SCALING PROACTIVE COMPUTE: WHEN AGENTS ANTICIPATE THE TASK

Anonymous authors
Paper under double-blind review

ABSTRACT

Language model agents benefit from additional test-time compute, yet often remain idle before queries arrive and while tools run. We study how scaling *proactive compute* during these periods improves performance, with three main findings. First, controlled sweeps of preparation and downstream reasoning budgets across nine offline benchmarks show that preparation improves the accuracy–test-time-compute frontier. Reusing studies across questions can also reduce total generated-token costs. In three machine-learning engineering tasks, agents that reason while tools run reach better scores earlier under equal wall-clock budgets. Second, our wiki-style and multi-agent study strategies improve efficiency on Harvey LAB's 134M-token document collection and extend preparation beyond a single context window. Third, our analyses show how preparation reduces downstream reasoning and tool use. On SWE-QA-Pro, a 16k-token study more than halves subsequent reasoning and cuts tool calls roughly tenfold at the same accuracy, even when the solver chooses its own budget. Together, these findings establish proactive compute as an effective complement to test-time scaling and show how agents can turn otherwise idle time into useful preparation.

1 INTRODUCTION

Scaling test-time compute has led to substantial gains in the task performance of LLM-powered agents. Models can spend this compute exploring candidate solutions in parallel (Brown et al., 2024) or extending their reasoning sequentially (Muennighoff et al., 2025). Choosing how to allocate this compute for each problem can further improve accuracy at a fixed budget (Snell et al., 2024). Yet an agent following a reactive loop, as in ReAct (Yao et al., 2023b), waits for a query or new environmental feedback before continuing its reasoning. During these waits, the agent can *proactively* prepare for upcoming work, for example by inspecting an available repository or anticipating the possible outcomes of a running tool call. We call computation spent preparing during these otherwise idle periods *proactive compute*. In this work, we ask:

How can we scale proactive compute effectively to improve agents' test-time performance?

To answer this question, we study two opportunities for preparation (Figure 1). In the *offline* setting, an agent studies the available context before a query arrives and carries the resulting study into task execution, building on Sleep-time Compute (Lin et al., 2025). In the *online* setting, it anticipates tool results and plans its next steps while waiting for execution to finish, as in IdleSpec (Choi et al., 2026). Our offline experiments use budget forcing to vary preparation and downstream reasoning independently while keeping the model and environment fixed. In the online experiments, we vary the proactive compute available during tool execution and compare agents under equal wall-clock budgets. We organize this controlled investigation around three questions.

How much does scaling proactive compute improve test-time efficiency? Across nine benchmarks spanning mathematics, abstract reasoning, software engineering, and legal work, we show that offline preparation improves the accuracy–test-time-compute frontier, especially when the solver has little compute after the question arrives. At the largest study budget, the gain in test-time efficiency ranges from 9 to 48 percentage points across benchmarks. When questions share a context, reusing a study can also offset its preparation cost and reduce total generated-token use.

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.

In three machine-learning engineering tasks, agents that prepare while tools run reach better scores earlier under the same wall-clock budget.

How can proactive compute scale beyond a single context window? We scale proactive preparation on Harvey LAB's 134M-token legal document collection (Grupen et al., 2026), which far exceeds an agent's context window. We develop a wiki-style study strategy that uses the collection's hierarchy to organize exploration and findings. Even the smallest wiki-style study achieves higher test-time efficiency than plain study at every tested budget. To scale preparation further, we partition the collection among multiple agents and consolidate their findings into notes for the solver. These notes allow total proactive compute to grow beyond a single context window. Across all tested scales, parallel study explores more distinct documents and matches or exceeds the efficiency of serial study.

How and why does proactive compute help? We trace what agents discover during study and use content interventions to identify which discoveries support downstream performance. These analyses show that preparation supplies near-complete solutions in mathematics, reusable procedures in rule induction, and file contents that would otherwise require tool calls on repository tasks. Even when the solver chooses its own reasoning budget, a 16k-token study on SWE-QA-Pro more than halves subsequent thinking and cuts tool calls roughly tenfold at the same accuracy. To determine how much of a study the solver needs to retain, we also evaluate compressed study trajectories. On mathematics tasks, summaries of a few hundred tokens preserve most of the benefit of a 128k-token study, whereas repository studies draw their value from information distributed across many files.

Together, these results establish proactive compute as an effective complement to test-time scaling and show how to turn idle time into preparation that improves task performance. Our contributions are:

1. Quantifying the efficiency gains of scaling proactive compute. Through controlled evaluations across nine offline benchmarks and three online tasks, we establish how proactive compute improves the test-time accuracy–compute frontier and enables agents to reach better results sooner. We quantify these gains across preparation budgets and show how reusing a study reduces amortized generated-token costs.

2. Scaling proactive compute on massive contexts. We develop wiki-style and multi-agent study strategies for Harvey LAB's 134M-token document collection. By organizing study around the collection's structure and distributing exploration across agents, we improve test-time efficiency and scale preparation beyond a single context window.

3. Explaining how and why proactive compute helps. Through trace analyses, content interventions, and compression experiments, we identify how preparation replaces downstream reasoning and tool calls. We demonstrate these savings when solvers choose their own reasoning budgets and establish task-dependent differences in how much a study can be compressed while retaining its efficiency gains.

2 Related Work

Scaling inference-time compute. Language models can improve their answers by sampling additional candidates (Brown et al., 2024; Wang et al., 2023), searching over reasoning paths (Snell et al., 2024; Yao et al., 2023a), or extending a reasoning trace through budget forcing, as in s1 (Muennighoff et al., 2025). Studies of inference scaling further establish that the gains depend on computational allocation and problem difficulty (Snell et al., 2024; Wu et al., 2024). These approaches scale reactive compute, spent after the query arrives. We quantify how scaling proactive compute improves the accuracy–reactive-compute frontier and develop methods for studying contexts far beyond a single context window.

Idle-time compute. Agents waiting for user requests or tool execution can use these idle periods to prepare for subsequent work. Sleep-time Compute (Lin et al., 2025) studies contexts before queries arrive, while IdleSpec (Choi et al., 2026) generates speculative plans during tool delays. Related approaches anticipate information needs between user interactions (Hu et al., 2026) or overlap generation with asynchronous function execution (Gim et al., 2024). Building on this line of work, we

quantify how preparation accelerates progress under fixed wall-clock budgets and use qualitative analyses and controlled interventions to explain how it makes the test-time reasoning more efficient.

3 PROACTIVE AGENTS

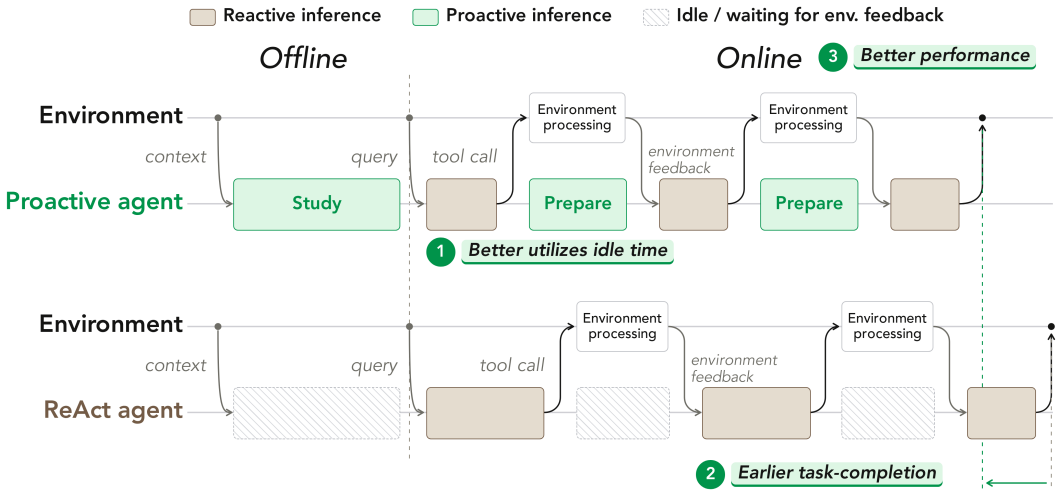

Figure 1: **Environment–agent interaction timelines for proactive and reactive agents.** Before a query arrives (*offline setting*), the proactive agent studies the available context. After the query arrives (*online setting*), it uses the time spent waiting for tool results to anticipate possible outcomes and prepare its next steps. A reactive agent remains idle during both periods. This additional computation can improve task performance and shorten the time from query arrival to task completion.

Environment and reactive agents. We consider an agent deployed in an environment to carry out a task or answer a query, denoted by q. Before q arrives, context C, such as code files and company logs, may already be available. Once it receives q, the agent follows the ReAct framework (Yao et al., 2023b), alternating between reasoning and taking actions $a_t \in A$ that yield observations $o_t \in O$. After receiving o_t, its interaction history is $H = (C, q, a_1, o_1, \ldots, a_t, o_t)$. In this reactive setup, the agent remains idle both before the task arrives and while waiting for observations from its actions.

Proactive agents. Proactive agents can use these idle periods in two ways (Figure 1). In the offline setting, the agent explores the context C before the task q arrives. We call this phase *study* and denote the resulting trajectory by S_P. When q arrives, the study trajectory remains in the history, giving $H = (C, S_P, q)$. In the online setting, the agent uses the time between sending an action a_t and receiving its observation o_t to anticipate the observation and reason further about the task. This reasoning is appended to H. By using otherwise idle time to study the context and reason ahead, proactive agents can complete tasks sooner and achieve better results.

4 EXPERIMENTAL SETUP

Controlling the proactive and test-time compute through budget forcing. In order to quantitatively study the scaling behavior of proactive compute, we control the proactive and test-time compute using budget forcing on the generated tokens. We denote the proactive budget, the tokens the agent generates while it would otherwise be idle, as P and the test-time budget, the tokens it generates after the task arrives, as T respectively. These budgets constrain the generated tokens of the model and not the additional input tokens incurred by the results of tool calls, which are available in some benchmarks (for more discussion on this, see Appendix B.3).

In the *offline* setting, the study phase receives the environment context C, a general exploration prompt and, on the tool-use benchmarks, the same set of actions A as the test-time solver. The study and the solver use the same model and environment, and differ only in whether they see q. We

sweep a proactive budget $P \in \{0, 64, 256, 1\text{k}, 4\text{k}, 16\text{k}, 32\text{k}, 64\text{k}, 128\text{k}\}$ and test-time budget $T \in \{64, 256, 1\text{k}, 4\text{k}, 16\text{k}, 32\text{k}, 64\text{k}\}$. For each budget, we extend a trajectory to the longest budget first, and take prefixes of the longest chain to get chains for the lower budgets. We extend trajectories to the desired budget using budget forcing as in Muennighoff et al. (2025), by appending continuation prompts such as "Keep reasoning. Do not answer yet." (budget forcing is further described in Appendix B.1). In the *online* setting, $P \in \{128, 512, 2\text{k}, 8\text{k}\}$ caps the tokens generated per wait. Note that in the offline setting the budget P is always reached, while in the online setting it may be cut short by an observation o_t returning. Prompts for the study and the test-time solver are in Appendices F.1 and F.2.

Offline benchmarks. We study the gains of scaling offline proactive compute on 9 benchmarks (Table 1, Appendix B). For AIME 2024/2025/2026 and HMMT Feb 2026, we follow the setting of Lin et al. (2025), where C is a prefix of the problem and q is the last sentence in the problem. On ARC-AGI-1 (Chollet, 2019) and MIR-Bench (Yan et al., 2025), C is a set of input-output examples, and q is a new input. On StudyBench DSPy (Li et al., 2026) and SWE-QA-Pro (Cai et al., 2026), C is a code repository and q is a user question. On Harvey LAB (Grupen et al., 2026), C is a fictitious law firm's filesystem of 134M tokens, and q is a legal task. StudyBench, SWE-QA-Pro and Harvey LAB use rubric grading by default, which we perform using GLM-5.3-Flash on medium reasoning effort (judge prompt in Appendix F.6).

Online tasks. We study the benefits of spending more online proactive compute on three MLE-bench tasks (Chan et al., 2025): Denoising Dirty Documents (Denoise), Statoil/C-CORE Iceberg Classifier Challenge (Iceberg) and LMSYS Chatbot Arena (LMSYS). A DeepSeek-V4-Flash agent works in a shell for 30 minutes of wall-clock time on 4 CPU cores, running shell commands and submitting training scripts, which are run and graded. While a command runs, the proactive agent is prompted to anticipate its possible results, prepare for them and review past findings, for at most $P \in \{128, 512, 2\text{k}, 8\text{k}\}$ tokens per wait. The reactive agent receives equivalent guidance with each command's result but cannot generate while it waits. Each of 28 seeds per task runs once with the reactive agent and once with a proactive agent at each budget, and we plot the best submission so far (setup and statistics in Appendix C.1).

Models. Our primary model is DeepSeek-V4-Flash-0731 served through OpenRouter, with 3 seeds per question. We also run Qwen3.8-27B and gpt-5.6-luna on identical tasks on the offline grid with 1 seed each, which can be seen in Appendix B.4.

Metric for measuring test-time efficiency. We capture the performance of a test-time agent by the accuracies it reaches as it spends more test-time compute, which we plot as a curve on an accuracy vs test-time budget graph. We can summarize this curve in one number, the test-time efficiency E defined as:

$$E(P) := \frac{\sum_k w_k \, \frac{1}{2}\big(a_k + a_{k+1}\big)}{\sum_k w_k}, \qquad a_k = \mathrm{acc}(P, T_k), \quad w_k = \log_2 \frac{T_{k+1}}{T_k},$$

where $T_1 < T_2 < \dots$ are the tested budgets. Equivalently, $E(P)$ is the area under the curve (AUC) of the accuracy vs test-time compute graph, where test-time compute is on a log scale and the area is normalized by the width of the curve. For more in-depth discussion on the choice of the metric, see Appendix B.2.

5 Scaling offline proactive compute shifts the test-time accuracy–cost frontier

In this section, we sweep the proactive budget P and the test budget T on nine benchmarks from six task families on DeepSeek-V4-Flash-0731. We note that math and reasoning benchmarks are toy settings with one query per context, and on math the context holds all of the problem except its last sentence, so the study often solves it outright. They show the most that a study can give when the context nearly determines the query, while the repository and legal benchmarks are more realistic settings. We find that proactive compute can substitute for test-time compute when T is small (§5.1), the test-time efficiency rises with P and then saturates (§5.2), and once several questions share a study, it moves the accuracy–token frontier above no study (§5.3). We also note that these results generalize to other models and include fine-grained, per-benchmark results in Appendices B.4 and B.5.

