.3	-7.0	+2.1	-2.5	-7.8	+2.9
Gemma-4-31B	-0.4	-4.1	+3.4	-0.2	-3.5	+3.2
Qwen3.5-27B	+0.2	-4.2	+4.5	-0.3	-4.4	+3.8
API models						
Gemini-3.5-Flash-Lite	+5.4	-0.2	+11.0	+1.1	-3.3	+5.5
GPT-5.6-Luna	-1.8	-8.3	+4.5	-2.8	-7.1	+1.3
Claude-Sonnet-5	-8.5	-13.5	-3.6	-2.9	-5.3	-0.6
GLM-5.3-Flash	-4.2	-10.0	+1.7	-1.7	-5.0	+1.4
DeepSeek-V4-Flash	-4.2	-9.7	+1.2	-2.2	-4.6	+0.4
GPT-6-Astra	-2.2	-7.0	+2.5	-1.7	-4.9	+1.5
Claude-Opus-5.5	-1.1	-3.9	+1.6	+0.3	-0.5	+1.2

Table H.10: **Paired In-Flight minus Upfront contrasts.** Estimates and pointwise 95% confidence limits are in percentage points. Positive favors In-Flight. Contrasts match the same instances, weight the three source-specific mean changes equally, and use 10,000 source-task-cluster bootstrap resamples. Estimate cells are tinted rose for negative and mint for positive changes, with intensity proportional to the magnitude up to 8 pp.

Model	Resolved difference (pp)			TPR difference (pp)		
	Estimate	Lower	Upper	Estimate	Lower	Upper
Open-weight models						
Granite-4.2-3B	-0.6	-1.4	+0.2	-1.6	-3.4	+0.0
GLM-4.7-Flash	-0.7	-4.1	+2.7	+0.5	-3.9	+4.8
Qwen3-Coder-30B	+0.2	-4.0	+4.4	+0.2	-4.4	+4.9
Laguna-XS	+1.1	-2.2	+4.5	+6.8	+2.7	+11.2
Granite-4.2-8B	+0.1	-2.7	+3.1	-0.9	-4.9	+3.2
Nemotron-3.5	-2.6	-6.4	+1.2	-2.5	-7.3	+2.2
Granite-4.2-30B	+3.0	-2.4	+8.7	+1.0	-4.7	+6.8
Qwen3.5-9B	+3.1	-0.4	+6.7	+4.5	-0.2	+9.4
Devstral-Small-2	-3.8	-9.8	+2.3	-1.8	-7.2	+3.6
KAT-Coder	-1.5	-7.2	+4.1	+0.7	-5.2	+6.5
Gemma-4-31B	-3.2	-8.8	+2.6	-1.5	-6.2	+3.4
Qwen3.5-27B	+0.1	-6.9	+7.0	-0.7	-5.8	+4.3
API models						
Gemini-3.5-Flash-Lite	-3.6	-10.0	+2.7	-3.1	-8.1	+1.9
GPT-5.6-Luna	+0.5	-6.5	+7.6	+2.2	-3.0	+7.2
Claude-Sonnet-5	-5.8	-11.2	-0.7	-0.4	-3.0	+2.5
GLM-5.3-Flash	-3.8	-9.8	+2.0	+1.2	-2.7	+5.1
DeepSeek-V4-Flash	-0.6	-6.4	+5.1	-1.5	-4.0	+1.0
GPT-6-Astra	-7.0	-10.9	-3.4	-6.5	-9.1	-4.1
Claude-Opus-5.5	-3.1	-6.3	-0.2	-0.2	-0.9	+0.1

Table H.11: **Outcome overlap across Sequential and In-Flight.** The first four columns partition jointly scored Sequential/In-Flight pairs by complete success. Retention is the percentage of Sequential-resolved instances also resolved In-Flight. Pink and blue bars show Seq.-only and IF-only shares of all pairs. Yellow bars show retention on a percentage scale. Counts reveal when similar aggregate rates come from different solved instances.

Model	Both resolved	Seq. only	IF only	Neither	Retention ↑ (%)
Open-weight models					
Granite-4.2-3B	6	4	4	267	60.0
GLM-4.7-Flash	22	6	6	241	78.6
Qwen3-Coder-30B	24	14	11	229	63.2
Laguna-XS	27	15	9	228	64.3
Granite-4.2-8B	23	9	13	230	71.9
Nemotron-3.5	32	13	10	224	71.1
Granite-4.2-30B	38	17	15	213	69.1
Qwen3.5-9B	35	18	18	210	66.0
Devstral-Small-2	38	15	18	207	71.7
KAT-Coder	46	17	13	207	73.0
Gemma-4-31B	61	19	15	183	76.2
Qwen3.5-27B	99	15	16	153	86.8
API models					
Gemini-3.5-Flash-Lite	55	23	26	175	70.5
GPT-5.6-Luna	119	29	22	116	80.4
Claude-Sonnet-5	185	28	9	61	86.9
GLM-5.3-Flash	200	29	17	37	87.3
DeepSeek-V4-Flash	203	28	15	38	87.9
GPT-6-Astra	204	29	20	33	87.6
Claude-Opus-5.5	253	10	4	19	96.2

Comparing timing effects across request types. Solved-instance overlap describes how outcomes change within a model. We also compare whether the timing contrast differs across request types. For this comparison, we use the same matched score-and-telemetry episodes for correctness and resources, average source categories equally within each model, and then average all 19 models equally. A shared source-task bootstrap preserves the dependence created by evaluating the same source tasks with multiple configurations. Requirement change and additional task share all three sources. Comparisons involving cancellation/replacement use only the two repository-derived sources present in both types.

Across all three sources, the requirement-change minus additional-task difference in paired Macro TPR is $+2.40$ percentage points, with a pointwise 95% interval of $[+0.43, +4.28]$ (Table H.12). The replacement-minus-addition comparison over their common sources is $+1.25$ points, with an interval of $[-1.74, +4.20]$. These are direct differences between timing contrasts, rather than a comparison of whether separate within-type intervals include zero. They support a more differentiated account of request timing than a single average over all task structures.

Generated-output ratios tell a different part of the story. Let r be the mean In-Flight output divided by the mean Sequential output on a type's matched cohort, with the same source and model weighting. All three direct differences in r have intervals spanning zero. Consequently, the observed requirement-change/addition difference in test correctness is not matched by clear evidence of a type-specific difference in relative generated volume. These exploratory intervals remain pointwise. They do not constitute a multiplicity-adjusted search for significant subgroups.

Table H.12: **Direct differences between request-type timing effects.** Each row subtracts two In-Flight–Sequential effects, with equal model and common-source weighting. TPR differences are percentage points. Output uses the dimensionless ratio $r = \overline{\text{tokens}}_{\text{IF}}/\overline{\text{tokens}}_{\text{Seq}}$. Lower and Upper are pointwise 95% shared source-task-cluster bootstrap limits from 4,000 draws. A–C uses three sources. A–B and B–C use the two common repository-derived sources. A denotes requirement change, B cancellation/replacement, and C additional task. Values therefore differ slightly from the instance-weighted estimates in Section 4.4.

Type contrast	Measure	Estimate	Lower	Upper
A−C	Δ TPR (pp)	+2.40	+0.43	+4.28
A−C	Output-ratio difference	+0.018	-0.078	+0.113
B−C	Δ TPR (pp)	+1.25	-1.74	+4.20
B−C	Output-ratio difference	+0.010	-0.066	+0.086
A−B	Δ TPR (pp)	+2.03	-0.81	+4.74
A−B	Output-ratio difference	+0.062	-0.052	+0.171

H.5 Taking up and completing an unrelated task

The paired comparisons show that delivery conditions can solve different requests. For unrelated additions, we can further locate the difficulty between taking up the new request and completing both obligations. The agent must work in a second repository while retaining responsibility for the first. Table H.13 separates beginning the added work, passing each source task, and completing both. This distinguishes failure to start the added request from an unsuccessful implementation, while showing whether the original task is completed.

For most configurations, beginning the second task is much more common than completing it. GPT-5.6-Luna takes an attributed action on the added task in all observed episodes, but passes that task in 28.6% and both tasks in 21.4%. Among open-weight models, Qwen3.5-27B has the highest added-task and joint success rates (both 14.6%), although Gemma-4-31B begins the added task more often (78.6% versus 73.2%). These contrasts show why activity and success need separate measurements.