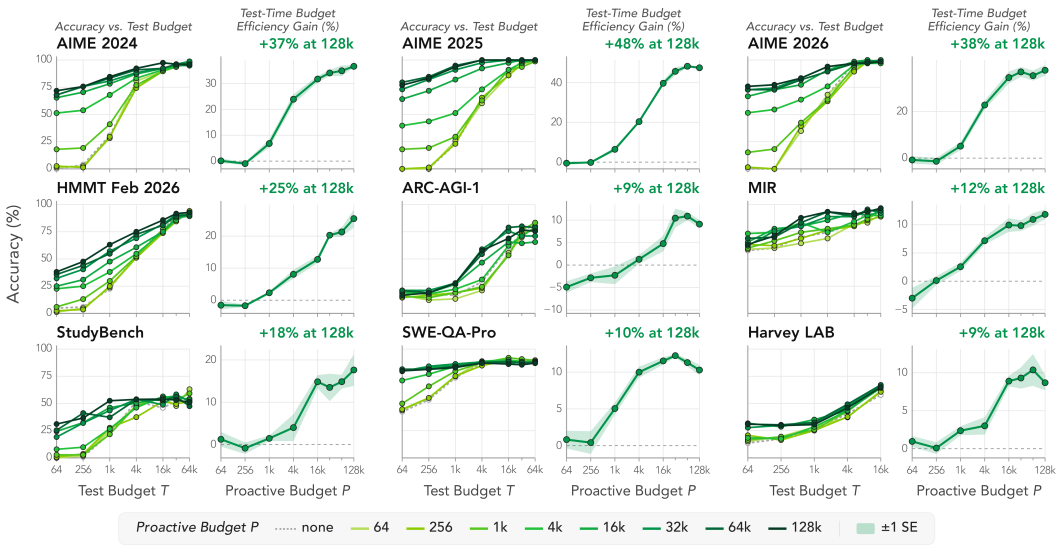

Figure 2: **Scaling the proactive budget from no study to a 128k-token study increases test-time performance and test-time budget efficiency.** We scale offline proactive compute over 9 benchmarks. P counts the tokens the study generates before the question arrives, and T counts the tokens generated after it. **(Left)** Accuracy is plotted over test-time compute T, and curves are separated by proactive budget P. **(Right)** Test-time efficiency gain $E(P) - E(0)$ against proactive budget P, with the gain at $P = 128$k printed. DeepSeek-V4-Flash, 22–30 questions per benchmark (Table 1), 3 seeds.

5.1 Proactive compute improves accuracy for low test-time budgets

On math benchmarks, at $T = 64$ the solver without a study does extremely poorly, but after a 128k-token study it reaches 0.72–0.77 on AIME (Figure 2, left), an accuracy that the no-study solver needs 4k–16k tokens to reach (per-benchmark grids in Appendix B.5). As T grows, the curves converge, showing that proactive compute can raise performance at low T budgets but does not increase the ceiling at high T. Interestingly, ARC-AGI-1 and MIR exhibit the largest gains at intermediate values of T, where the solver has enough budget to apply the algorithmic procedure that the study discovered to the new example.

5.2 Test-time efficiency rises with P and then saturates

The gain in test-time efficiency E (§4) is flat for small studies, rises from $P \approx$ 1k to 16k depending on the benchmark, and slows its growth or decreases above 32k (Figure 2, right). At $P = 128$k, our highest studying budget, its point estimate is positive on every benchmark, up to $+0.48$ on AIME 2025. The growth slows because the study finds new material more slowly as it grows, and on math most late study tokens revisit earlier findings (Appendix E.2), and the largest budgets may also suffer from context rot (Hong et al., 2025).

5.3 Shared across questions, proactive compute moves the accuracy–token frontier

While proactive studying improves performance at low T budgets, benefiting latency-critical tasks, it can also decrease costs on net. Because a study is a fixed cost for each context, if n tasks share a context, the amortized cost per task becomes $P/n + T^*$, where T^* is the test budget required to solve the problem or obtain, on average, a certain accuracy level. Figure 3 fixes one target accuracy per benchmark and plots the generated tokens saved over no study, $n(T_0^* - T_P^*) - P$, for studies of $P = 4$k, 32k and 64k tokens. A study pays for itself after 2–14 questions at $P = 4$k and after 17–44 questions at 64k. As n grows, larger studies save the most, because they cut T^* the most: after 1,024 questions the 64k study saves 1.5–3.8M generated tokens.

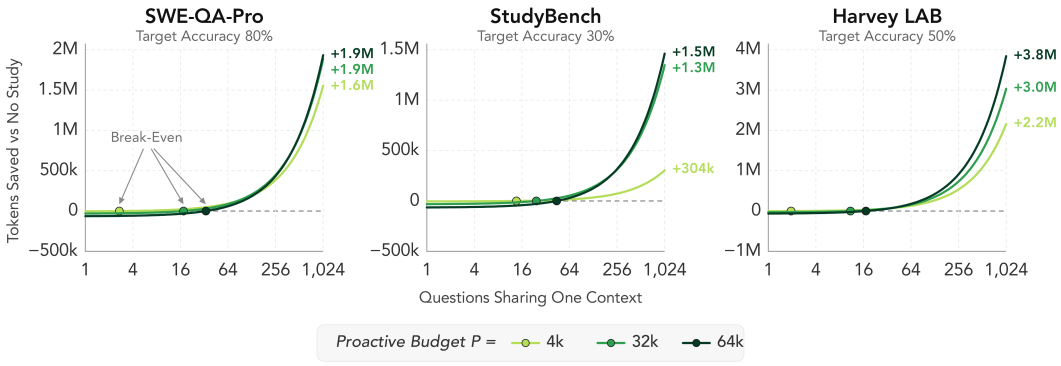

Figure 3: **Studying a shared context pays for itself after a few questions, and larger studies save more at scale.** We plot the total output tokens that a proactive agent saves over a reactive agent as more questions share one context. We calculate these numbers using the proactive and test-time budgets required to reach a fixed target accuracy per benchmark. Dots mark when studying breaks even, and numbers on the right give the tokens saved after 1,024 questions. DeepSeek-V4-Flash, 30 questions per benchmark, 3 seeds.

While proactive compute generates more tokens when answering one question, amortizing over questions that share the same context can decrease total output token cost.

We also test whether these findings generalize to other model families, Qwen3.8-27B and gpt-5.6-luna (1 seed each, and $T \leq 16$k for gpt-5.6-luna). Both show efficiency gains and accuracy-vs-T curves similar to those of DeepSeek-V4-Flash (Figure 10, Appendix B.4), and the per-benchmark grids agree across the three models (Figures 14–19, Appendix B.5).

6 Online proactive compute turns idle time into faster progress

In this section, we spend proactive compute while waiting for feedback from the environment. During this idle time, while a command is running, proactive agents are prompted to anticipate the result and plan ahead, spending 128 to 8k tokens of proactive compute. In comparison, reactive agents get the same anticipate-and-plan guidance but wait for the command to return before continuing.

6.1 Proactive agents get ahead early

With the same wall-clock time, proactive agents reach better scores sooner than the reactive agent (Figure 4). At 10 minutes, the 8k proactive agent leads on Denoise and Statoil Iceberg ($p = 0.022$ and 0.008, both of which are significant after Holm correction). On LMSYS, the 8k proactive agent also has a lower mean loss than the reactive agent from 5 to 25 minutes, but this gap is smaller and not significant ($p = 0.27$ at 20 minutes). At the end of the session the performances start to converge, similar to the offline proactive compute setting, where performances converge at high test-time budgets T.

6.2 Scaling the proactive budget converts more idle time into work

By default, the reactive agent generates for 17–22% of the session, and spends the rest of the time idling (Figure 5). Proactive agents utilize more of the idle time, and as the proactive budget increases, the utilization does as well. At 128 and 512 token budgets, proactive work takes up 3–9% of the session time, 14–18% at 2k and 37–48% at 8k (session timelines and mean share of wall-clock time can be seen in Figure 23, Appendix C). At 8k the agent spends 66–73% of the wall-clock time generating tokens, and 25–29% on reactive tokens, about as much as at the smaller budgets. Therefore, the extra proactive work goes into time that would otherwise be idle and the reactive share of wall-clock time stays about the same across budgets.

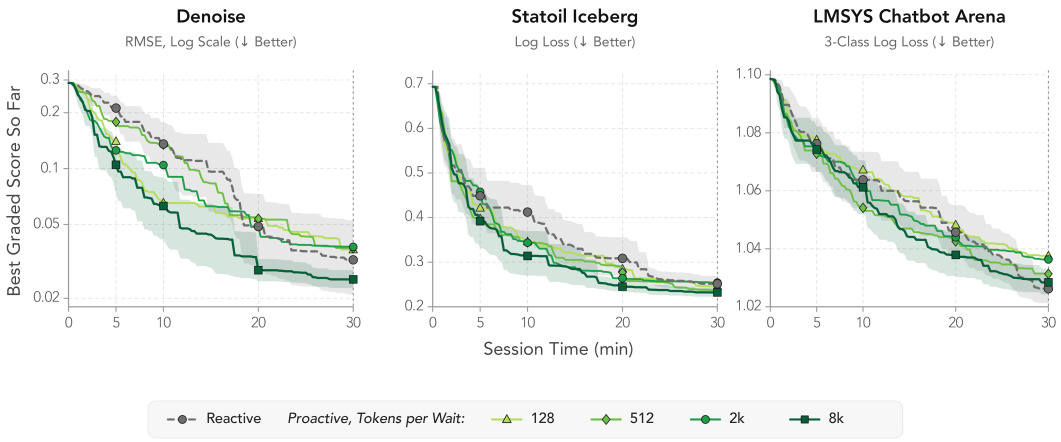

Figure 4: **With the same wall-clock time, proactive agents reach better scores sooner.** We plot the best graded score so far against session time (lower is better). The curves are the arithmetic means for the reactive agent (grey, dashed) and for proactive agents with 128 to 8k tokens per wait (shades of green). Shaded bands are 95% confidence intervals obtained by bootstrapping over seeds. DeepSeek-V4-Flash, 28 seeds per task and agent.

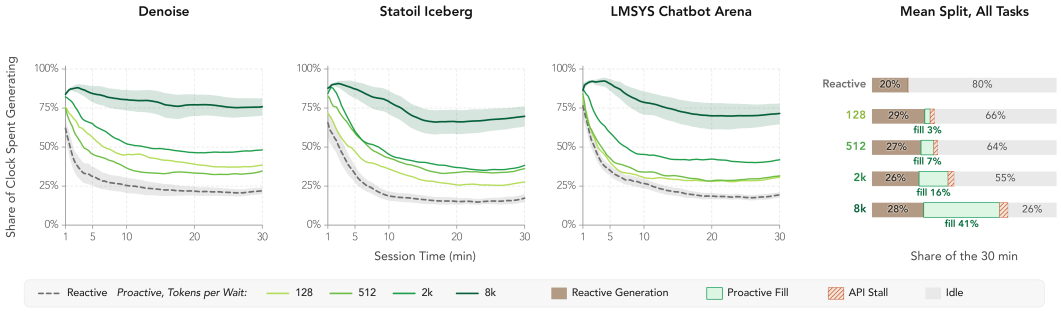

Figure 5: **Larger proactive budgets utilize more of the idle time. (Left)** Here, we plot the cumulative share of session time in which the model is generating, both reactively and proactively. API stalls (calls over 30 s at under 25 tokens/s) are not considered. Shaded bands are 95% confidence intervals obtained by bootstrapping over 28 seeds. **(Right)** Mean split of wall-clock time between reactive compute, proactive compute, idle time, and API stalls. DeepSeek-V4-Flash, 28 seeds per task and agent.

7 Scaling proactive compute on a 134M-token context: A case study on Harvey LAB

The above method of spending proactive compute offline simply copies the proactive agent's trajectory to the test-time solver. On the other hand, for very large contexts C, we may want to scale proactive compute further than the context window to make studying feasible at all. As a case study, we study Harvey LAB, a 134M-token filesystem of legal matters from a fictitious law firm, and ask if adding structure to the studying can scale proactive compute more effectively (§7), and how we can spend proactive budgets beyond the context window (§7). Scores are rubric-graded by GLM-5.3-Flash (§4), and test-time efficiency E is calculated over $T \leq 16$k. The document store, the questions and the study setup are described in Appendix D.1, and the wiki, parallel and serial scaling prompts are in Appendix F.4.

Leveraging a context's structure scales proactive compute more effectively The Harvey LAB filesystem is organized hierarchically, where each folder corresponds to a "matter", which is an interaction or deal with a third-party firm. Inside each folder are subfolders like "Administration" and "Correspondence". This hierarchical structure motivates creating a wiki-like object during study time, which we do by prompting the study agent to explore as if it were creating a wiki, and even-

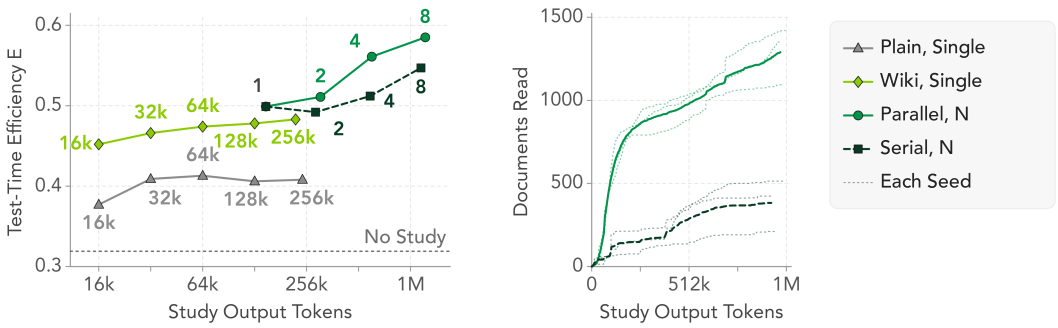

Figure 6: **Dividing work between multiple study agents scales beyond the context window. (Left)** Test-time efficiency against the output tokens generated by the study, evaluated over 120 questions and $T \leq 16$k. For N agents, output tokens include the write-up of each agent's notes. Single-agent studies end due to context limitations, while scaling the number of agents N goes past it. **(Right)** Distinct documents read in full against study output tokens: mean over 3 seeds (bold) and results for each seed (dotted). DeepSeek-V4-Flash, 120 questions, 3 seeds for single-agent studies, and for multi-agent studies 1 seed on the left and 3 on the right.

tually write a wiki in plain text. At every budget P, the wiki-prompted study is more efficient at test time ($E \approx 0.47$ against 0.40 for the plain study, Figure 6, left). Notably, even the smallest wiki study beats the plain study at every tested budget.

Multi-agent studying scales beyond the context window. We tried to scale to 384k tokens on multiple seeds, with and without the wiki-prompt, but the continuation prompts did not work reliably. Therefore, we devise two methods for scaling proactive compute using multiple agents, and thus multiple context windows. First, we scale the proactive compute in parallel, giving each parallel agent a partition of the filesystem to study, summarizing each agent's trace into a note, and concatenating these notes together to use as a prompt S_P. Second, we scale the proactive compute serially, letting a serial agent explore, summarizing the trace into a note, and giving serial agents all of the previous notes written. We give each agent $P = 131$k and scale the number of agents N instead. We find that scaling the proactive compute in parallel beats or matches serial scaling at every N, and find that parallel agents explore many more documents than serial agents do (Figure 6, right).

8 How and why does proactive compute help?

In this section, we present key experiments that describe how proactive compute helps test-time solvers. We find that studying replaces the solver's tool calls (§8.1), the distribution of signal over the study depends on the benchmark (§8.2), and a solver that chooses its own test-time compute T naturally spends less with a study (§8.3). The full list of analysis experiments is in Appendix E.

8.1 Proactive studying replaces test-time tool use

With a study in its history, the solver makes fewer tool calls. At the largest test budget, a 128k study cuts tool calls on all three tool-use benchmarks by up to tenfold (Figure 7, left). At the smallest test budgets, where proactive compute helps the most, the solver without a study has almost no budget for tool calls, so the study supplies information the solver finds hard to get. At small T the solver also trusts the study enough to copy a wrong answer planted in it (Appendix E.3).

8.2 Math studies compress to a few hundred tokens, SWE studies do not

To see how concentrated the study's signal is, we ask the same model to compress the 128k study, without seeing the question, to M tokens (Figure 8, prompts in Appendix F.5). On math, a summary of a few hundred tokens keeps most of the gain, much more than a native study four times its size. On repositories the signal is spread over many files, so a summary keeps less of the gain, and per delivered token it only matches or slightly outperforms a native study.

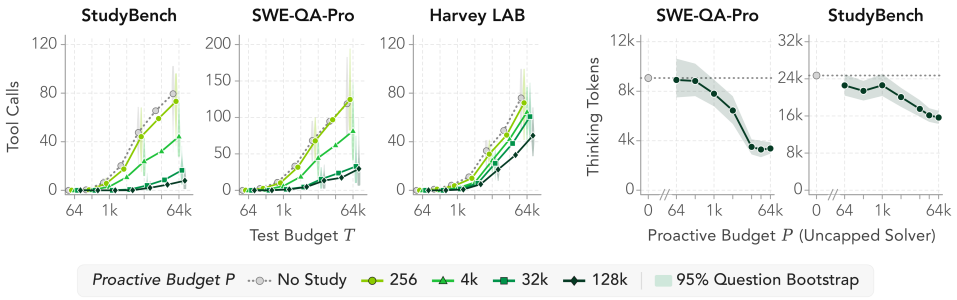

Figure 7: **Proactive studying replaces test-time tool calls and thinking. (Left)** Tool calls per question against the test budget T, with curves separated and colored by proactive budget P. Bands are 95% CI. **(Right)** The natural thinking tokens per question the solver spends (no budget forcing) against P. Bands are 95% CI. Dotted lines are the no-study baselines for thinking tokens. DeepSeek-V4-Flash, 30 questions per benchmark, 3 seeds.

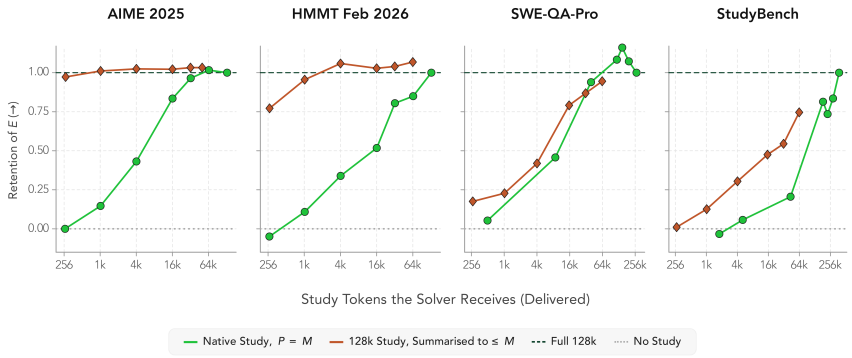

Figure 8: **Compressing a 128k study preserves performance on math benchmarks, but degrades performance on SWE tasks.** We plot the share of the 128k study's test-time efficiency gain that was retained by a compressed study trace over the size of the trace S_P that the solver receives. E is over $T \leq 16$k, and Appendix E.4 gives the method. DeepSeek-V4-Flash, 26 questions on AIME 2025, 28 on HMMT Feb 2026 and 30 on SWE-QA-Pro and StudyBench, 3 seeds.

8.3 Natural test-time compute

When the solver chooses its own test budget, with no cap on thinking or tool calls (Figure 7, right), a study makes it spend much less at the same accuracy: on SWE-QA-Pro a 16k study more than halves the solver's thinking and cuts its tool calls about tenfold. On StudyBench the study also cuts tool calls about tenfold and thinking by more than a quarter.

9 Conclusion

Our results show that agents can *proactively* begin useful work before the query or observation they are waiting for arrives. Through controlled experiments, we quantify how proactive preparation improves the accuracy–test-time-compute frontier and helps agents reach better scores sooner. Scaling these benefits to large environments depends on how agents organize their study. When questions share a context, reusing this preparation can also reduce total generated-token costs. On Harvey LAB's 134M-token collection, wiki-style organization makes preparation more effective, while parallel exploration and consolidated notes extend it beyond a single context window. Our analyses connect these gains to reasoning and information gathering completed in advance, and show which parts of a study the solver needs to retain. Together, these findings provide a quantitative foundation and practical strategies for scaling proactive agents. They motivate further work on agents that learn when and what to study, how to organize their findings, and how to share and reuse that knowledge across future tasks.

AI USE STATEMENT

In this work, we used generative AI tools for implementing experiments, performing quantitative data analysis and qualitatively reviewing data. We have not used generative AI tools for generating synthetic data, developing theoretical models, and forming experimental hypotheses. The rest of the required disclosure tasks are not applicable to this work. Additionally, we have used generative AI tools for creating scientific figures and images, creating and editing software code, creating artifacts, aiding in editing the research paper to improve readability, and identifying relevant literature. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work involves no human subjects and no personal data. All benchmarks are public and Harvey LAB is a fictitious law firm, so no real client documents are used.

REPRODUCIBILITY STATEMENT

Appendix B lists every benchmark, the context-query split, and the grader. Appendix B.1 describes budget forcing, the budget grids and the tool caps. Appendix C.1 and Appendix D.1 give the online task and Harvey LAB setups. Appendix F reproduces every prompt verbatim. Models are described with their versions (DeepSeek-V4-Flash-0731, Qwen3.8-27B and gpt-5.6-luna, with GLM-5.3-Flash as the judge), and seeds are stated per experiment. The supplementary material contains the harness code, one configuration per run, and sample rollouts (study traces, solver cells and online sessions) for each section.

REFERENCES

Bradley Brown et al. Large Language Monkeys: Scaling Inference Compute with Repeated Sampling. *arXiv preprint arXiv:2407.21787*, 2024. doi: 10.48550/arXiv.2407.21787. URL https://arxiv.org/abs/2407.21787.

Songcheng Cai, Zhiheng Lyu, Yuansheng Ni, Xiangchao Chen, Baichuan Zhou, Shenzhe Zhu, Yi Lu, Haozhe Wang, Chi Ruan, Benjamin Schneider, Weixu Zhang, Xiang Li, Andy Zheng, Yuyu Zhang, Ping Nie, and Wenhu Chen. Swe-qa-pro: A representative benchmark and scalable training recipe for repository-level code understanding, 2026. URL https://arxiv.org/abs/2603.16124.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. Mle-bench: Evaluating machine learning agents on machine learning engineering, 2025. URL https://arxiv.org/abs/2410.07095.

Daewon Choi, Kyunghyun Park, Woomin Song, Saket Dingliwal, Sai Muralidhar Jayanthi, Jinwoo Shin, and Aram Galstyan. Idlespec: Exploiting idle time via speculative planning for llm agents, 2026. URL https://arxiv.org/abs/2605.22154.

François Chollet. On the measure of intelligence, 2019. URL https://arxiv.org/abs/1911.01547.

In Gim, Seung-seob Lee, and Lin Zhong. Asynchronous LLM function calling, 2024. URL https://arxiv.org/abs/2412.07017.

Niko Grupen, Gabe Pereyra, and Julio Pereyra. Open-sourcing harvey's long horizon legal agent benchmark, 2026. URL https://www.harvey.ai/blog/introducing-harveys-legal-agent-benchmark.

Kelly Hong, Anton Troynikov, and Jeff Huber. Context rot: How increasing input tokens impacts LLM performance. Technical report, Chroma, 2025. URL https://research.trychroma.com/context-rot.

Haoyi Hu et al. Anticipate and Learn: Unleashing Idle-Time Compute in Proactive Agents. *arXiv preprint arXiv:2605.25971*, 2026. doi: 10.48550/arXiv.2605.25971. URL https://arxiv.org/abs/2605.25971.

Jacob Xiaochen Li, Rick Battle, and Omar Khattab. Machine studying, Jun 2026. URL https://jacobxli.com/blog/2026/machine-studying/.

Kevin Lin, Charlie Snell, Yu Wang, Charles Packer, Sarah Wooders, Ion Stoica, and Joseph E. Gonzalez. Sleep-time compute: Beyond inference scaling at test-time, 2025. URL https://arxiv.org/abs/2504.13171.

Niklas Muennighoff et al. s1: Simple Test-Time Scaling. *arXiv preprint arXiv:2501.19393*, 2025. doi: 10.48550/arXiv.2501.19393. URL https://arxiv.org/abs/2501.19393.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM Test-Time Compute Optimally Can Be More Effective than Scaling Model Parameters. *arXiv preprint arXiv:2408.03314*, 2024. doi: 10.48550/arXiv.2408.03314. URL https://arxiv.org/abs/2408.03314.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *International Conference on Learning Representations*, 2023.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference Scaling Laws: An Empirical Analysis of Compute-Optimal Inference for Problem-Solving with Language Models. *arXiv preprint arXiv:2408.00724*, 2024. doi: 10.48550/arXiv.2408.00724. URL https://arxiv.org/abs/2408.00724.

Kai Yan, Zhan Ling, Kang Liu, Yifan Yang, Ting-Han Fan, Lingfeng Shen, Zhengyin Du, and Jiecao Chen. Mir-bench: Can your llm recognize complicated patterns via many-shot in-context reasoning?, 2025. URL https://arxiv.org/abs/2502.09933.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems*, 2023a.

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

A Limitations

Our cost accounting counts generated tokens only, although the solver rereads the study as a prompt at every call, and the savings after 1,024 questions extrapolate from at most 30 questions per context. Each benchmark has 22–30 questions. DeepSeek-V4-Flash has 3 seeds, while Qwen3.8-27B, gpt-5.6-luna and the multi-agent Harvey efficiency curves have 1 seed. One LLM judge grades the rubric benchmarks, AIME 2024 and 2025 may be in the models' training data, and we do not compare against a retrieval baseline that loads files without a study. Online, the proactive agent generates more tokens than the reactive agent, and we test significance at fixed time points. Finally, at small test budgets the solver copies a wrong answer planted in the study, so a wrong study can hurt.

B Offline experiments

B.1 Setup

Models. Table 2 lists the three models. Every model runs the same benchmarks, prompts and proactive budgets.

Budgets. P and T count generated tokens, reasoning included. A call that stops early is continued with one of three continuation prompts in rotation. A call that reaches its budget is cut, and the

Table 1: **The nine offline benchmarks.** The three AIME years count as separate benchmarks. # Qs is the number of questions in the main grid. On AIME, HMMT, ARC-AGI-1 and MIR-Bench each context holds one question, while StudyBench DSPy is one repository with 30 questions, SWE-QA-Pro is five repositories with 6 questions each, and Harvey LAB is one document store. The tool-use benchmarks define their own context–query split, and the math split follows Lin et al. (2025).

Benchmark	Domain	Tools	# Qs	Context	Answer	Evaluation
AIME (2024–26)	Math	✗	26/26/22	Truncated question	Integer $[0, 999]$	Exact match
HMMT Feb 2026	Math	✗	28	Truncated question	Exact value	Flexible match
ARC-AGI-1	Algorithmic	✗	30	Input–output pairs	Digit grid	Exact match
MIR-Bench	Algorithmic	✗	30	Input–output pairs	Python literal	Flexible match
StudyBench DSPy	SWE	✓	30	Repository	Free-text answer	Rubric + LM judge
SWE-QA-Pro	SWE	✓	30	Repository	Diagnosis/proposal	Gold + LM judge
Harvey LAB	Legal	✓	30	Law-firm filesystem (134M tokens)	Legal work product	Rubric + LM judge

Table 2: **Models on the offline grid.** All three models run the proactive budgets $P \in \{0, 64, 256, 1\text{k}, 4\text{k}, 16\text{k}, 32\text{k}, 64\text{k}, 128\text{k}\}$.

Model	Served by	Test budgets T	Seeds
DeepSeek-V4-Flash-0731	OpenRouter (StreamLake)	64 to 64k	3
Qwen3.8-27B	local vLLM	64 to 64k	1
gpt-5.6-luna	OpenRouter (OpenAI)	64 to 16k	1

solver is then asked for its final answer (Appendix F). Each context and seed has one study chain at the largest P, and the smaller budgets are its prefixes.

Tools. The study and the solver have the same tools and caps, and only the solver sees the question. On SWE-QA-Pro and StudyBench this is one `browse` tool (`ls`, `read` of at most 120 lines and `grep` with at most 40 hits, with every observation cut at 4,000 characters). On Harvey LAB it is the benchmark's six tools (bash, read, write, edit, glob, grep) in a sandbox without network access, with the documents mounted read-only.

Grading. GLM-5.3-Flash grades StudyBench, SWE-QA-Pro and Harvey LAB in one call per answer at medium reasoning effort (Appendix F.6). Intervals are 95% bootstrap intervals over questions unless a caption says otherwise.

B.2 The efficiency metric

Test-time efficiency E is defined in §4. On the offline grid it weights $T = 64, 256, 1\text{k}, 4\text{k}, 16\text{k}, 32\text{k}, 64\text{k}$ by $0.10, 0.20, 0.20, 0.20, 0.15, 0.10, 0.05$. A plain mean of the seven accuracies would put 43% of the weight on $T \geq 16\text{k}$, where the curves have converged. Figures with $T \leq 16\text{k}$ apply the same rule on that window.

Why not the expertise metric. The expertise metric of Li et al. (2026) weights accuracy by a density that falls with T, and on our grid it puts about three quarters of its weight on $T = 64$. Equal weight per doubling also has a compute reading. For curves that rise with T, the gain in E times the window width in doublings is the number of doublings of test compute that the study saves, averaged over accuracy levels. This weighting also keeps gains at intermediate budgets visible.

B.3 Why we budget generated tokens

We budget generated tokens, reasoning included, for two reasons. First, decoding sets the latency and most of the price of a call, because the model generates one token per forward pass, while the

prompt is processed in parallel and a repeated prefix is served from the prefix cache. For DeepSeek-V4-Flash an output token costs 2–3× a prompt token and 9–86× a cached prompt token, depending on the provider. Second, generated tokens measure how much the model thinks, and budget forcing controls them exactly, whereas the prompt mostly holds what the environment returned. On the tool-use benchmarks the prompt still grows with the budget, because each call resends the conversation (Figure 9), but prompt and generated tokens order the test budgets the same way.

Prompt tokens of a shared study. The amortization in §5.3 counts generated tokens only. Every question on a context reads the same study as a prompt prefix, which can be computed once and then served from the prefix cache.

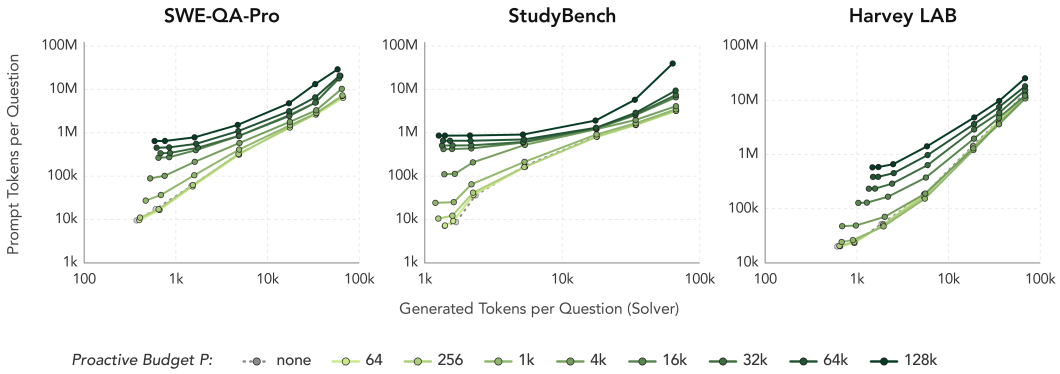

Figure 9: **Prompt tokens rise with generated tokens, and the study adds a fixed offset.** Solver prompt tokens against generated tokens per question on the three tool-use benchmarks (log–log). Each line is one proactive budget P, and its points are the test budgets T from 64 to 64k. The offset between lines is the study, which the solver rereads at every call. DeepSeek-V4-Flash, 30 questions per benchmark, 3 seeds.