Table H.13: **Activity and completion on unrelated additions.** Percentages use a common In-Flight cohort per model with observed action attribution and both outcomes (Table H.2). Started requires an action attributed to the added task. The pass columns require complete source-task success, and Both requires success on the combined request. Each value sits on a bar proportional to it. Bold marks the best value in a column within each model group and underlining the second best.

Model	Added task started ↑	Original task passed ↑	Added task passed ↑	Both tasks passed ↑
Open-weight models				
Granite-4.2-3B	9.8	7.3	0.0	0.0
GLM-4.7-Flash	59.5	21.4	2.4	0.0
Qwen3-Coder-30B	70.0	20.0	5.0	2.5
Laguna-XS	39.0	34.1	0.0	0.0
Granite-4.2-8B	23.8	14.3	4.8	2.4
Nemotron-3.5	34.1	26.8	4.9	4.9
Granite-4.2-30B	31.0	26.2	4.8	2.4
Qwen3.5-9B	51.2	36.6	12.2	4.9
Devstral-Small-2	26.8	31.7	2.4	2.4
KAT-Coder	47.6	31.0	7.1	7.1
Gemma-4-31B	**78.6**	40.5	11.9	4.8
Qwen3.5-27B	73.2	**56.1**	**14.6**	**14.6**
API models				
Gemini-3.5-Flash-Lite	76.3	44.7	10.5	2.6
GPT-5.6-Luna	**100.0**	61.9	28.6	21.4
Claude-Sonnet-5	97.4	84.6	61.5	59.0
GLM-5.3-Flash	92.7	82.9	65.9	61.0
DeepSeek-V4-Flash	95.2	83.3	76.2	71.4
GPT-6-Astra	95.2	**95.2**	76.2	76.2
Claude-Opus-5.5	100.0	92.9	**97.6**	**90.5**

I Budget and Message Controls

Request timing changes both the remaining opportunity to work and the moment at which a user message appears. We examine these two aspects with separate controls. The first varies the post-change action allowance and traces how the added actions are used. The second introduces a check-in that leaves the objective unchanged.

I.1 Remaining budget and use of additional actions

The main comparison changes when a request reaches the agent. That timing also determines how much work can be performed afterwards. We therefore evaluate a crossed control in which Sequential and In-Flight each receive either 250 steps in total or 250 steps after the update. This design keeps the final requirements fixed while varying the allowance available for responding to them.

Specifically, the control covers 23 instances per model, chosen before examining outcomes from the populated source-by-change-type cells. Qwen3.5-27B and GLM-4.7-Flash, both with reasoning disabled, each execute the four settings, giving 184 runs. We keep the frozen progress-based and budget-based arrival schedules separate. Qwen has 12 and 11 instances in these strata, respectively. GLM has 2 and 21. The main panels show the three larger groups, with the two remaining GLM cases described below.

The extra allowance primarily changes Sequential performance in the budget-based groups (Fig. I.1). For Qwen, its mean TPR rises from 34.3% to 48.2%, while In-Flight remains close to 48%. Consequently, the In-Flight-minus-Sequential difference moves from $+14.0$ to -0.4 percentage points. GLM shows the same direction in the point estimates, from $+9.1$ to -0.8 points. The point estimates indicate a contribution of the delivery protocol: giving an unfinished trajectory an earlier update also gives it more opportunities to implement that update. The paired changes in the condition

contrast have intervals that include zero, so the control describes these profiles without establishing a population-wide budget interaction.

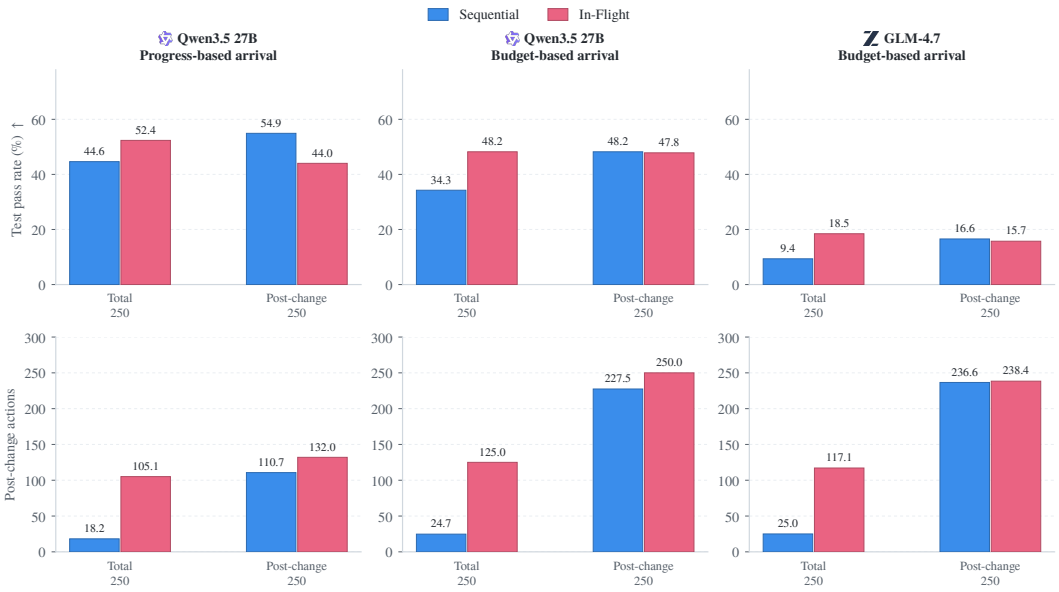


Figure I.1: **Correctness and post-change work under two action allowances.** Bars compare blue Sequential and pink In-Flight within each budget protocol. Top: final Macro TPR (↑). Bottom: post-change actions over episodes receiving the update. Each configuration begins from the initial repository state. The cohort sizes are given in the text. The paired comparisons appear in Table I.1.

Table I.1: **Paired TPR differences in the budget control.** Total and post-change columns are In-Flight minus Sequential (percentage points). The last column subtracts the total-budget contrast from the post-change contrast. Each row uses the same selected instances across the four settings.

Model	Arrival	250 total	250 after change	Difference
Qwen3.5-27B	Progress-based	+7.7	-10.9	-18.6
Qwen3.5-27B	Budget-based	+14.0	-0.4	-14.3
GLM-4.7-Flash	Budget-based	+9.1	-0.8	-9.9

For completeness, the two progress-based GLM cases retain the same qualitative ordering under both allowances. The constraint-update case has Sequential/In-Flight TPR of 50.0/0.0% under the total budget and 33.3/0.0% under the post-change budget. The related-addition case has 84.6/38.5% and 69.2/38.5%, respectively. These cases remain distinct from the budget-based group because their update schedules follow a different rule.

How trajectories use the allowance. An expanded action allowance is useful when the added steps produce new information or implementation progress. To examine how trajectories use that allowance, we measure the share of exact repeated action–result pairs. An action is matched without its thought field, and tool results are matched after removing harness annotations. This measurement describes repeated behavior directly. Correctness continues to be determined by the final tests.

Figure I.2 shows the full per-instance distributions, while Table I.2 places them beside action and generation counts. The substantial variation within each setting explains why a single mean step count cannot describe how the budget is used. Reading resource use together with repetition and final correctness therefore gives a more informative account of the budget control than treating every extra action as useful work.

I.2 No-change message control

The budget control concerns the time available to react. A complementary control asks whether simply receiving another user message changes the result even when that message introduces no

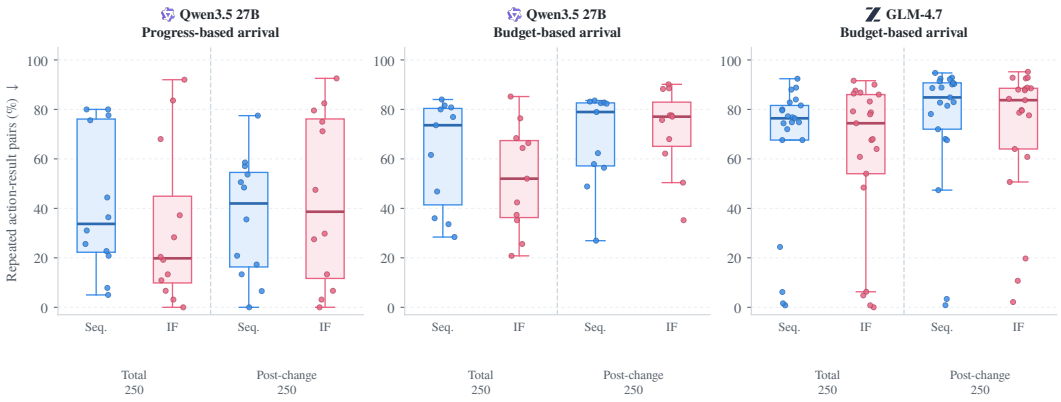


Figure I.2: **Repeated behavior varies within the same action allowance.** Points represent individual trajectories. Boxes show medians and interquartile ranges, with 1.5-IQR whiskers. The metric is the fraction of exact repeated action–result pairs (↓). Blue Sequential and pink In-Flight denote the delivery condition, and the two groups of boxes distinguish the allowance protocols.