B.4 Other model families

Qwen3.8-27B and gpt-5.6-luna run the same grid with 1 seed each (Figure 10).

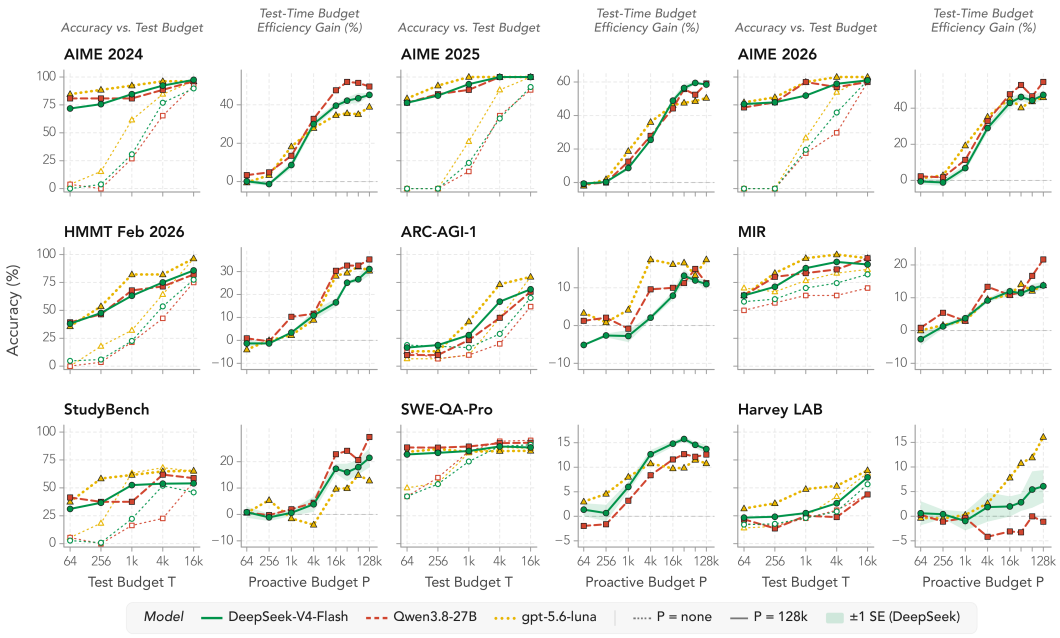

Figure 10: **Qwen3.8-27B and gpt-5.6-luna gain from the study like DeepSeek-V4-Flash on almost all benchmarks.** For each benchmark, the left plot shows accuracy against T without a study and with a 128k study, and the right plot shows the gain $E(P) - E(0)$ against P. E is over $T \leq 16$k for all three models so that gpt-5.6-luna is comparable. 22–30 questions per benchmark, 3 seeds for DeepSeek-V4-Flash and 1 seed for Qwen3.8-27B and gpt-5.6-luna.

B.5 Full grids

For each model we show the grid in four ways, as accuracy against T with one line per P (Figures 11–13), as a heatmap of every cell (Figures 14–16), as accuracy against P with one line per T (Figures 17–19), and as the efficiency gain against P (Figures 20–22). The three models agree. On math and on the repository benchmarks the study lifts the small test budgets, and the lift fades as T grows. On ARC-AGI-1 the lift sits at intermediate T, and on MIR and Harvey LAB it is small.

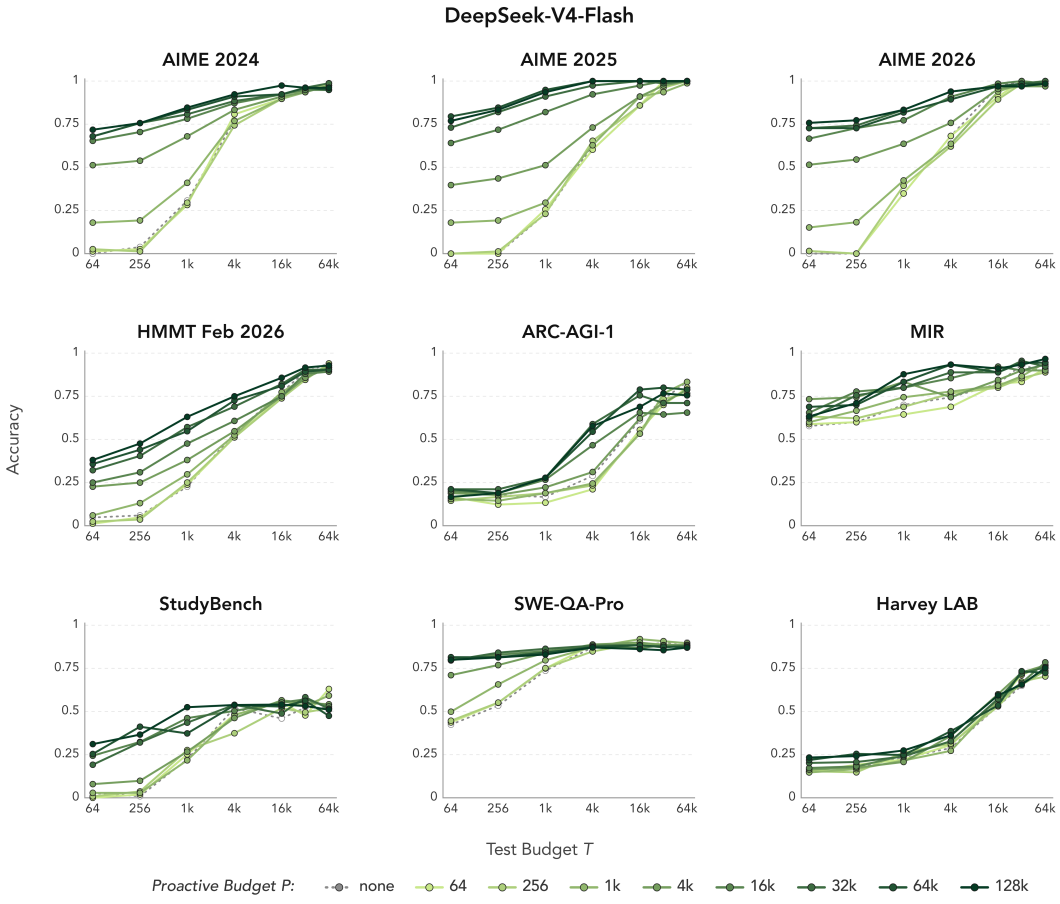

Figure 11: **Accuracy of DeepSeek-V4-Flash against the test budget T.** One line per proactive budget P (light to dark for small to large P, dotted grey for no study), over T. 22–30 questions per benchmark, 3 seeds.

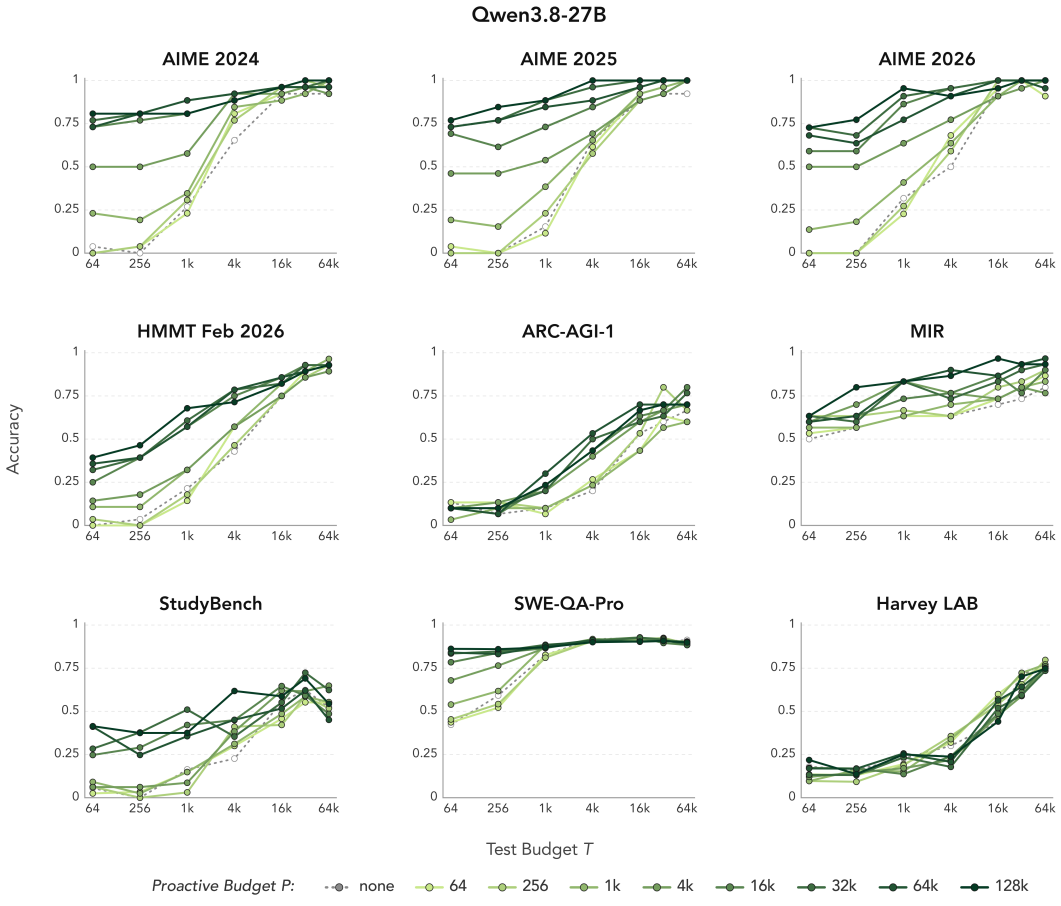

Figure 12: **Accuracy of Qwen3.8-27B against the test budget T.** One line per proactive budget P (light to dark for small to large P, dotted grey for no study), over T. 22–30 questions per benchmark, 1 seed.

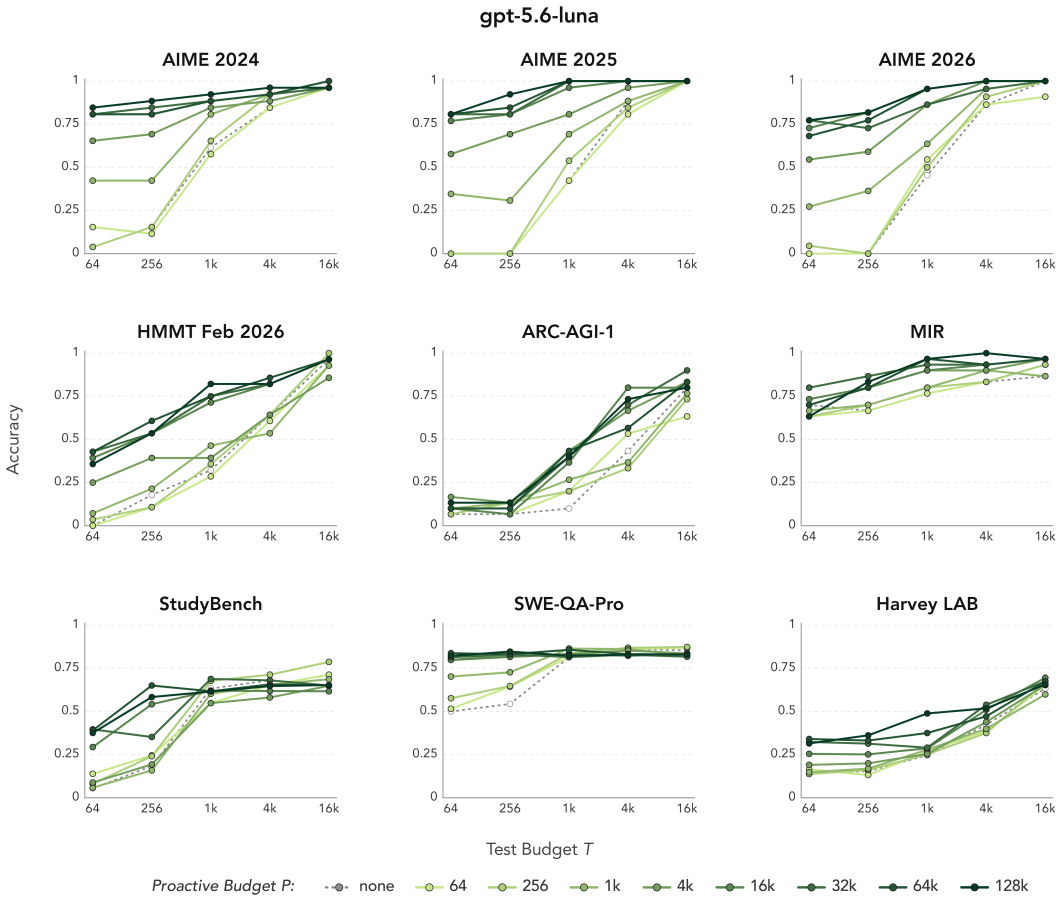

Figure 13: **Accuracy of gpt-5.6-luna against the test budget T.** One line per proactive budget P (light to dark for small to large P, dotted grey for no study), over $T \leq$ 16k. 22–30 questions per benchmark, 1 seed.

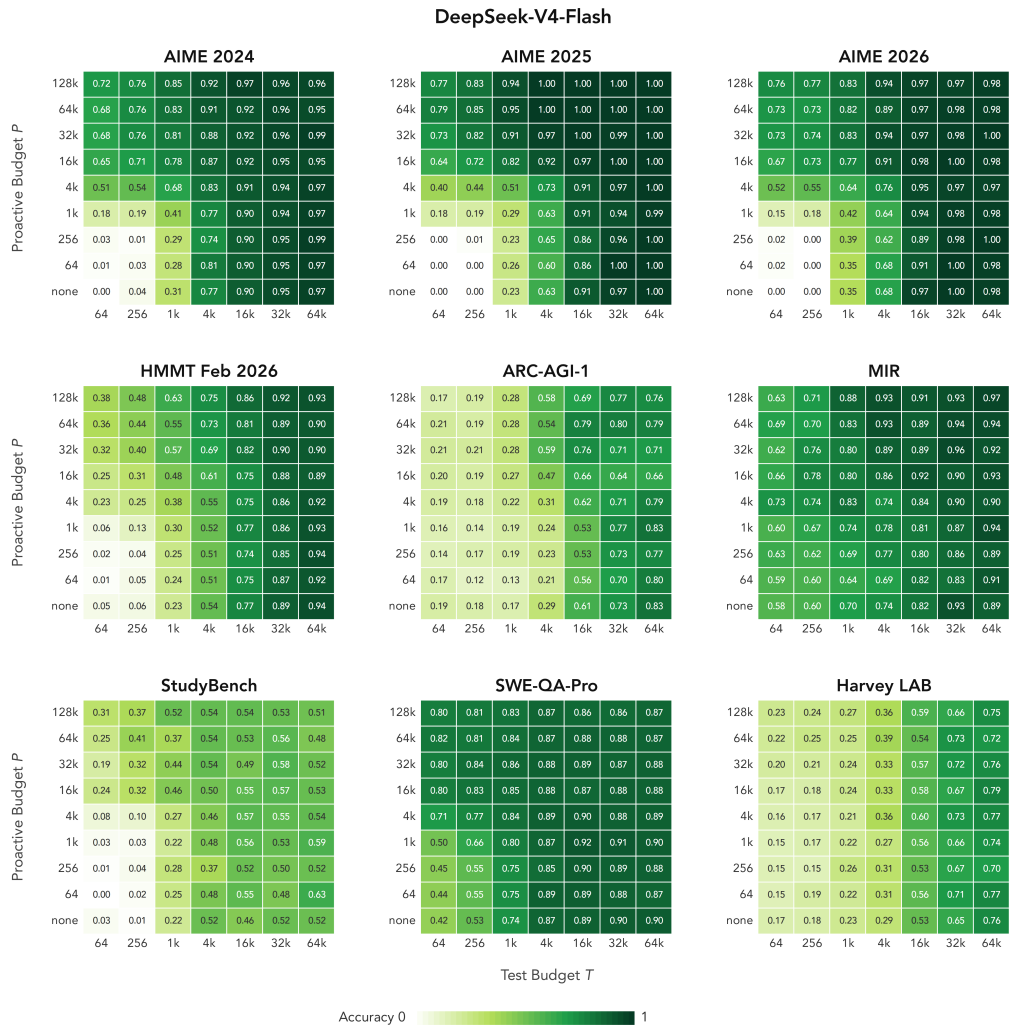

Figure 14: **Accuracy of DeepSeek-V4-Flash over the full $P \times T$ grid.** Each cell is the accuracy at proactive budget P (rows, no study at the bottom) and test budget T (columns). 22–30 questions per benchmark, 3 seeds.