Table I.2: **Correctness, generation, and repetition in the budget control.** TPR, token counts, and repetition use all assigned panel outcomes. Post-change actions use delivered episodes. Generated output is in thousands. Lower consumption is interpreted jointly with correctness. Blue Sequential and pink In-Flight bars show TPR on a common percentage scale.

Condition	Allowance	TPR ↑	Post actions	Tokens ↓	Repeat ↓
Qwen3.5-27B / Progress-based arrival					
Sequential	Total	44.6	18.2	11.8	42.3
Sequential	After change	54.9	110.7	21.1	36.6
In-Flight	Total	52.4	105.1	10.5	31.9
In-Flight	After change	44.0	132.0	10.2	44.1
Qwen3.5-27B / Budget-based arrival					
Sequential	Total	34.3	24.7	18.7	62.1
Sequential	After change	48.2	227.5	33.8	67.8
In-Flight	Total	48.2	125.0	18.9	52.2
In-Flight	After change	47.8	250.0	24.7	71.9
GLM-4.7-Flash / Budget-based arrival					
Sequential	Total	9.4	25.0	12.4	65.3
Sequential	After change	16.6	236.6	16.6	75.3
In-Flight	Total	18.5	117.1	7.6	61.7
In-Flight	After change	15.7	238.4	10.9	71.7

new objective. For GLM-4.7-Flash with reasoning disabled, we add a check-in to the Sequential protocol that states there are no changes and asks the agent to continue. The message is scheduled at the In-Flight arrival step. The actual request change still arrives at the Sequential boundary.

This study uses a fixed hash selection of 40 instances per change type, giving 120 three-condition comparisons and 360 trajectories. Seventeen instances use progress-based arrival and 103 use budget-based arrival. A check-in is inserted while the initial-work phase is active. Outcomes retain the assigned instances even when the update has already occurred before that point. Thus the control measures the scheduled message intervention with the same final objective.

In the larger budget-based group, check-in minus Sequential is -1.1 TPR points and $+1.8$ steps. In-Flight minus check-in is -1.6 points and -2.8 steps. The corresponding intervals span zero (Fig. I.3). These comparisons place the no-change message alongside the substantive revision rather than assuming that every user message has the same effect. Together with the budget experiment, they separate two mechanisms that would otherwise be combined in a single interruption score.

Estimation for the controls. We finally state how both controls are estimated. The controls use 10,000 source-task-cluster bootstrap resamples within each arrival stratum. Qwen's budget panel has valid scores in all four settings. The registered GLM panel rule assigns zero to two unscorable

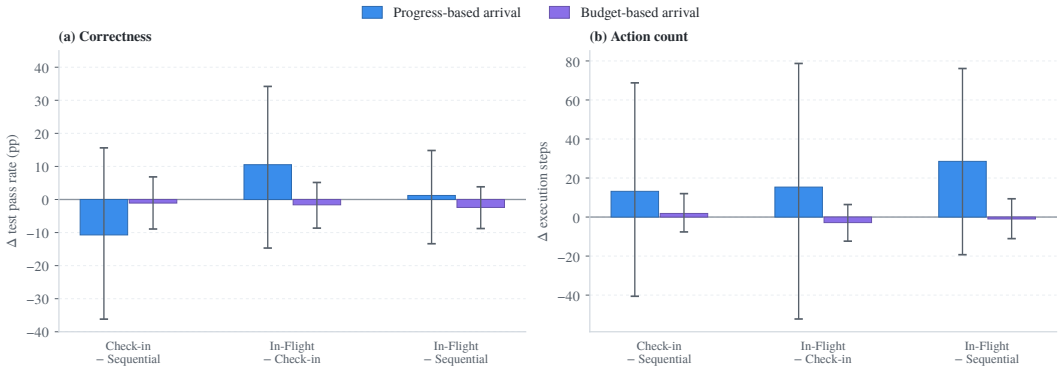


Figure I.3: **No-change messages and revised requests are evaluated separately.** GLM-4.7-Flash OFF. Vertical bars show paired differences for the three indicated contrasts. Error bars are pointwise 95% source-task-cluster bootstrap intervals. Blue progress-based and violet budget-based arrival are separated throughout. Each correctness contrast uses jointly scorable pairs. Action comparisons use completed pairs.

endpoints of one budget-based instance. Retaining score-valid cases gives 20 total-budget pairs and 21 post-change pairs, with the same direction of the changes above. The released supporting statistics retain every interval, cohort count, and sensitivity estimate. Post-change resource means are restricted to delivered episodes, while final outcomes retain the assigned panel population.

J Comparison with Native Codex

The main experiments hold the agent harness fixed to compare when a revised request arrives. To examine how this choice affects the observed results, we evaluate GPT-5.6-Luna through native Codex and compare it with the same model in our benchmark harness on matched instances. We first report overall correctness and the timing contrast in each harness, and then examine where the two systems differ by request type. The comparison retains the original pilot sample and adds instances selected by a fixed hash within task-family and source strata, without selecting for correctness or agreement between harnesses. Both systems receive the same task requirements and are scored against the same final acceptance tests. Native runs use Codex CLI 0.154.0 with reasoning effort set to `none`, a limit of 250 native tool actions, and a 5,400-second time limit. Sequential delivers the revision after the original request is declared complete. In-Flight uses the corresponding harness's Sequential calibration and the benchmark's fallback rule. Tool interfaces, prompts, context defaults, and action granularity therefore remain properties of each system.

Table J.1: **Same-model comparison with native Codex.** GPT-5.6-Luna is evaluated on the same $n = 120$ instances in each harness–condition cell, with 40 instances per change type. The supplementary actual-mid-execution intersection has $n = 105$. Macro TPR and Resolved are percentages. Actions and Output are per-episode means, with output tokens in thousands. Actions follow each harness's own tool-call definition. Model-caused failures remain scored outcomes, including partial test passes.

Harness	Condition	Macro TPR ↑	Resolved ↑	Actions	Output (k) ↓
Custom	Sequential	67.6	47.5	31.6	2.07
	In-Flight	62.1	45.8	27.1	1.75
Native Codex	Sequential	78.9	58.3	12.9	2.80
	In-Flight	76.5	57.5	11.5	2.58

Table J.1 separates absolute performance from the change associated with request timing. Native Codex resolves more instances in both conditions and has higher Macro TPR, while producing more output tokens. Moving from Sequential to In-Flight changes Resolved by -1.7 percentage points in the custom harness and -0.8 points in native Codex. The corresponding Macro TPR changes are -5.5 and -2.4 points. The difference between the two Resolved timing contrasts is $+0.8$ points, with a pointwise 95% source-task-cluster bootstrap interval of $[-10.8, 13.0]$. These estimates are

compatible with appreciable differences in either direction and do not establish equivalence. Restricting the comparison to instances where both systems actually received the In-Flight event during execution gives a -2.9-point Resolved change in each harness. This restriction is a supplementary trajectory-defined analysis. The primary comparison retains all preselected instances. Equal weighting of source categories also leaves both timing estimates near zero (-2.3 and $+0.3$ points).

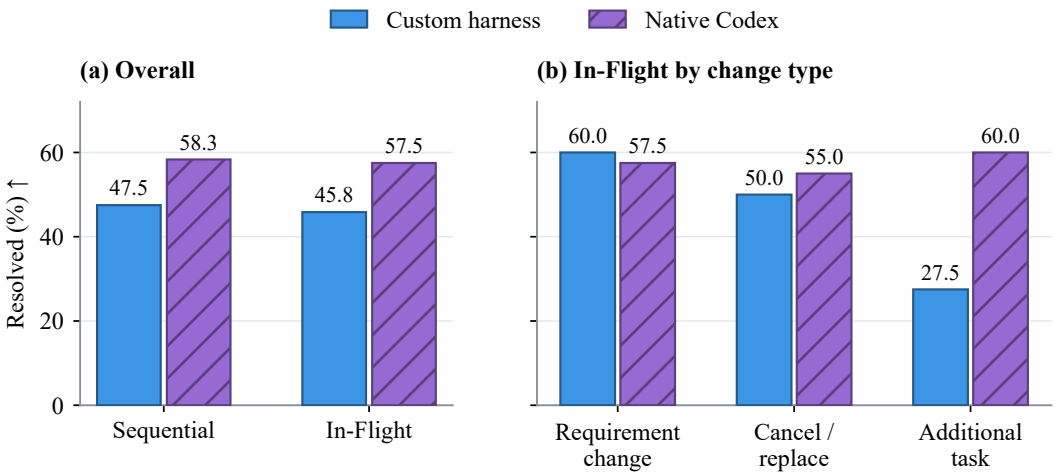


Figure J.1: **Absolute performance and task-type patterns depend on the harness.** GPT-5.6-Luna Resolved rates on the matched sample in Table J.1: (a) overall performance under Sequential and In-Flight, and (b) In-Flight performance by change type. Each bar is an instance-equal mean.

The difference between systems is concentrated in additional-task instances (Figure J.1). Under In-Flight, native Codex resolves 60.0% and the custom harness resolves 27.5%, a paired gap of 32.5 points. Requirement-change rates are close (57.5% versus 60.0%), as are cancellation/replacement rates (55.0% versus 50.0%). Thus, the low joint completion rate for additional requests observed in the fixed-harness experiments does not transfer unchanged to native Codex. The present comparison identifies a system-dependent performance pattern. Attributing it to workspace navigation, tool design, or another mechanism would require a common coding of the two systems' trajectories. Native and custom actions also reflect different tool interfaces, so their counts do not establish a directly comparable reduction in execution steps.

K Paired Thinking-Mode Experiments

The thinking-mode extension tests whether improved correctness is accompanied by a different use of execution and generation. We first establish the paired cohorts, then examine gains across delivery conditions, resource distributions, and instance-level outcome transitions. Finally, we test whether the gains are specific to In-Flight and examine sensitivity to incomplete observations. The analysis combines eight new model pairs with the existing DeepSeek pair, retaining their different delivery protocols. These comparisons supplement the main configurations in Table H.1.

K.1 Frozen configurations and paired evaluation

The eight local ON arms each schedule 286 executable instances under three conditions, giving 6,864 planned cells. The final records contain 6,754 completed runs, 90 runs whose environment failed, and 20 excluded cells, with no pending cells. Within each model, the only generation-parameter change is `enable_thinking=false` to `true`. The planned arrival files and budgets are unchanged. GLM-4.7-Flash pairs against its OFF configuration in the main table, with a 16,384-token generation ceiling and 73,728-token server context. The other seven pairs retain 8,192 and 65,536 respectively. The final GLM records additionally classify seven previously unresolved OFF cases as observed model-caused failures, so the denominators here need not match the main table.

Pair construction. Each model is evaluated on instances with the same execution-environment setup in both modes. We use the official-image setup when matched runs are available and the documented substitute setup otherwise. Laguna-XS and Nemotron-3.5 use the latter. This choice is fixed by availability rather than performance. Results are computed separately for each model. Serving topology varies, so elapsed time is not interpreted as an isolated effect of thinking.

Evidence and pairing. We then state how many pairs survive the evidence rule. The final pairing lists contain 6,434 local pairs whose two scorers both returned valid. Applying the paper's evidence rule retains 171 additional pairs with observed model-caused failures, giving 6,605 jointly observed pairs in the full audit across the three conditions before applying the pair-selection rule above. Infrastructure failures and unresolved verdicts remain missing, not zero. All local pairs agree in planned arrival. Eight Granite-4.2-3B OFF rows lack recorded arrival or resource telemetry and are omitted from analyses requiring those fields. DeepSeek's two modes calibrate In-Flight delivery independently: only 31 of its 285 valid In-Flight pairs share the recorded arrival value. Its row therefore compares reasoning together with the corresponding delivery policy, rather than a frozen-arrival toggle alone.

Inference and accounting. Finally, we state how the paired estimates are computed and reported. The extension's primary summaries weight paired instances equally, with source-category-balanced estimates as sensitivity analyses. Intervals resample source-task clusters with paired outcomes retained together (4,000 draws, fixed seed 20260919). Resource-ratio intervals recompute the ratio of means in each draw. These are pointwise, exploratory intervals, without multiplicity adjustment. The recorded thinking flag verifies the requested parameter, not a nonempty reasoning trace in every episode. Total output tokens include thinking, and the vLLM export does not provide a separate reasoning-token count. A recorded zero in that field is not interpreted as zero thought. Granite-4.2-8B was included after more execution capacity became available, and the audit retains its original exclusion for lack of capacity. Final retry checks for two of the Granite configurations found no approved retries left unrun. Neither inclusion nor retry selection was based on obtaining favorable scores.

Figure K.1 places the complete-solution rates beside their output-token ratios. The two measures need not move by comparable amounts: Qwen3.5-9B gains 10.7 percentage points with 3.3 times the output, while Granite-4.2-8B gains 10.3 points with 9.9 times the output.

Table K.1: **Thinking OFF and ON under In-Flight.** Instance-equal means for paired OFF and ON runs of each model. n_{score} counts jointly scored pairs. n_{res} counts pairs with both steps and output recorded. Resolved and TPR are percentages. Output is generated tokens in thousands. Blue OFF and pink ON bars use a 0 to 100% scale for Resolved and a 0 to 150k scale for output.

(a) Correctness on paired scored instances

Model	n_{score}	Resolved (%)		TPR (%)	
		OFF	ON	OFF	ON
GLM-4.7-Flash	136	11.8	27.9	22.8	46.4
Granite-4.2-3B	275	3.6	21.1	5.6	39.2
Granite-4.2-8B	175	15.4	25.7	27.2	42.0
Granite-4.2-30B	168	16.1	26.2	31.1	46.3
KAT-Coder	139	20.9	36.0	31.1	53.6
Qwen3.5-9B	280	18.9	29.6	36.4	50.5
Laguna-XS	205	11.2	43.9	21.6	60.1
Nemotron-3.5	151	19.2	35.1	30.7	53.4
DeepSeek-V4-Flash†	285	76.1	80.0	90.3	88.5

(b) Consumption on paired recorded instances

Model	n_{res}	Steps		Output (k)	
		OFF	ON	OFF	ON
GLM-4.7-Flash	136	202.9	189.1	9.0	37.3
Granite-4.2-3B	267	238.5	169.1	16.2	97.6
Granite-4.2-8B	175	211.3	208.1	13.5	133.9
Granite-4.2-30B	168	202.3	189.6	15.8	76.8
KAT-Coder	139	215.5	185.9	11.4	24.1
Qwen3.5-9B	280	199.7	159.2	16.1	52.8
Laguna-XS	205	246.4	219.3	9.8	69.1
Nemotron-3.5	151	201.2	149.5	12.6	37.1
DeepSeek-V4-Flash†	285	68.6	89.4	10.4	35.7

†DeepSeek uses the same API executor, with independently calibrated arrival schedules. It is not a frozen-arrival local twin.