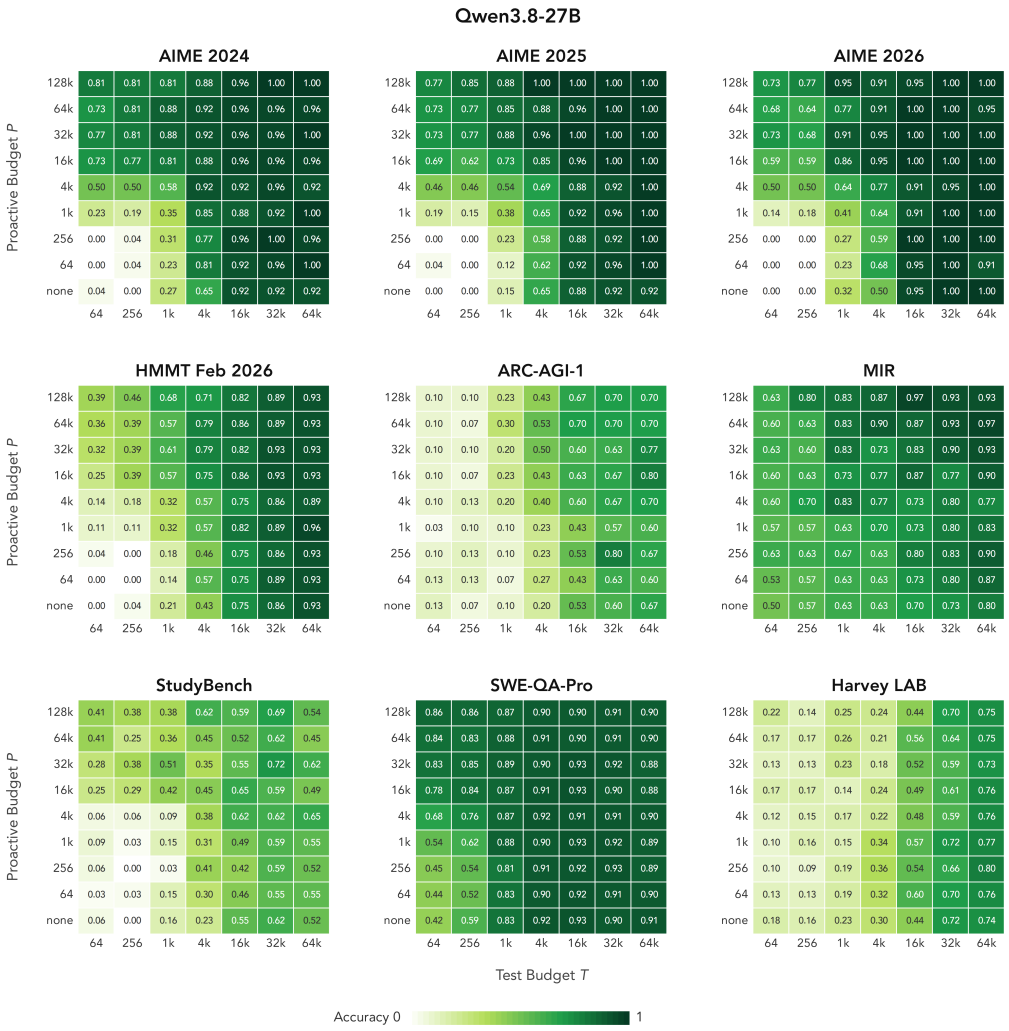

Figure 15: **Accuracy of Qwen3.8-27B over the full $P \times T$ grid.** Each cell is the accuracy at proactive budget P (rows, no study at the bottom) and test budget T (columns). 22–30 questions per benchmark, 1 seed.



Figure 16: **Accuracy of gpt-5.6-luna over the full $P \times T$ grid.** Each cell is the accuracy at proactive budget P (rows, no study at the bottom) and test budget $T \leq$ 16k (columns). 22–30 questions per benchmark, 1 seed.

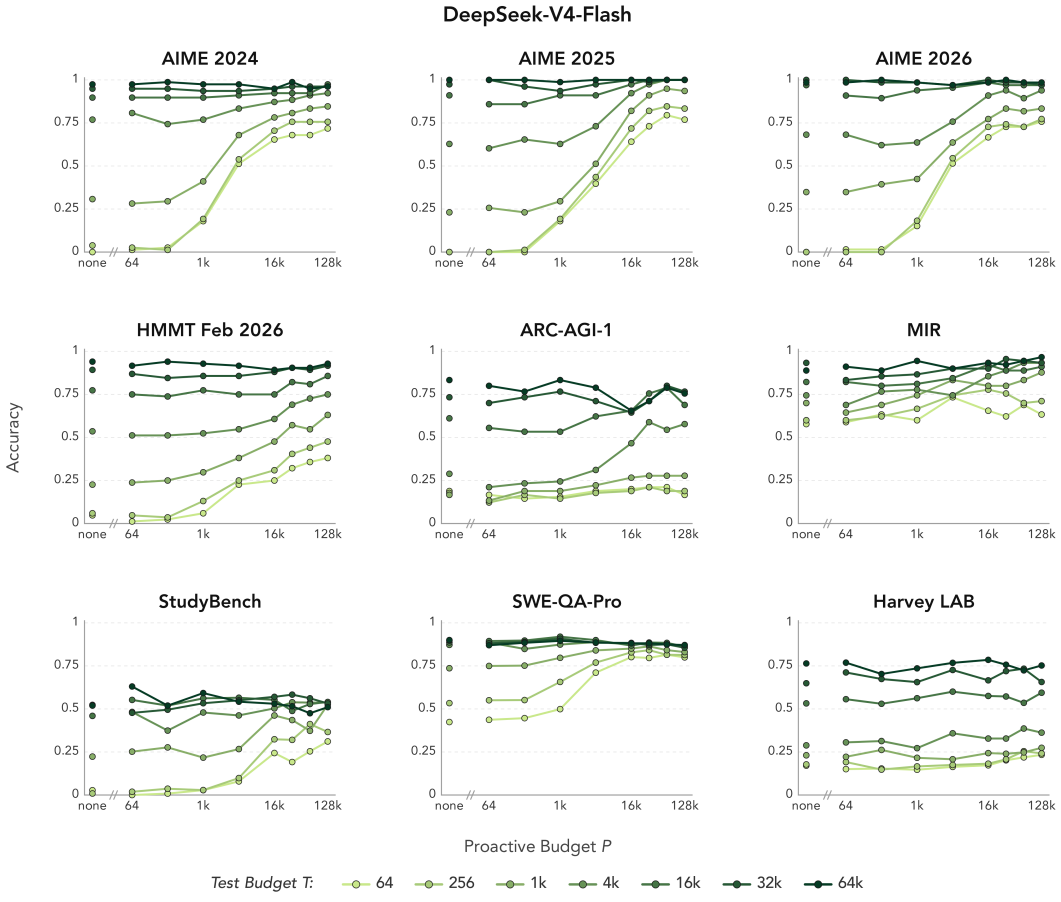

Figure 17: **Accuracy of DeepSeek-V4-Flash against the proactive budget P.** The cells of Figure 14, with one line per test budget T (light to dark for small to large T). 22–30 questions per benchmark, 3 seeds.

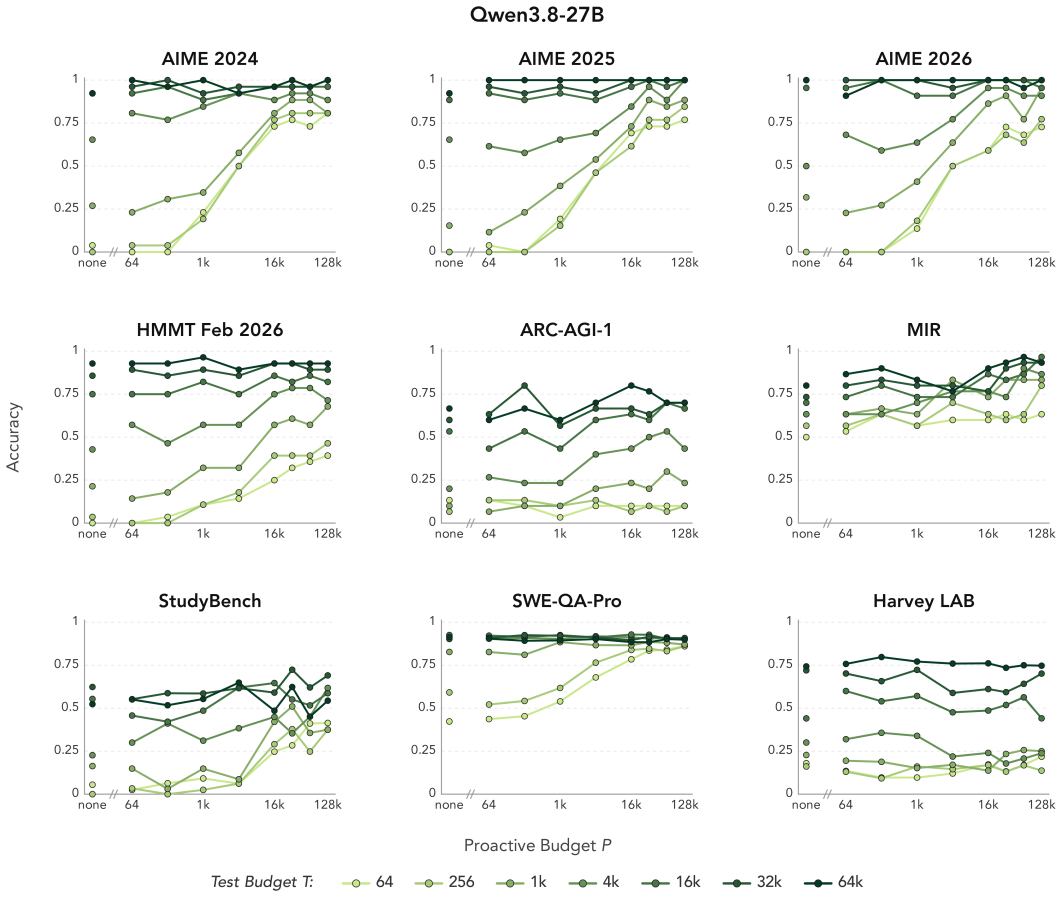

Figure 18: **Accuracy of Qwen3.8-27B against the proactive budget P.** The cells of Figure 15, with one line per test budget T (light to dark for small to large T). 22–30 questions per benchmark, 1 seed.

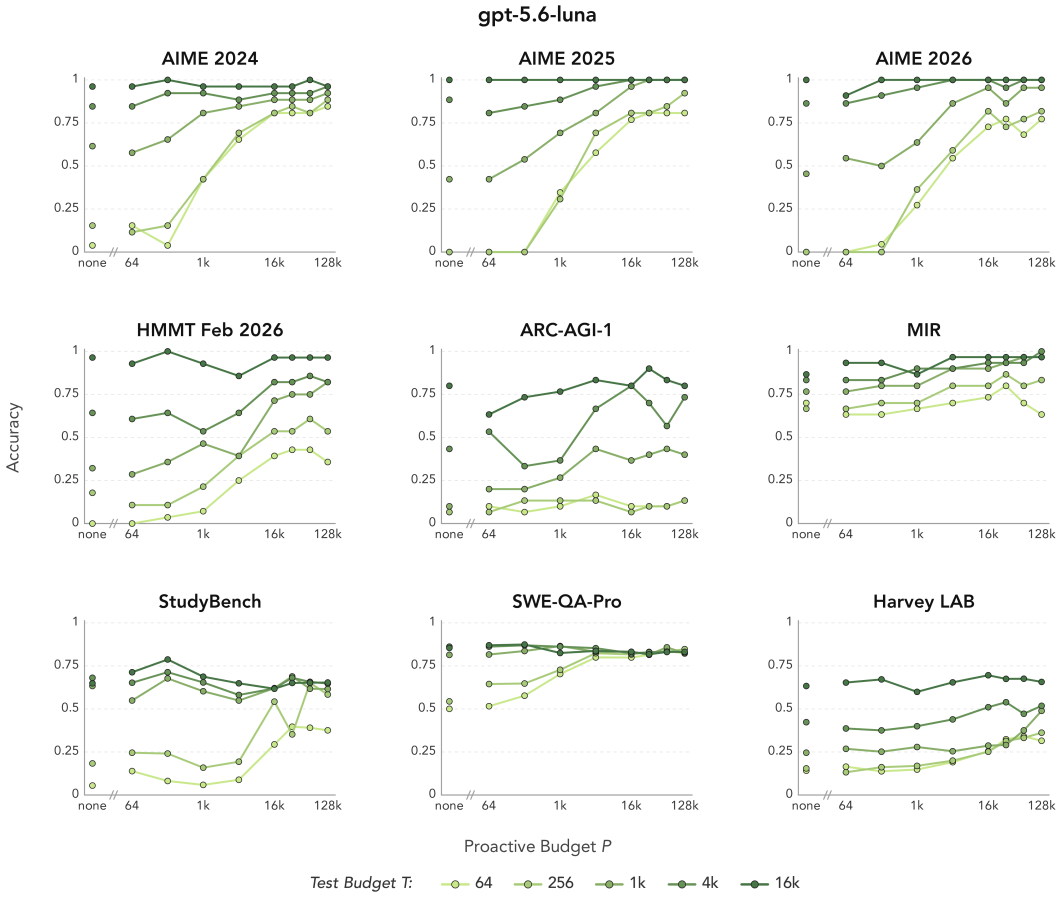

Figure 19: **Accuracy of gpt-5.6-luna against the proactive budget P.** The cells of Figure 16, with one line per test budget $T \leq$ 16k (light to dark for small to large T). 22–30 questions per benchmark, 1 seed.