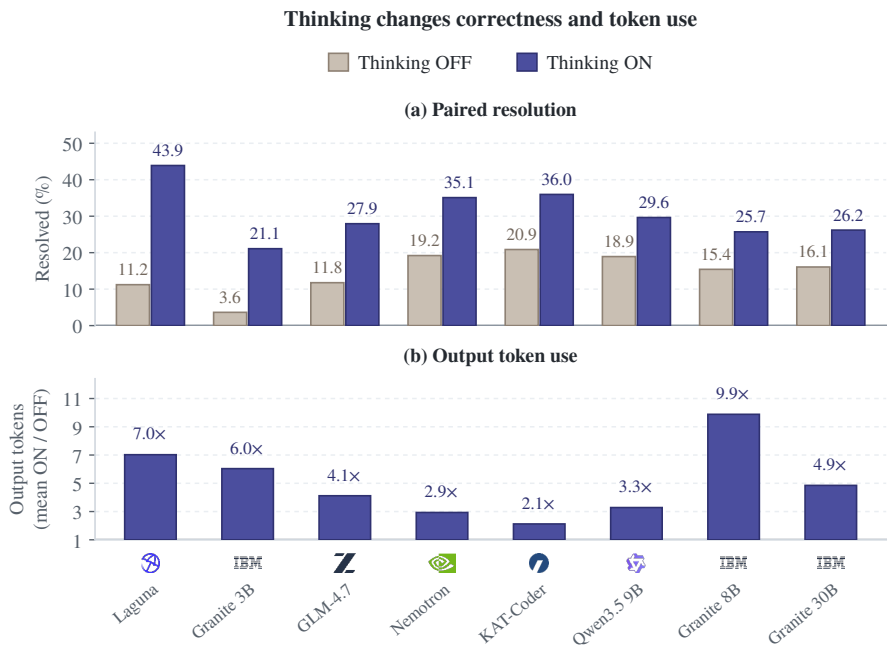


Figure K.1: **Thinking improves complete-solution rates with higher token use.** (a) Bars show paired In-Flight Resolved rates with thinking OFF (grey) and ON (indigo). (b) Bars show the ratio of mean output tokens, starting at the no-change value of one. Output includes thinking tokens. Models are ordered by Resolved gain. Table K.1 reports paired sample sizes, including the separate correctness and token counts for Granite-4.2-3B.

K.2 Correctness and cost across all delivery conditions

Figure K.2 summarizes the paired In-Flight differences in Resolved, TPR, and output tokens.

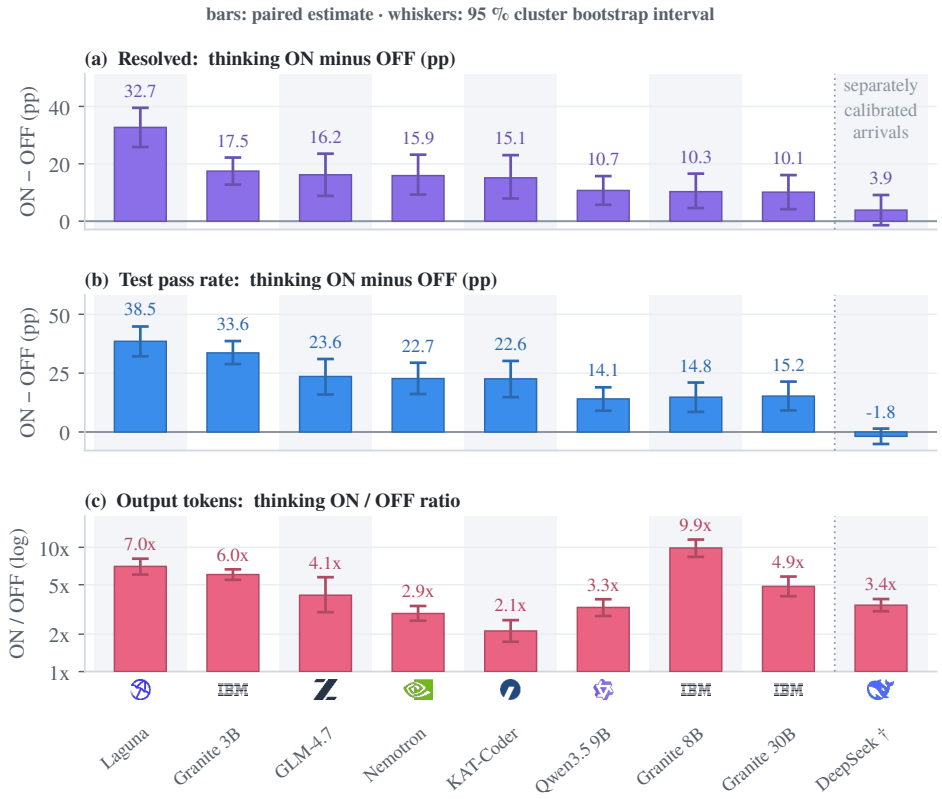


Figure K.2: **Thinking changes correctness and generation cost differently.** Paired In-Flight ON-minus-OFF differences in Resolved (a) and TPR (b), and the ratio of mean output tokens (c, logarithmic), with pointwise 95% source-task-cluster bootstrap intervals (4,000 draws). †DeepSeek uses mode-specific arrival calibration. Paired counts are in Table K.1.

Before attributing a gain to adaptation during execution, we compare the thinking effect under all three delivery conditions. Tables K.2 to K.4 report paired differences and uncertainty. The eight frozen-arrival models improve their absolute Resolved and TPR point estimates under every condition. This consistency motivates the interaction analysis below, which tests whether the improvement under In-Flight is larger than under the reference conditions.

Table K.2: **Paired thinking contrasts under Upfront.** Correctness differences are ON minus OFF in percentage points. Resource contrasts are ratios of ON to OFF means. Brackets show 95% intervals. n_{score} and n_{res} give paired score and resource denominators.

Model	n_{score}	n_{res}	ΔResolved (pp)	ΔTPR (pp)	Steps ON/OFF	Output ON/OFF
GLM-4.7-Flash	139	139	+8.6 [2.2, 15.8]	+13.7 [6.3, 20.9]	0.98 [0.915, 1.062]	4.45 [3.046, 6.308]
Granite-4.2-3B	271	271	+14.4 [10.0, 18.8]	+26.3 [21.4, 31.2]	0.76 [0.715, 0.796]	5.70 [5.020, 6.390]
Granite-4.2-8B	99	99	+13.1 [5.1, 21.2]	+17.6 [9.5, 26.0]	1.04 [0.977, 1.114]	10.14 [8.172, 12.494]
Granite-4.2-30B	115	115	+12.2 [4.3, 20.0]	+13.2 [7.1, 19.9]	0.95 [0.892, 1.021]	4.47 [3.587, 5.456]
KAT-Coder	208	208	+9.1 [2.9, 15.4]	+13.5 [7.2, 19.5]	0.90 [0.844, 0.957]	2.36 [1.912, 2.955]
Qwen3.5-9B	223	223	+9.0 [4.0, 14.3]	+14.9 [9.3, 20.6]	0.88 [0.822, 0.941]	2.87 [2.322, 3.516]
Laguna-XS	52	52	+26.9 [15.4, 40.4]	+38.5 [26.6, 51.1]	0.86 [0.770, 0.936]	6.22 [4.574, 8.063]
Nemotron-3.5	127	127	+7.9 [0.8, 15.0]	+12.4 [5.4, 18.8]	0.77 [0.695, 0.848]	2.66 [2.056, 3.406]
DeepSeek-V4-Flash†	284	284	-1.8 [-7.0, 3.5]	-7.2 [-11.0, -3.6]	1.34 [1.252, 1.440]	3.67 [3.233, 4.151]

Intervals are pointwise 95% paired source-task-cluster bootstrap intervals (4,000 draws), not adjusted for multiple comparisons. Estimates weight paired instances equally. Positive correctness differences favor ON. Resource ratios above 1 mean more use under ON. Correctness tint is centered on zero, with red/green for negative/positive changes on a shared ±40 pp scale. †DeepSeek is the separate API protocol with independent arrival calibration.

Table K.3: **Paired thinking contrasts under Sequential.** Correctness differences are ON minus OFF in percentage points. Resource contrasts are ratios of ON to OFF means. Brackets show 95% intervals. n_{score} and n_{res} give paired score and resource denominators.