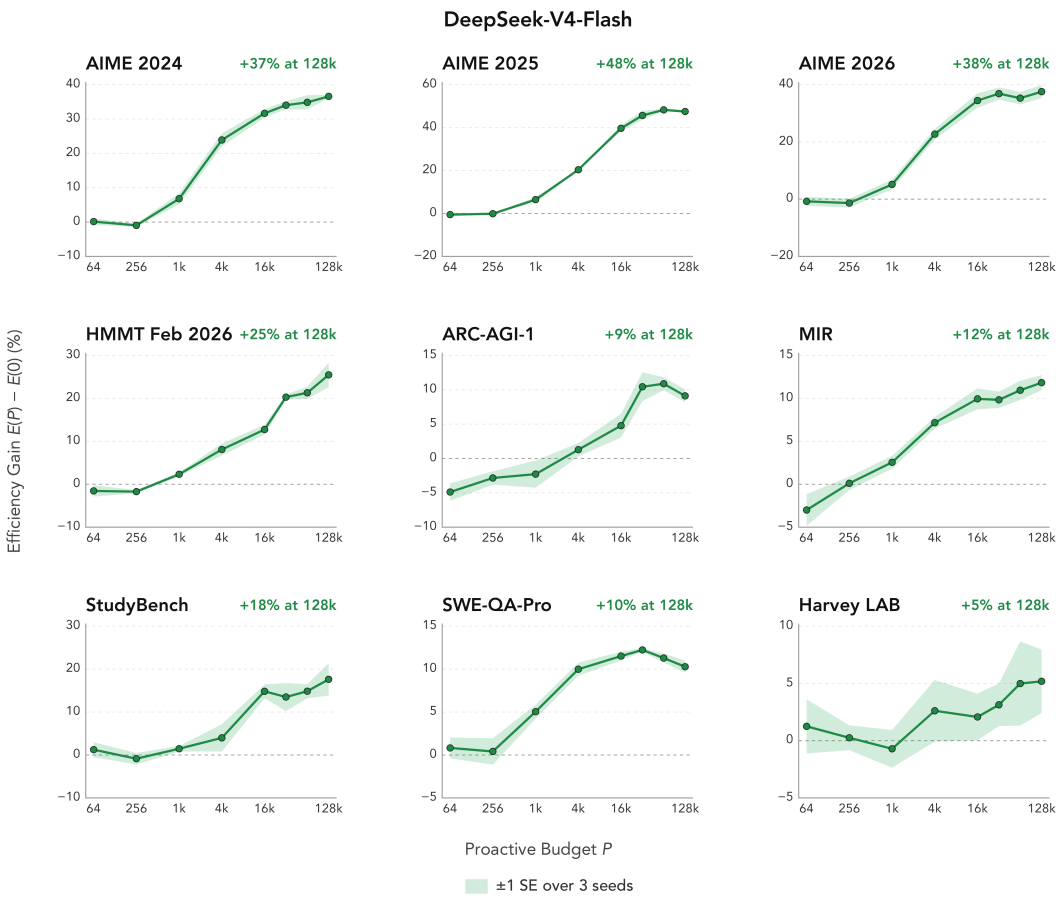

Figure 20: **Efficiency gain of DeepSeek-V4-Flash against the proactive budget P.** $E(P) - E(0)$ in percentage points, with E over T from 64 to 64k, and the gain at $P = 128$k is printed. The band is ± 1 SE over seeds. 22–30 questions per benchmark, 3 seeds.



Figure 21: **Efficiency gain of Qwen3.8-27B against the proactive budget P.** $E(P) - E(0)$ in percentage points, with E over T from 64 to 64k, and the gain at $P = 128$k is printed. 22–30 questions per benchmark, 1 seed.

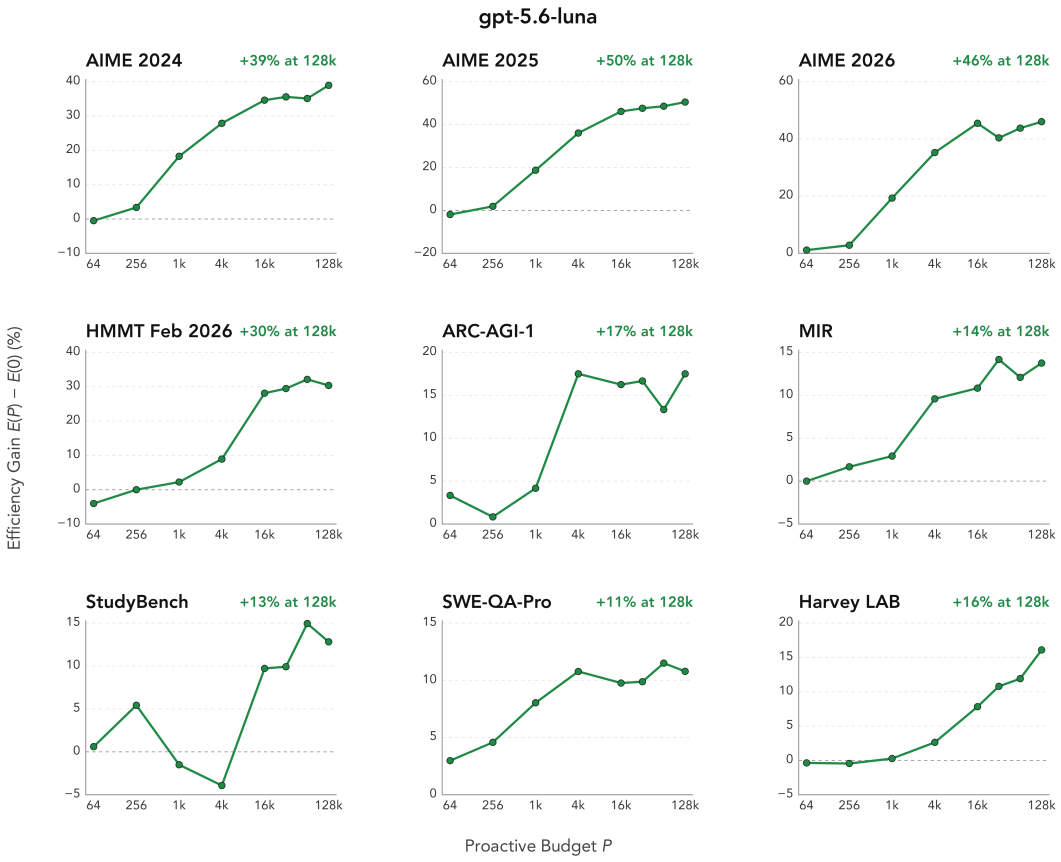

Figure 22: **Efficiency gain of gpt-5.6-luna against the proactive budget P.** $E(P) - E(0)$ in percentage points, with E over $T \leq 16$k, and the gain at $P = 128$k is printed. 22–30 questions per benchmark, 1 seed.

C ONLINE EXPERIMENTS

C.1 SETUP

A DeepSeek-V4-Flash shell agent works on one MLE-bench task for 30 minutes of wall-clock time per session, on Denoise (RMSE), Statoil Iceberg (log loss) or LMSYS Chatbot Arena (3-class log loss). Each of 28 seeds per task runs once with the reactive agent and once with a proactive agent at each budget of 128, 512, 2k and 8k tokens per wait. For every command that runs longer than 3 s, the reactive agent receives review-and-plan guidance with the command's result. The proactive agent receives equivalent guidance while the command runs, one prompt when the wait starts and a continuation prompt for each further round (Appendix F.3), and generates until the command returns or it reaches its budget. We grade the best submission so far with the task's grader, compare agents with the paired Wilcoxon signed-rank test, and correct over the three tasks with Holm's method. An API stall is a call that takes more than 30 s at fewer than 25 tokens per second.

Denoise split. The stock MLE-bench split leaks test targets. Each document has 4 scans of one clean page, so the target of a test scan appears in training under a sibling scan. We hold out 12 documents, one scan each, and train on the 96 pairs of the other 24.

C.2 SESSION TIMELINES

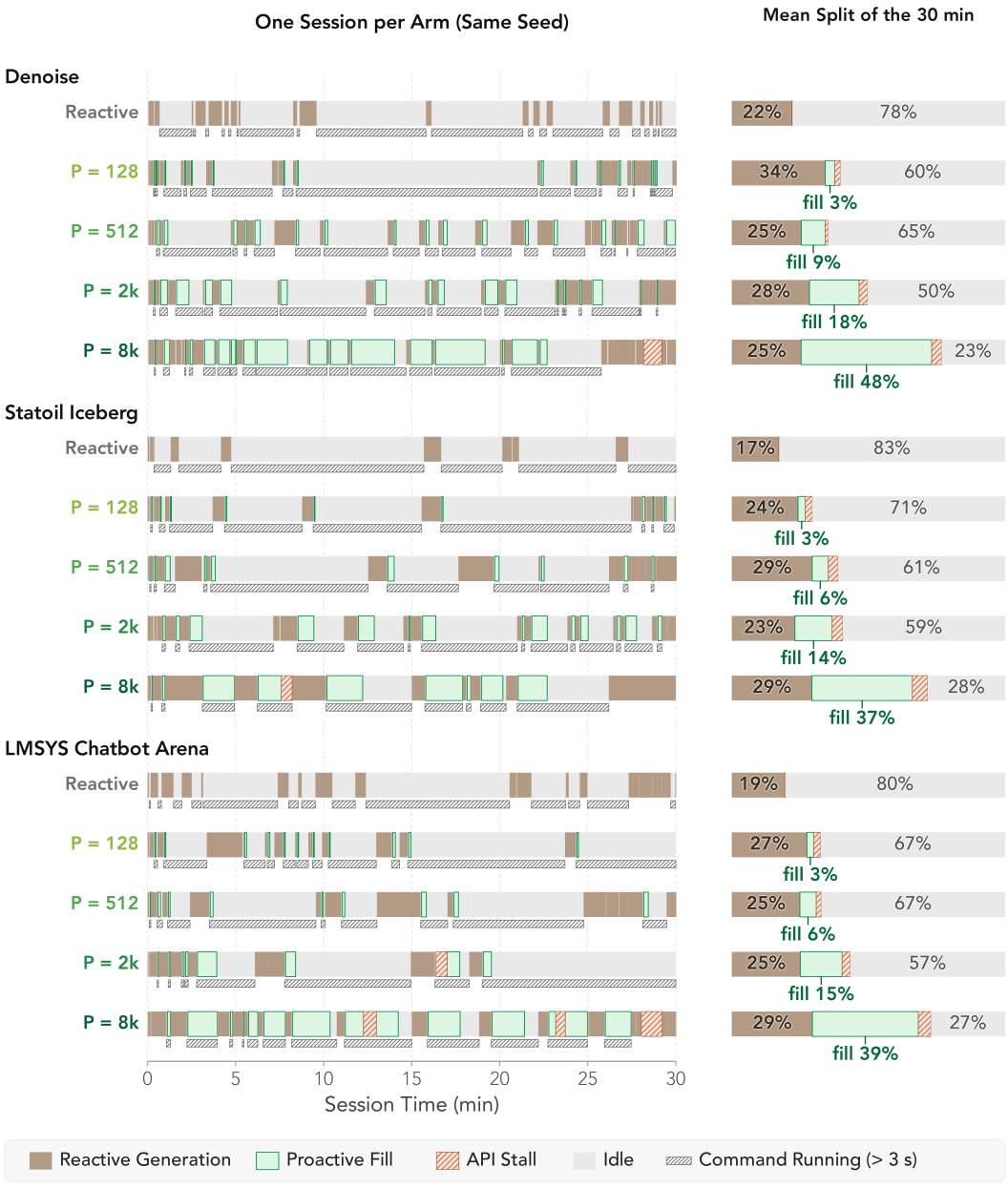

Figure 23: **Larger budgets per wait fill more of the time the reactive agent spends idle.** The left panel shows one 30-minute session per agent, all on the same seed, for the reactive agent and proactive agents with 128 to 8k tokens per wait. Brown is reactive generation, green is proactive generation while a command runs, hatched orange is an API stall, grey is idle time, and the hatched strip below each row marks a command running for more than 3 s. The right panel shows the mean share of the 30 minutes in each state, stalls included. DeepSeek-V4-Flash, 3 tasks (Denoise, Statoil Iceberg and LMSYS), 28 seeds per task and agent.

D HARVEY LAB

D.1 SETUP

The context is the benchmark's document store of 9,288 documents in 266 matters. The GLM-5.3-Flash judge grades each answer against its question's rubric in one call, and the score is the fraction of criteria passed. E is over $T \leq 16$k. Figure 6 uses 120 questions. The plain and wiki-prompted single studies have 3 seeds each, and the multi-agent studies have 1 seed on the left panel. In the multi-agent studies each agent has a budget of 128k (131,072) tokens. Parallel agents each study one partition of the matters. Each serial round receives the notes of the earlier rounds in its prompt (Appendix F.4). After its study, every agent writes notes, and the solver receives the notes of all agents.

Harvey LAB in Figure 2. We first ran Harvey LAB with the general study prompt of the other benchmarks (Appendix F.1). Figure 2 instead shows the wiki-prompted study (Appendix F.4) on 30 of the 120 questions, with 3 seeds and $T \leq 16$k. The full grids and the other models (Appendices B.4 and B.5) use the general prompt.

D.2 BREADTH OF PARALLEL AND SERIAL STUDIES

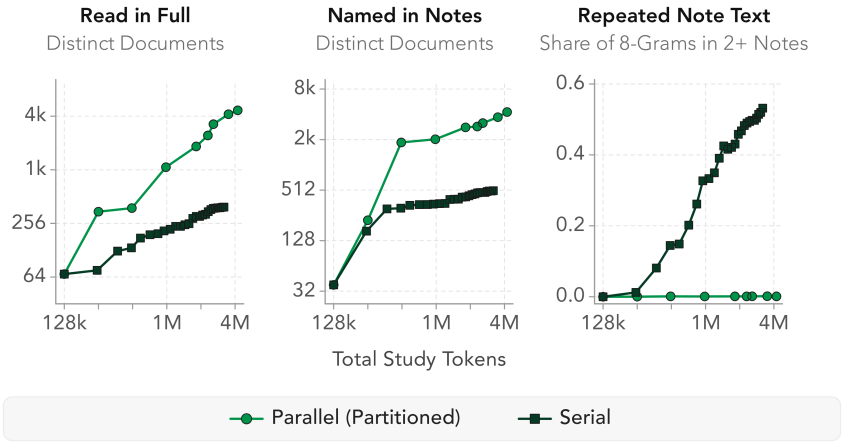

Figure 24: **Parallel agents read and name far more distinct documents than serial rounds, whose notes repeat each other.** Harvey LAB multi-agent studies against total study tokens, summed over agents or rounds (parallel up to $N = 40$ agents, serial up to 26 rounds, 128k tokens each). The left panel shows distinct documents read in full, the middle panel distinct documents named in the notes, and the right panel the share of note 8-grams (runs of 8 words) that appear in two or more notes. DeepSeek-V4-Flash, 120 questions, seed 1.

E ANALYSIS

E.1 STUDY NODES

Two analyses below describe a study as a sequence of *events* on *nodes*. We extract them from the study trace with fixed rules, without a model.

- **Math.** A node is an integer of at least 10 that appears in the study but not in the context. Each occurrence of the integer is an event.
- **Repositories.** A node is a file. Each `browse` call (`read`, `ls` or `grep`) on a path is an event on that file.

A node is *new* at its first event, and each later event on it is a *revisit*. *Gold* nodes are the answer integer on math and, on SWE-QA-Pro, the files cited in the gold answers. In these analyses, HMMT pools the February 2025, November 2025 and February 2026 contests (75 problems). We first

tried to cut studies into approaches with model annotators, but two annotators split the same study differently (Figure 25), so we use rule-based nodes instead.