Model	n_{score}	n_{res}	ΔResolved (pp)	ΔTPR (pp)	Steps ON/OFF	Output ON/OFF
GLM-4.7-Flash	138	138	+15.2 [8.7, 22.5]	+14.4 [7.6, 21.4]	1.03 [0.961, 1.106]	4.58 [3.214, 6.581]
Granite-4.2-3B	273	273	+16.8 [12.1, 21.6]	+28.8 [24.3, 33.5]	0.78 [0.741, 0.819]	5.85 [5.208, 6.504]
Granite-4.2-8B	166	166	+14.5 [7.8, 21.1]	+17.4 [10.0, 24.5]	0.98 [0.941, 1.031]	11.79 [10.257, 13.500]
Granite-4.2-30B	154	154	+8.4 [1.9, 15.6]	+9.8 [3.0, 16.9]	0.94 [0.879, 0.999]	4.34 [3.543, 5.320]
KAT-Coder	148	148	+9.5 [2.0, 16.9]	+20.0 [12.5, 27.9]	0.92 [0.848, 1.003]	2.54 [2.016, 3.163]
Qwen3.5-9B	280	280	+6.8 [1.8, 11.8]	+12.7 [8.1, 17.2]	0.94 [0.886, 1.003]	2.95 [2.503, 3.501]
Laguna-XS	206	206	+27.2 [20.9, 33.5]	+37.2 [31.2, 43.2]	0.87 [0.833, 0.910]	6.75 [5.803, 7.792]
Nemotron-3.5	164	164	+19.5 [12.8, 26.8]	+21.0 [14.3, 27.6]	0.74 [0.682, 0.805]	2.44 [2.010, 2.881]
DeepSeek-V4-Flash†	284	284	-2.5 [-7.0, 2.1]	-5.7 [-9.0, -2.5]	1.36 [1.269, 1.446]	3.90 [3.442, 4.404]

Intervals are pointwise 95% paired source-task-cluster bootstrap intervals (4,000 draws), not adjusted for multiple comparisons. Estimates weight paired instances equally. Positive correctness differences favor ON. Resource ratios above 1 mean more use under ON. Correctness tint is centered on zero, with red/green for negative/positive changes on a shared ±40 pp scale. †DeepSeek is the separate API protocol with independent arrival calibration.

Table K.4: **Paired thinking contrasts under In-Flight.** Correctness differences are ON minus OFF in percentage points. Resource contrasts are ratios of ON to OFF means. Brackets show 95% intervals. n_{score} and n_{res} give paired score and resource denominators.

Model	n_{score}	n_{res}	ΔResolved (pp)	ΔTPR (pp)	Steps ON/OFF	Output ON/OFF
GLM-4.7-Flash	136	136	+16.2 [8.8, 23.5]	+23.6 [15.9, 31.0]	0.93 [0.855, 1.022]	4.12 [3.007, 5.749]
Granite-4.2-3B	275	267	+17.5 [12.7, 22.2]	+33.6 [28.8, 38.6]	0.71 [0.672, 0.746]	6.04 [5.474, 6.650]
Granite-4.2-8B	175	175	+10.3 [4.6, 16.6]	+14.8 [8.5, 21.0]	0.98 [0.937, 1.037]	9.88 [8.384, 11.551]
Granite-4.2-30B	168	168	+10.1 [4.2, 16.1]	+15.2 [9.2, 21.4]	0.94 [0.880, 0.997]	4.85 [4.042, 5.818]
KAT-Coder	139	139	+15.1 [7.9, 23.0]	+22.6 [14.8, 30.2]	0.86 [0.795, 0.931]	2.12 [1.738, 2.593]
Qwen3.5-9B	280	280	+10.7 [5.7, 15.7]	+14.1 [9.0, 19.0]	0.80 [0.746, 0.849]	3.28 [2.803, 3.821]
Laguna-XS	205	205	+32.7 [25.9, 39.5]	+38.5 [32.2, 44.8]	0.89 [0.858, 0.920]	7.02 [6.040, 8.100]
Nemotron-3.5	151	151	+15.9 [9.3, 23.2]	+22.7 [16.1, 29.4]	0.74 [0.682, 0.804]	2.93 [2.568, 3.372]
DeepSeek-V4-Flash†	285	285	+3.9 [-1.4, 9.1]	-1.8 [-5.1, 1.4]	1.30 [1.218, 1.396]	3.42 [3.057, 3.835]

Intervals are pointwise 95% paired source-task-cluster bootstrap intervals (4,000 draws), not adjusted for multiple comparisons. Estimates weight paired instances equally. Positive correctness differences favor ON. Resource ratios above 1 mean more use under ON. Correctness cells are tinted rose for negative and mint for positive changes, with intensity proportional to the magnitude up to 40 pp. †DeepSeek is the separate API protocol with independent arrival calibration.

K.3 Instance-level resource changes and trajectory endings

The condition-level means show that thinking changes both correctness and consumption. We therefore look at two instance-level records, i.e., the distribution of those consumption changes and the way trajectories end. Figure K.3 reports separate action and token ratios across individual instances. With thinking OFF and ON, Granite-4.2-3B uses 238.5 and 169.1 mean steps on its 267 resource pairs, while output increases from 16.2k to 97.6k. Granite-4.2-8B uses 211.3 and 208.1 steps but 13.5k and 133.9k output tokens on 175 pairs. Thus, similar or lower mean step counts can accompany substantially more generation. These recorded quantities do not identify a latent reasoning mechanism or a hardware-normalized efficiency improvement.

Beyond consumption, Figure K.4 separates a declared finish from budget termination. For Granite-4.2-3B, the declared-finish share rises from 4.0% to 65.5%, but Resolved rises only from 3.6% to 21.1% on the 275 scored pairs. A finish declaration therefore remains distinct from passing all required and regression tests. Stopping behavior also differs within the Granite family: Granite-4.2-30B's time-limit share rises from 8.9% to 20.2%, whereas Granite-4.2-3B's declines from 4.0% to 2.2%. These are descriptive endpoint changes, not evidence that a particular stopping category caused the correctness change.

K.4 Outcome transitions and remaining request-type difficulty

Changes in resources and stopping behavior leave open which requests benefit. We therefore decompose each net correctness gain into newly solved instances and lost solutions, and then ask whether the request-type difficulty pattern changes. Granite-4.2-3B's 275 In-Flight pairs contain 50 gains and two losses. Figure K.5 reports this decomposition for all eight models.

DeepSeek illustrates why this distinction matters. Among 285 valid In-Flight pairs, 36 previously unresolved instances become Resolved and 25 previously resolved instances cease to be Resolved. Its instance-equal Resolved difference is $+3.9$ points with a 95% interval from -1.4 to $+9.1$, while the source-balanced estimate is $+7.2$ points with an interval from $+1.1$ to $+13.0$. These weight different target populations. The point estimates for TPR instead decrease, and independently calibrated delivery adds another limitation. We therefore report the transition pattern without claiming a robust, isolated benefit from its thinking toggle.

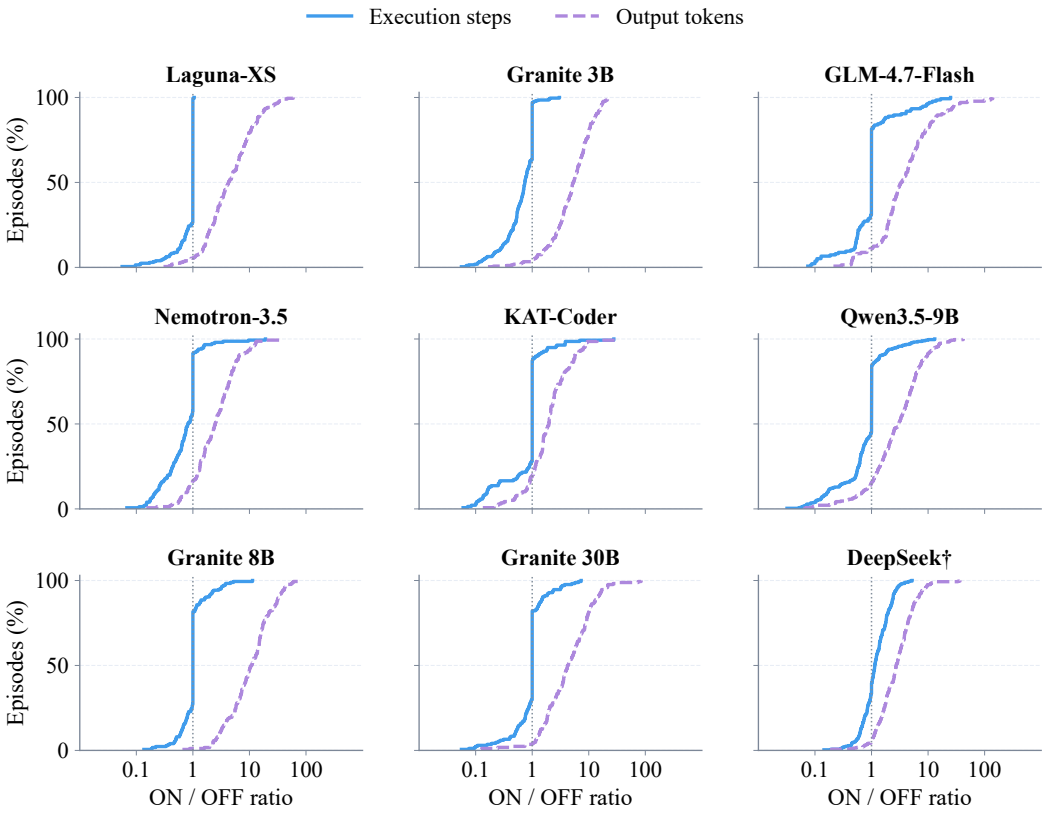


Figure K.3: **Action ratios and token ratios have different distributions.** Empirical cumulative distributions of within-instance ON/OFF ratios under In-Flight. The dotted line at one denotes unchanged use. Curves farther right indicate larger ratios. Axes are shared, with logarithmic ratio scales. All observed ratios are retained. Each metric requires both recorded values and a positive OFF denominator. These distributions differ from the ratios of means in Figure K.2. † marks DeepSeek's separate delivery protocol.

We next examine whether these instance-level gains change the request-type difficulty pattern. Additional tasks remain the lowest-Resolved group under ON for each of the eight frozen-arrival models. For Granite-4.2-3B, ON Resolved is 24.7% for requirement changes, 32.0% for cancellation/replacement, and 4.7% for added tasks. Table K.5 reports subgroup pair counts and OFF-to-ON changes. Because the groups contain different instances, these levels describe remaining difficulty rather than a paired causal comparison between request types.



Figure K.4: **Thinking changes how trajectories end.** Paired OFF and ON bars per model partition the same valid In-Flight instance pairs by the recorded termination reason, with values on the vertical axis. A finish is an agent declaration, not a correctness verdict. Other includes nonstandard or unrecorded endings. †DeepSeek uses its separate calibration protocol.

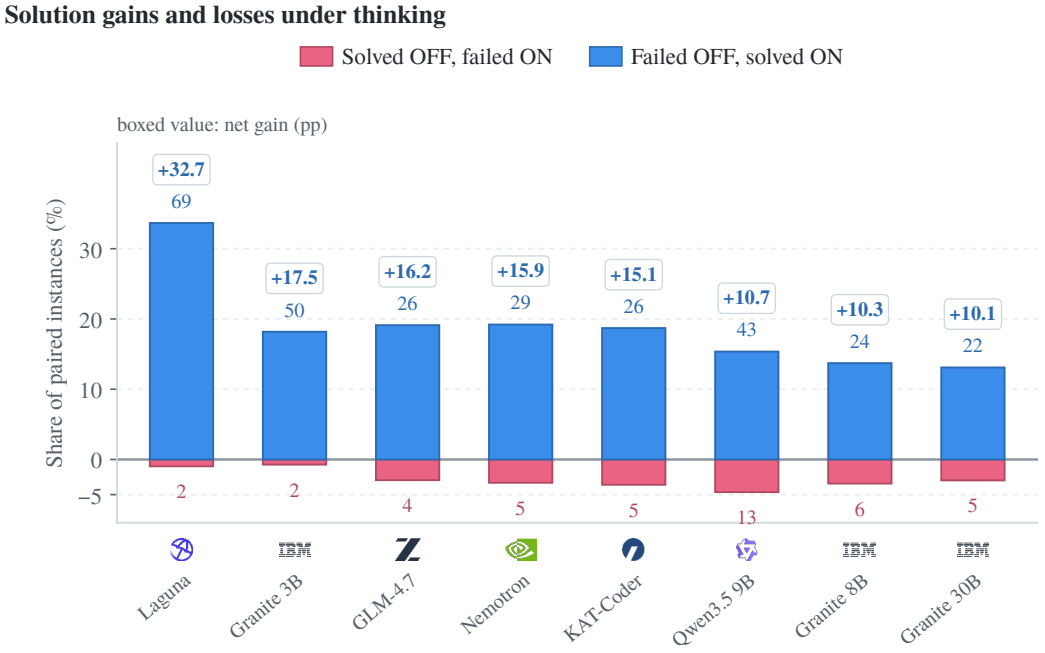


Figure K.5: **Net improvements combine newly solved instances and lost solutions.** Blue bars count instances solved only with thinking ON and pink bars those solved only OFF, below zero. Heights are percentages of each model's paired sample, while labels at bar ends are instance counts. The boxed value above each pair is the net gain in percentage points. Models follow the order in Figure K.1, with paired counts in Table K.1. Instances solved in both modes or neither mode are omitted from the bars.

Table K.5: **In-Flight thinking contrasts across request types.** Instance-equal ON-minus-OFF differences for each model. n counts jointly scored pairs within that request type. These descriptive subgroup estimates are not tests of differences between request types. The subsets vary in source and language composition.

Model	n	ΔResolved (pp)	ΔTPR (pp)
Requirement change			
GLM-4.7-Flash	42	+4.8	+16.4
Granite-4.2-3B	93	+19.4	+38.9
Granite-4.2-8B	65	+12.3	+11.0
Granite-4.2-30B	67	+11.9	+21.3
KAT-Coder	45	+15.6	+25.5
Qwen3.5-9B	95	+11.6	+14.2
Laguna-XS	65	+32.3	+39.8
Nemotron-3.5	55	+16.4	+22.3
DeepSeek-V4-Flash†	98	+3.1	-1.5
Cancellation / replacement			
GLM-4.7-Flash	54	+31.5	+35.8
Granite-4.2-3B	97	+27.8	+38.4
Granite-4.2-8B	59	+13.6	+17.5
Granite-4.2-30B	52	+11.5	+11.2
KAT-Coder	49	+12.2	+16.0
Qwen3.5-9B	97	+13.4	+19.7
Laguna-XS	75	+42.7	+41.8
Nemotron-3.5	51	+17.6	+19.1
DeepSeek-V4-Flash†	97	+4.1	-0.1
Additional task			
GLM-4.7-Flash	40	+7.5	+14.5
Granite-4.2-3B	85	+3.5	+22.4
Granite-4.2-8B	51	+3.9	+16.4
Granite-4.2-30B	49	+6.1	+11.2
KAT-Coder	45	+17.8	+26.8
Qwen3.5-9B	88	+6.8	+7.7
Laguna-XS	65	+21.5	+33.5
Nemotron-3.5	45	+13.3	+27.2
DeepSeek-V4-Flash†	90	+4.4	-4.1

†DeepSeek is the separate API protocol with independent arrival calibration. Delta tint uses the same signed scale as the paired contrast tables.

K.5 Separating General Gains from Adaptation-Specific Gains

The preceding analyses describe absolute gains across the three delivery conditions. We now test whether thinking provides an additional gain specifically under In-Flight. This requires comparing the thinking effects on common instances across two conditions, rather than comparing their separate confidence intervals. For a reference condition $R \in \{\text{Sequential}, \text{Upfront}\}$, define the per-instance interaction

$$\left(Y_{\text{ON,IF}} - Y_{\text{OFF,IF}}\right) - \left(Y_{\text{ON},R} - Y_{\text{OFF},R}\right).$$

This calculation retains only instances with all four observed outcomes and the same ON/OFF environment pairing across the two conditions. Figure K.6 shows the mean interaction for Resolved. All eight local intervals include zero against both references (the Nemotron–Upfront lower endpoint is exactly zero). These results limit an adaptation-specific interpretation of the absolute gains. They neither establish equivalence nor rule out effects within the interval. DeepSeek's Sequential interaction is positive in this exploratory analysis, but it combines thinking with independently calibrated delivery and cannot resolve that distinction.

Table K.6 restates the comparison against Sequential as a per-model gap, i.e., the In-Flight minus Sequential difference in Resolved with thinking OFF and with thinking ON, which Figure 7 plots in

the main text, together with the change between the two. Thinking raises Sequential and In-Flight Resolved by similar amounts (Tables K.3 and K.4), so across the eight local models the gap changes by -7.1 to $+5.8$ points while In-Flight Resolved rises by 10.1 to 32.7 points. As in Figure K.6, the interval for the change includes zero for each of the eight local models.