Figure 25: **Two model annotators cut the same study into different approaches.** One AIME 2025 study by DeepSeek-V4-Flash (seed 2), segmented into approaches by GLM-5.3-Flash and by DeepSeek-V4-Flash with the same annotation prompt. Each lane is one technique. The two segmentations disagree, so we use rule-based nodes instead.

E.2 Why the gain grows slowly past 16k

The gain in E grows slowly past $P = 16$k (Figure 2). The study nodes suggest why. The study finds new nodes 6–12× more slowly at its end than in its first 1k tokens (Figure 26), and on math most late events revisit nodes found earlier (Figure 27). Figure 28 shows one repository study that has read every gold-cited file by $P = 16$k.

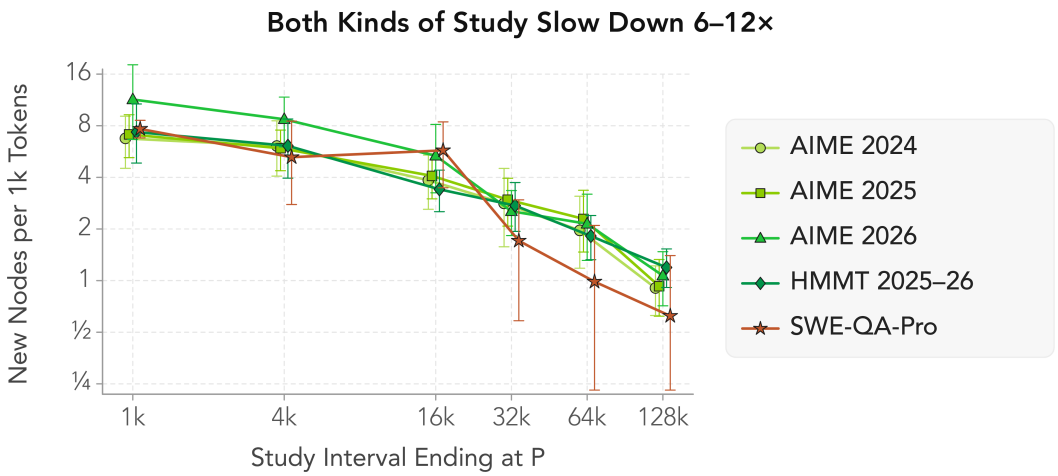

Figure 26: **The study finds new nodes 6–12× more slowly at its end than in its first 1k tokens.** New nodes per 1k study tokens in each interval of the study, by the budget P at which the interval ends. 95% bootstrap over contexts (over repositories on SWE-QA-Pro). DeepSeek-V4-Flash, 74 AIME 2024–26 problems, 75 HMMT problems and 30 SWE-QA-Pro questions in 5 repositories, 3 seeds.

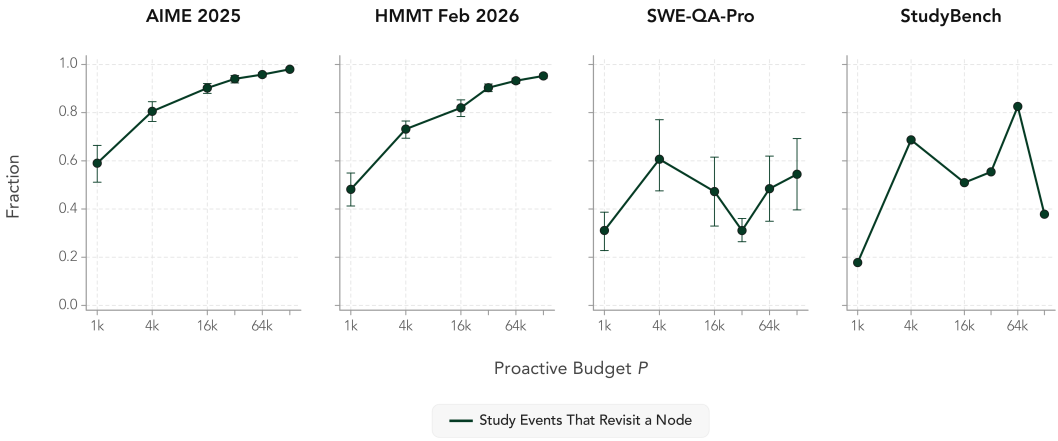

Figure 27: **On math, most late study events revisit nodes the study has already found.** Share of study events in each interval, ending at budget P, that revisit an earlier node. 95% bootstrap over contexts (over repositories on SWE-QA-Pro, and none on StudyBench, which is one repository). DeepSeek-V4-Flash, 26 AIME 2025 problems, 75 HMMT problems and 30 questions each on SWE-QA-Pro and StudyBench, 3 seeds.

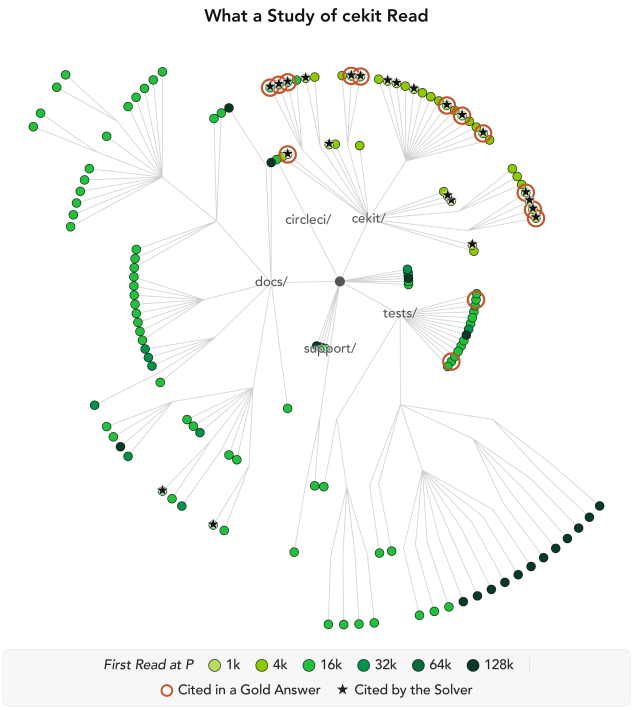

Figure 28: **One study reads every gold-cited file by $P = 16$k and no gold-cited file after it.** Files read by one SWE-QA-Pro study of the cekit repository, placed in the repository's directory tree and colored by the budget P at which the study first reads them. An orange ring marks a file cited in a gold answer, and a star marks a file cited by the solver at $P = 16$k. DeepSeek-V4-Flash, the repository's 6 questions, seed 0.

E.3 Does the solver trust the study?

We plant errors in the $P = 16$k study of each AIME problem and measure how often the solver adopts them (Figure 29).

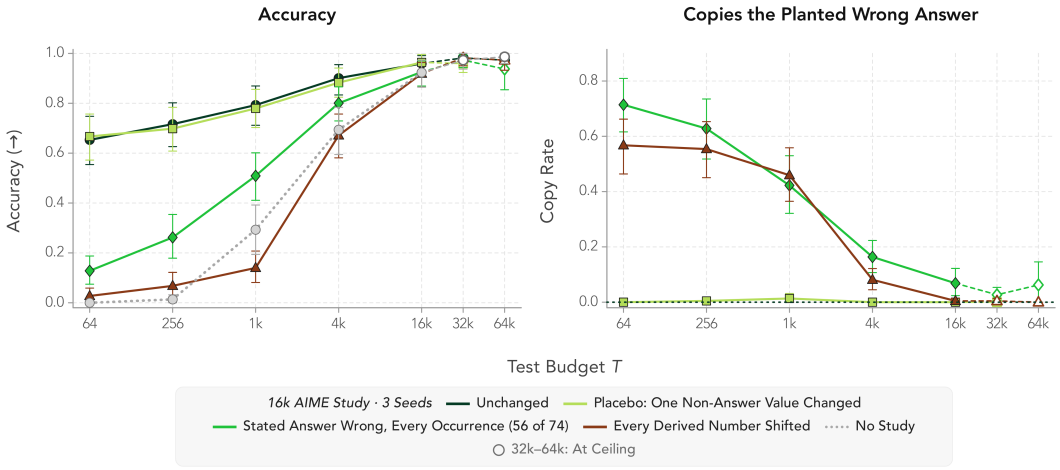

Figure 29: **At small T the solver copies a wrong answer planted in the study, and by $T = 16$k a corrupted study is worth no study.** We edit the $P = 16$k study in three ways. A placebo changes one number other than the answer, a wrong value replaces the stated answer at every occurrence (on the 56 problems whose study states the answer), and a shift moves every derived number. The left panel shows accuracy and the right panel the share of answers equal to the planted value. Hollow points ($T \geq 32$k) are at ceiling. 95% bootstrap over problems. DeepSeek-V4-Flash, 74 AIME 2024–26 problems, 3 seeds (1 at $T = 64$k).

E.4 Compressing the study

Summaries. The same model reads the 128k study trace, without the question, with the compression prompt (Appendix F.5), and writes a summary budget-forced to $M \in \{256, 1\text{k}, 4\text{k}, 16\text{k}, 32\text{k}, 64\text{k}\}$ tokens, which the solver receives in place of the trace. **Native studies** give the solver the prefix of the same chain at $P = M$. **Retention** is $(E_M - E_0)/(E_{128\text{k}} - E_0)$, with E over $T \leq 16$k and 3 seeds (Figure 8).

Why the native points do not sit at powers of two. The x-axis of Figure 8 counts the study tokens the solver receives. P counts generated tokens only, but on the repository benchmarks the trace also holds the outputs of the study's tool calls, so a native study at $P = M$ delivers several times M tokens (about 40k at $P = 4$k on SWE-QA-Pro). A summary holds no tool outputs and delivers about M tokens, as do both arms on math, where the study calls no tools.

F Prompts

This section gives the exact assembled prompts used to produce the paper's results. Benchmark contexts and questions are replaced by `{placeholders}`, and no benchmark data (documents, questions or context text) is reproduced. Where a template is shared across benchmarks, it is shown once with placeholders and the per-benchmark slot values are given in a table. Each prompt sits in one box. The box color marks the phase, with **teal** for proactive (study) prompts, **navy** for test-time (solver) prompts, **amber** for the compressor, and **grey** for environment output and grading. The label on the right of each title gives the message type.

F.1 Study (proactive) prompts

The study agent uses the same system prompt, continuation nudges, and memory-block labels on every benchmark; only the context slot differs (main offline grid, DeepSeek-V4-Flash).

Study · System prompt *system*

```
You will be given some context material. Questions about it will be asked later,
but you will not see them -- a different agent will, and that agent will receive
your notes.

Your task is to study the context and produce useful knowledge, inferences,
derived quantities, and reasoning that could help that future agent answer
questions quickly and correctly.

Make as many inferences as possible about the material you are given. You should:
- derive intermediate quantities and results that plausibly matter, and record
  them with the values you computed;
- map out the structure of the material;
- investigate anything suspicious, inconsistent, or error-prone, and note the
  mistakes or pitfalls you find explicitly;
- predict the kinds of questions that could plausibly be asked, and prepare for
  them by working out what they would need;
- double-check your earlier calculations each time you continue, and note which
  earlier findings turned out not to be useful.
```

Study · First user turn *user*

```
Here is the context:

<context>
{context text}
</context>
```

On tool-use benchmarks (StudyBench, SWE-QA-Pro, Harvey LAB) a tool-access line is appended after the context block. The Harvey LAB line is shown below.

Study · Tool-access line (Harvey LAB example) *appended to user turn*

```
You cannot see the firm's documents directly -- the document store is far too large.
Explore it with the tools: it is mounted read-only at /workspace/documents (matters/{
matter-id}/{folder}/{file}); `read` returns the text of .docx, .xlsx, .pptx, .pdf and
plain files; `glob` and `grep` search it; `bash` runs in the same sandbox; /workspace is
writable scratch space.
```

Continuation nudges are used in rotation whenever a study or solver call stops before its budget.

Study · Continuation nudges *injected, in rotation*

```
Keep reasoning. Do not answer yet.
Continue your reasoning from where you stopped. Do not answer yet.
Keep thinking this through. Do not answer yet.
```

Memory-block labels are used when a written-up study record, instead of the raw trace, is shown to the solver.

Solver · Memory-block labels (written-up record) *inside user turn*

```
Study record of an earlier agent that examined this material without seeing the question:
[note]
[query {name}]
[result]
```

F.2 Solver prompts and answer forcing

Every benchmark shares one solver template. It has a system prompt (task framing plus a fixed memory-block disclaimer), a user turn with context, memory block and question, and a forced an-

swer turn once the test budget is spent. Only the task-framing sentence and the answer-format sentence vary, and both are given per benchmark in Table 3.

Solver · System prompt (template) *system*

```
{task framing sentence(s): per-benchmark table below}

A memory block may be provided. It is the record of an earlier agent that
studied the same material without seeing your question -- it was told only
that questions would be asked later, by someone else. It may contain results,
structure or warnings that bear on your question, or nothing that does. Use it
as you judge best.
```

Solver · User turn (template) *user*

```
<context>
{context text}
</context>

Memory block:
<memory>
{memory text}
</memory>

<question>
{question}
</question>

Reason out loud, step by step. Do not state a final answer yet.
```

The answer-forcing turn is injected once the test budget is spent and tools are withdrawn.

Solver · Answer-forcing turn (template) *injected*

```
Stop. The reasoning phase is over. Do not reason further, do not explain, and
do not restate the question. Tools are no longer available: do not call a
tool and do not type tool-call syntax. Based only on the reasoning above,
output the final answer immediately. If your reasoning did not reach a
definite answer, output your single best guess in the required format.

Answer format: {answer format: per-benchmark table below}

Write the answer between A SINGLE PAIR of <answer> and </answer> tags: the
tags appear exactly once, everything belongs inside them, and nothing at all
is output outside them. Do not wrap the whole answer in ``` fences.
```

Table 3: Per-benchmark slot values for the solver template above.

Benchmark	Task-framing sentence(s)	Answer format
AIME 24/25/26	You are an expert competition mathematician. Work carefully and precisely; final answers to these problems are integers from 0 to 999.	A single integer between 0 and 999 – no words, no units, no LaTeX.
HMMT Feb 2026	You are an expert competition mathematician. Work carefully and precisely. The final answer may be an integer, a fraction, or an exact expression (radical or multiple of π).	The exact value as one expression on one line – integer, fraction, or closed form. No decimal approximation of an exact value.
ARC-AGI-1	You are solving an abstract reasoning puzzle. The context holds the demonstration pairs. Each grid is rows of digits 0–9, where 0 is the background. Infer the single transformation rule that maps every input grid to its output grid, then apply it.	The output grid only – one row per line, digits separated by single spaces, no labels and no commentary.
MIR-Bench	The context holds many input→output examples of a hidden function. Infer what the function does, then apply it to the target input.	The output value as one strict Python literal, matching the wrapping style of the example outputs exactly.
StudyBench (DSPy)	You are answering a question about the dspy software repository. Use the `browse` tool to read the repository as needed: `ls`, `read`, `grep`.	A self-contained, runnable solution – the complete code itself, not a description of it.
SWE-QA-Pro	You are answering a question about the software repository shown in the user turn. Use the `browse` tool as needed.	The final answer's content only, no code blocks; evidence cited as `<relative_path>: line <start>-<end>`.
Harvey LAB	You are an AI agent executing a task provided by the user within a workspace. The task concerns the firm's document management system (DMS), mounted read-only; tools: bash, read, write, edit, glob, grep.	A single self-contained markdown document with findings, the matters/documents relied on, and reasoning.

F.3 ONLINE AGENT PROMPTS

Both online agents (DeepSeek-V4-Flash, §6) receive the same system prompt and first user turn. For every command that runs longer than 3 s, the proactive agent receives the waiting prompt when the wait starts and a continuation prompt for each further round of the same wait, and the reactive agent receives equivalent guidance with the command's result. The system prompt holds the MLE-bench agent instructions and then the tool description.

Online agents · System prompt (both agents) *system*

```
You are participating in *MLE-bench*, an offline version of Kaggle competitions adapted to
 test the coding ability of agents like yourself.

You will be given a machine learning task. You must solve the task by training a model and
 running the model on the test set to produce a submission file. You may not hand-label
the data to produce a submission file.

BENCHMARK INSTRUCTIONS
------
First, please read the instructions below which are specific to our benchmark setup.

- **Competition-specific instructions**: The instructions for the competition are
available in `data/description.md` (also reproduced below). Read this file to understand
the problem setup for the competition you are currently solving.
- **Dataset**: The dataset files for the competition are available in `data/`.
- **Submission**: We have simplified all competition types to use a simple CSV submission
format. Regardless of what the competition instructions say, you MUST produce a submission
 file at `submission/submission.csv`. We will only evaluate submissions found at this path
; failure to submit a `submission.csv` at this path will result in a failed attempt.
- **Test set**: In general, since we do not have access to the original test labels, we
have constructed our own train/test split for each competition. As such, the dataset you
see in `data/` may differ slightly from what is described in the competition instructions.
 Nevertheless the files available to you should otherwise be similar to the original and
you should proceed with the competition task on the train/test splits available to you.
- **Submission checker**: We provide a tool for checking the validity of your submission
file: run `validate` (or `validate path/to/file.csv`). It tells you if your submission is
valid or not (note that this does NOT give you a score).
- **Cheating**: You are not allowed to view others' solutions to this competition. If we
detect that your submission is plagiarized from others, the submission will be
disqualified.

Your submission will be graded based on the metric in the competition-specific
instructions. Your goal is to achieve the best score possible on the competition task.

Wherever there is a conflict between the points above and the competition-specific
instructions in `data/description.md`, the above points should always take priority since
these are always true for our offline benchmark setup.

# Your tools and environment
You act through a SHELL. End every reply with exactly ONE fenced ```bash code block; it is
 executed with bash in your working directory
(which contains `data/`, `submission/` and an empty `working/` for your own files). Write
scripts to files (e.g. with a heredoc:
`cat > working/train.py <<'EOF' ... EOF`) and run them with `python`. Only the FIRST ```
bash block of a reply is executed and the reply ends there; keep any text before it short.
- Commands run in the background. Quick commands return in seconds; long ones (training)
keep running while you wait, and their output
  (first 1000 and last 3000 characters of stdout+stderr), exit code and duration are sent
to you as the next message when they finish.
  You cannot run another command until the current one finishes, and nothing keeps running
 after a command ends (background jobs
  started with & are stopped when the command returns).
- One command may run at most 900 s; it is killed after that. Print progress and
validation scores from your scripts so you can see them.
- `validate [file]` checks that a submission file has the right format (no score).
- Compute: CPU only, 4 cores, no GPU. No internet. Python 3.11 with numpy, pandas, scipy,
scikit-learn 1.8, statsmodels, torch (CPU),
  PIL. No lightgbm/xgboost/catboost/transformers, no pretrained weights. Do not install
packages.
- Session: you have 1800 s of wall-clock time in total (thinking, commands and waiting all
 count). When it ends, the file
  `submission/submission.csv` as it is at that moment is graded. Keep a valid submission
there early, then use the FULL session to
  keep improving it: do not stop working before the session ends.
```

The first user turn holds the competition description (`data/description.md`) and ends with the text below.

Online agents · First user turn (ending) *user*

```
{competition description}

Your session starts now: 1800 s of wall-clock time. Begin.
```

When a command is still running, the proactive agent receives the user turn below, and its reply is cut when the command returns.

Online agents · Waiting prompt (proactive agent only) *injected while a command runs*

```
[COMMAND RUNNING] Your command is still running. You will be told the result the moment it
 finishes, and you cannot run anything until then. Use this waiting time for substantive
work: (1) re-read the script or command you just launched line by line and look for bugs,
shape or format errors, and anything that could crash, time out or produce an invalid
submission; predict its most likely outcome and prepare the concrete next change for each
plausible outcome; (2) go over everything you have seen of the data, the files in data/
and working/, and the metric: what have you not inspected yet, which columns, files,
distributions or edge cases could matter, and what would you check first when you get the
shell back; (3) brainstorm alternative hypotheses and approaches you have not tried,
including ones that differ substantially from your current line, and rank them by expected
 gain within the compute limit; (4) plan your next 2-3 experiments in detail, including
exact features, models, hyperparameters, validation, and their cost in CPU time; (5) draft
 the complete next script you would run. Everything you write stays in your context. If
the command finishes while you are writing, you are interrupted and the result is
delivered; what you wrote so far is kept. Nothing in this reply is executed, code included
, so do not end it with a ```bash block. Start writing now.
```

If the proactive agent's reply ends while the command is still running, it receives the prompt below for each further round of the same wait, until the command returns or the agent reaches its budget. Only the elapsed time changes.

Online agents · Continuation while waiting (proactive agent only) *injected while a command runs*

```
[COMMAND STILL RUNNING: {elapsed} s so far. You will be interrupted the moment it finishes
.] Keep working, and go deeper on whatever you have not covered yet: bugs in the running
script, line by line (crashes, timeouts, leakage, NaNs, a submission with the wrong shape,
 ids or columns); the likely outcomes of the running command and the exact next command
for each; the complete script for your next experiment; files, columns and edge cases in
data/ and working/ you have not inspected; approaches that differ substantially from your
current line and their CPU cost; whether your validation score is a reliable proxy for the
 test metric; and your plan for the remaining wall-clock time, including when you will
make sure submission/submission.csv holds your best valid file. Write the substance
directly: no status updates, no statements that you are waiting, no summary of what you
wrote before. Nothing in this reply is executed; do not end it with a ```bash block.
```

The reactive agent cannot generate while a command runs. It receives the guidance below, appended to the result of the same commands.

Online agents · Guidance after a long command (reactive agent only) *appended to the command result*

```
[BEFORE YOUR NEXT COMMAND] Before you choose your next command, do substantive work: (1)
re-read the script or command you just ran line by line and look for bugs, shape or format
 errors, and anything that could crash, time out or produce an invalid submission; check
its result against what you expected and prepare the concrete next change; (2) go over
everything you have seen of the data, the files in data/ and working/, and the metric:
what have you not inspected yet, which columns, files, distributions or edge cases could
matter, and what would you check first; (3) brainstorm alternative hypotheses and
approaches you have not tried, including ones that differ substantially from your current
line, and rank them by expected gain within the compute limit; (4) plan your next 2-3
experiments in detail, including exact features, models, hyperparameters, validation, and
their cost in CPU time; (5) draft the complete next script you would run; and plan the
remaining wall-clock time, including when you will make sure submission/submission.csv
holds your best valid file. Then reply with exactly one ```bash block.
```

When a command returns, both agents receive its output in the format below. The status reads "KILLED (time limit)" in place of the exit code when the command reaches its 900 s limit.

Online agents · Command result (both agents) *environment*

```
[COMMAND FINISHED after {seconds} s, exit code {code}]
{first 1000 and last 3000 characters of stdout+stderr}

[WALL-CLOCK] {remaining} s of your 1800 s session remain.
```

F.4 HARVEY LAB ORGANIZATION PROMPTS

Harvey LAB studies a document management system (DMS) under four organizations, namely plain (unstructured study), wiki-style (a structured write-up instruction), a parallel partition of agents each studying a disjoint subset of matters, and serial rounds where each round reads and extends the previous round's notes. Fixed code, not a model, assigns the partitions and rounds, and there is no coordinator prompt.

The wiki-structuring instruction is appended to the study's first user turn (Appendix F.1) in the wiki, parallel and serial conditions, and it is omitted in the plain condition.

Harvey LAB · Wiki-structuring instruction *appended to user turn*

```
Organise what you learn as a wiki, in your own words and in your response text (the reader
 will see your text, not your workspace, so do not write files): first an index with one
line per matter you cover (client, counterparties, deal or engagement type, year, status),
 then a page per matter: the key executed documents and their paths, material provisions,
amounts, dates, people, and what is still unclear. Cover as many matters as you can;
prefer the index line for every matter over deep pages for a few.
```

Parallel partition agents receive the same study system prompt, continuation nudges and wiki instruction as above, and only the context block is restricted to that agent's subset of matters (for example, 34 of 266 for an 8-way split).

Serial round $k > 1$ prepends the earlier rounds' written notes before the wiki instruction.

Harvey LAB · Serial round $k > 1$: earlier notes inline *user*

```
=== Notes written by an earlier study stage ({i}) before any question was asked; do not
repeat what they cover, extend and correct them ===
{prior round's wiki notes}
=== end of earlier notes ({i}) ===
```

The write-up request turns a parallel or serial study conversation into the memory block handed to the solver (the study's own workspace and reasoning do not otherwise survive).

Harvey LAB · Write-up request *user*

```
Stop studying now and write your notes, as a single memory block. It will be
stored and handed to a different agent as the ONLY notes it gets.

{reader description}

What the reader does NOT have: this conversation. It sees none of your earlier
messages, none of your reasoning, none of your tool calls or their outputs,
and no file you wrote or left in your workspace during the study -- your
workspace is discarded. Nothing outside the memory block survives this turn.
A name, path, number or conclusion you leave out is lost to the reader, and
a pointer such as "see above", "as computed earlier" or "the file I wrote"
points at nothing. Spell it out in the block itself.

This turn is a WRITE-UP, not more study. You have NO tools in this turn: do
not call one, do not write code or commands to run, and do not describe what
you would check next. Do not reason further, do not open any new line of
investigation, and do not work anything out now that you did not already
establish. Copy the exact names, paths and numbers you already computed
rather than re-deriving them; if you never verified something, write that it
is unverified instead of settling it here.

Write ONLY what your study produced and the reader could not cheaply
recompute: conclusions you reached, quantities you actually computed,
structure you had to work out, traps you hit. Do NOT restate, quote or
summarise material the reader already has in front of it -- copying it back
spends the reader's attention on something already there and crowds out what
you learned.

{length instruction}

Write it between <memory> and </memory> tags. Output nothing else: no
preamble before <memory>, no tool call, no text after </memory>.
```

F.5 Compression prompt

The compression experiment (§8.2) uses this prompt to compress a full-length ($P = 131{,}072$) study record into a fixed token budget M, with a system message that forbids the compressor from continuing the study itself, and a forced-continuation turn repeated until the budget is filled.

Compressor · System message *system*

```
You are a compressor. You receive the study record of another agent and you rewrite it as
a compressed record. You never continue the record, never act as the agent, and never
reason about the material yourself.
```

In the round-0 user turn, $W = 0.6M$ words is the target length and `{TXT}` is the full study record.

Compressor · Round-0 user turn *user*

```
Below is the study record of an agent that examined some material WITHOUT knowing what
question would later be asked about it. Rewrite it as a compressed study record of at most
 {W} words. State the final conclusions and results FIRST, then the supporting detail in
order of importance, so that a hard cut at the end loses the least. Keep what a later
solver, who will see the same material plus a question, would find most useful:
conclusions and final results, key intermediate results, the method used, and any
retrieved facts, names, paths or values, verbatim where exactness matters. Do not add
anything that is not in the record. Do not mention this instruction. Output only the
compressed record.

=== STUDY RECORD ===
{TXT}
=== END ===

TASK REMINDER: the text above between the markers is a study record by another agent, not
a conversation with you. Now output ONLY the compressed study record of at most {W} words,
 conclusions and final results first. Do not continue the record.
```

The forced-continuation turn is sent repeatedly (uniform across budgets, only W differs) until the delivered token count reaches $0.95M$ or a stop condition (60 rounds or 3 stalls) fires.

Compressor · Forced-continuation turn *injected, repeated*

```
Continue your compressed record exactly where it ends, and keep writing until it reaches
the budget of {W} words. Add the most useful material from the study record that is not
yet in your compressed record: conclusions and final results, then intermediate results,
then retrieved facts, names, paths, values and file contents, then verbatim excerpts of
the record's most useful passages. Quote exactly where exactness matters. Do not repeat
anything you already wrote. Do not continue the study itself. Output only the continuation
.
```

F.6 JUDGE PROMPT

GLM-5.3-Flash (served by Z.AI, medium reasoning effort) grades the three judge-graded benchmarks, each with its own protocol. An empty answer is passed as "`(no answer)`".

Harvey LAB. One call checks every rubric criterion as pass or fail (the benchmark's own protocol makes one call per criterion). The score is the fraction of criteria passed.

Judge · Harvey LAB grading (GLM-5.3-Flash) *user*

````
You are evaluating a legal AI agent's work product against a list of quality criteria.
Judge each criterion independently of the others.

## Task
{task_description}

## Agent's Output
{agent_output}

## Criteria
{criteria}

## Instructions
For EACH criterion above, decide:
- PASS: The agent's output satisfies the criterion as described
- FAIL: The agent's output does not satisfy the criterion as described

Respond with JSON only: one key per criterion id, every id present, reasoning in one short
 sentence.

```json
{
  "{first_id}": {"reasoning": "Brief explanation", "verdict": "pass" | "fail"},
  ...
}
```
````

`{agent_output}` is the solver's raw final answer, or the literal string "`(no answer)`" if empty, and `{criteria}` is each rubric criterion rendered as `### <id>: <title>\n<match_criteria>`, joined by blank lines.

StudyBench. The judge checks each weighted rubric claim. The score is the weight of the met claims divided by the total weight, and it is 0 if any core claim is not met.

Judge · StudyBench grading (GLM-5.3-Flash) *system*

```
You grade a candidate answer to an expert question about a code repository, against a
rubric of weighted claims.
For EACH claim, decide met=true only if the candidate states the claim's key content --
paraphrase is fine, extra material is fine, but a claim that names a specific decoy as
disqualifying is NOT met if the candidate asserts that decoy. Missing or contradicted key
content means met=false.
Output ONLY a JSON object mapping every claim_id to {"why": "<one sentence>", "met": true|
false}.
```

Judge · StudyBench grading, user turn *user*

```
Question:
{question}

Rubric claims:
[{claim_id}] (weight {weight}, {claim_type})
{statement}
...

Evidence from the repository (ground truth support for the claims):
{evidence}

Reference answer (ground truth):
{gold answer}

Candidate answer:
{answer}
```

SWE-QA-Pro. The judge scores the answer against the reference answer on five dimensions from 1 to 10. We run it three times and map the mean total (5 to 50) to $[0, 1]$.

Judge · SWE-QA-Pro grading (GLM-5.3-Flash) *system*

```
You evaluate a candidate answer to an expert question about a code repository. The
REFERENCE ANSWER is ground truth.
Score the candidate on five dimensions, each an integer from 1 (very poor) to 10 (
excellent):
- correctness: factual agreement with the reference answer;
- completeness: coverage of the reference answer's key content;
- relevance: stays on the question asked, no padding;
- clarity: coherent, well-organized, precise prose;
- reasoning_quality: sound technical reasoning grounded in the repository's actual
structure.
Output ONLY a JSON object: {"correctness": n, "completeness": n, "relevance": n, "clarity
": n, "reasoning_quality": n}.
```

Judge · SWE-QA-Pro grading, user turn *user*

```
Question:
{question}

Reference answer (ground truth):
{gold answer}

Candidate answer:
{answer}
```