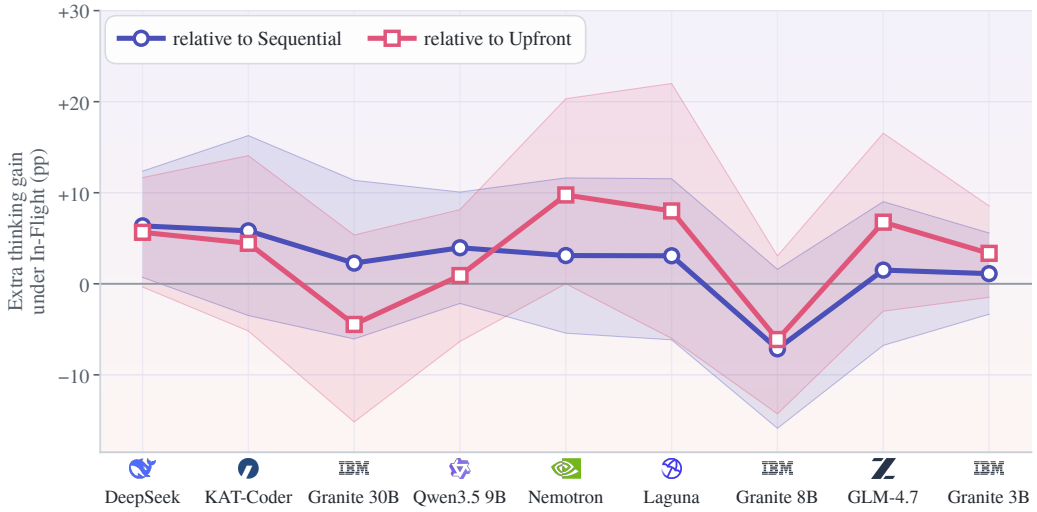


Figure K.6: **Absolute gains do not establish an extra gain specific to In-Flight.** Each point is the thinking gain (ON minus OFF Resolved) under In-Flight minus the same gain under Sequential (blue circles) or Upfront (pink squares), using common four-outcome cohorts. Positive values mean thinking helps more under In-Flight. Shaded bands are pointwise 95% paired source-task-cluster bootstrap intervals, and lines only guide the eye across configurations. DeepSeek is an independently calibrated protocol comparison.

Table K.6: **Thinking and the In-Flight minus Sequential gap.** Each gap is the paired In-Flight minus Sequential difference in Resolved, in percentage points, with thinking OFF or ON. The change is the ON gap minus the OFF gap. n counts instances with all four outcomes observed.

		In-Flight − Sequential gap (pp)		
Model	n	**Thinking OFF**	**Thinking ON**	**Change (ON−OFF)**
GLM-4.7-Flash	133	0.0	+1.5	**+1.5**
Granite-4.2-3B	269	−0.4	+0.7	**+1.1**
Granite-4.2-8B	126	+4.0	−3.2	**−7.1**
Granite-4.2-30B	132	−0.8	+1.5	**+2.3**
KAT-Coder	86	+1.2	+7.0	**+5.8**
Qwen3.5-9B	278	0.0	+4.0	**+4.0**
Laguna-XS	130	−2.3	+0.8	**+3.1**
Nemotron-3.5	129	−2.3	+0.8	**+3.1**
DeepSeek-V4-Flash†	283	−4.6	+1.8	**+6.4**

Positive gaps favor In-Flight. †DeepSeek uses mode-specific arrival calibration.

K.6 Coverage, Inclusion, and Missingness Sensitivities

Finally, we examine whether the reported direction depends on coverage, inclusion, or missing outcomes. Table K.7 reports paired coverage by arrival timing, and Table K.8 compares the evidence-based population with the strict scorer-valid subset. Each model retains its original paired cohort and resource settings, so these checks address the observations entering the comparison.

As a separate missingness check, assign each unavailable binary outcome the interval $[0, 1]$ and bound the mean ON-minus-OFF difference over every scheduled instance covered by the pairing rule. The resulting finite-cohort lower bounds for In-Flight Resolved remain positive for all eight

frozen-arrival models, ranging from 7.1 to 30.6 pp. Thus arbitrary values for missing outcomes cannot reverse their observed descriptive direction within those scheduled cohorts. These are sharp missing-data bounds, not confidence intervals, and do not replace the cluster uncertainty estimates or isolate the effect of an execution image.

Table K.7: **Paired coverage by arrival stratum and delivery condition.** Counts refer to paired runs of each model. U is the unified evidence-based paired cohort. S is its strict scored subset (both scorer-valid). Bars highlight S counts on a shared 0 to 300 scale. Local rows use the frozen planned In-Flight progress/budget stratum and carry that same task grouping into Upfront and Sequential. API rows use the recorded In-Flight bases in OFF/ON order: P denotes progress-based and B budget-based arrival.

Model	Stratum	Upfront		Sequential		In-Flight	
		U	S	U	S	U	S
GLM-4.7-Flash	Progress	29	28	29	29	28	27
GLM-4.7-Flash	Budget	110	107	109	107	108	106
Granite-4.2-3B	Progress	5	4	5	5	5	4
Granite-4.2-3B	Budget	266	252	268	253	270	259
Granite-4.2-8B	Progress	19	17	24	20	23	20
Granite-4.2-8B	Budget	80	80	142	141	152	150
Granite-4.2-30B	Progress	26	26	33	32	35	34
Granite-4.2-30B	Budget	89	89	121	121	133	132
KAT-Coder	Progress	31	29	20	19	21	19
KAT-Coder	Budget	177	172	128	124	118	115
Qwen3.5-9B	Progress	63	60	71	69	70	69
Qwen3.5-9B	Budget	160	159	209	206	210	208
Laguna-XS	Progress	0	0	6	6	5	5
Laguna-XS	Budget	52	50	200	191	200	192
Nemotron-3.5	Progress	22	20	29	29	25	24
Nemotron-3.5	Budget	105	103	135	132	126	125
DeepSeek-V4-Flash†	P/P	239	239	239	239	240	240
DeepSeek-V4-Flash†	B/B	11	11	11	11	11	11
DeepSeek-V4-Flash†	B/P	6	6	6	6	6	6
DeepSeek-V4-Flash†	P/B	28	28	28	28	28	28

†DeepSeek uses the same API executor, with independently calibrated arrival schedules. It is not a frozen-arrival local twin. Strict coverage is a sensitivity subset, not a second schedule. Counts describe coverage, not task success.

Table K.8: **In-Flight conclusions under the strict scored subset.** ON-minus-OFF instance-equal differences for each model. Unified retains evidence-classified observed outcomes. Strict additionally requires both OFF and ON to be scorer-valid. Counts are paired instances. These descriptive subset estimates use the same settings, not additional experiments.

Model	n_{U}	n_{S}	ΔResolved (pp)		ΔTPR (pp)	
			Unified	Strict	Unified	Strict
GLM-4.7-Flash	136	133	+16.2	+16.5	+23.6	+23.6
Granite-4.2-3B	275	263	+17.5	+18.3	+33.6	+34.2
Granite-4.2-8B	175	170	+10.3	+10.6	+14.8	+15.6
Granite-4.2-30B	168	166	+10.1	+10.2	+15.2	+14.5
KAT-Coder	139	134	+15.1	+14.9	+22.6	+23.2
Qwen3.5-9B	280	277	+10.7	+10.8	+14.1	+13.7
Laguna-XS	205	197	+32.7	+33.5	+38.5	+38.4
Nemotron-3.5	151	149	+15.9	+16.1	+22.7	+23.0
DeepSeek-V4-Flash†	285	285	+3.9	+3.9	-1.8	-1.8

†DeepSeek is the separate API protocol with independent arrival calibration. Delta tint uses the same signed scale as the paired contrast tables.

L Data and Licensing

This appendix covers licensing and provenance, the handling of public request text, and what the design does and does not control regarding contamination. The repository-derived tasks use public source material with resolved permissive licenses, including MIT, Apache-2.0, BSD-3-Clause, and ISC. Each task retains its repository and commit identifiers, applicable license, and validation provenance. The release includes task metadata, request text, test identifiers, reference changes, and environment recipes. Upstream source trees are obtained from their original repositories. Controlled projects are authored for the benchmark.

Requests are assembled from public issue or commit text, authored specifications, and fixed interaction templates. The benchmark uses no private repositories, private communications, or human-subject interaction study. Public version-control material can include author names or handles. The task records use repository and commit identifiers without profiling individuals. Exact construction rules and licenses are provided with the dataset.

For contamination, within-instance comparisons hold repository exposure constant across arrival conditions, while source-specific results distinguish authored projects from public-repository tasks. This controls a shared property of the paired tasks. It does not establish that every evaluated model was unexposed to the public source material. Reference patches are used for validation and remain outside the agent's context during evaluation.