Machine Studying: A System-Level Reframing of Continual Adaptation from Declarative Corpora

Jacob Xiaochen Li [1] **Rick Battle** [2] **Omar Khattab** [1]

Abstract

We increasingly want AI agents to work in novel domains, which are often described only through a corpus of documents. Current agents mostly rely on late inference-time compute, relatively shallow indexing and retrieval, or on hand-engineered RL environments to understand new domains. Humans can turn reading a textbook and actively thinking about the material into expertise, so why can't agents yet? We call this problem Machine Studying and define "expertise" as the weighted area under an agent's performance curve as inference compute grows. We instantiate the framework in STUDYBENCH, a new open-book, hidden-task benchmark built on three corpora, and evaluate simple study procedures that act on weights and context. We offer this as a new frame through which continual adaptation should be studied.

1. Introduction

Beyond the canonical recipe of pre-training and post-training, recent agentic AI systems leverage inference-time compute (Wei et al., 2025), agentic scaffolds (Yao et al., 2023), and tool ecosystems (Anthropic, 2024). These methods enable agentic systems to reason, search, and complete long-horizon tasks across diverse domains, often achieving high performance (Shen et al., 2026; Gonzalez-Pumariega et al., 2026). Owing to their advanced capabilities, such agents are often deployed in domains not encountered during training, with the expectation that they will operate effectively. For example, agents may be tasked with utilizing a new programming library or engaging with emerging literature on a novel disease. A new domain like these most naturally appears as a declarative corpus of documents, such as textbooks on a technical subject or manuals describing new tools.

[1]MIT CSAIL [2]Broadcom. Correspondence to: Jacob Xiaochen Li <jacobli@mit.edu>.

Presented at the ICML 2026 Workshop "Continual Adaptation at Scale: Towards Sustainable AI". Copyright 2026 by the author(s).

When working with these corpora, current agents primarily depend on inference-time compute. Most approaches either reduce it to retrieval-augmented generation (RAG) problems (Lewis et al., 2020), relying on real-time agentic search, or to in-context learning (Brown et al., 2020), assuming that a large context window will be sufficient for an agent to learn new materials in context. For important domains, the prevailing best approach is to construct a reinforcement learning environment so agents can develop relevant skills (Shao et al., 2025). Across all of these, our agents today engage with new domains in shallow and hand-engineered ways. However, humans can turn reading a textbook or actively thinking about the material into deep knowledge and even expertise (Chi et al., 1994; Dunlosky et al., 2013; Fiorella, 2023). Why can't our AI agents yet?

We refer to this problem as *Machine Studying*. Notably, our setup shifts the role of benchmark scores, since the corpus remains available at test time; a sufficiently aggressive agent could, in principle, buy accuracy by spending arbitrarily more compute, say by rereading every file before each answer, thereby achieving high scores. As a result, evaluation should instead reflect the tradeoff among study-time compute, test-time compute, and downstream performance. We thus define an expert as an agent that can *efficiently* convert inference compute into accurate results, and we measure expertise as the weighted area under the agent's performance curve as inference compute grows. We operationalize this setup in STUDYBENCH, a benchmark across three corpora, and use it to compare a number of studying procedures that act on weights and context under multiple inference budgets.

We make four contributions. First, we formulate machine studying as an open-book, hidden-task adaptation problem for agentic AI systems. Second, we define expertise as the weighted area under an agent's compute-performance curve. Third, we introduce and open-source STUDYBENCH.[1] Fourth, we present an initial study across representative study procedures, finding that these procedures struggle to improve agents' expertise.

[1]https://huggingface.co/datasets/jacobli/studybench

2. Motivation and Related Work

Continual learning is canonically framed as a closed-book problem and approached through weight updates. This assumption makes catastrophic interference (widely known as catastrophic forgetting) the canonical failure mode (McCloskey & Cohen, 1989; Kirkpatrick et al., 2017).

In-context learning shows that the input alone can induce new behavior (Brown et al., 2020). The immediate response is thus to enlarge the context window; some work scales it to millions of tokens (DeepSeek-AI, 2026), and other architectures pursue effectively unbounded context (Tandon et al., 2025). A second response seeks to capture the state the model attains under a given context. Context distillation achieves this by training the model to reproduce the behavior elicited when the relevant context is present, so that the context is not needed at test time (Snell et al., 2022). Later work follows this logic. Some methods train hypernetworks that map a corpus to a LoRA adapter, reducing the training cost of context distillation (Charakorn et al., 2026; Liu et al., 2026). Others encode corpus information into trainable KV caches (Eyuboglu et al., 2025; Zweiger et al., 2026). These methods serve the same purpose and can be considered forms of context distillation.

Most such existing work assumes that the ceiling is the state of the model with the corpus in its context window, and that the job is to approximate that state more efficiently. In practice, agents retain access to the corpus at test time, so this goal is arguably imprecise. Machine studying thus asks how study-time compute turns that test-time access into efficient use.

3. Machine studying

We define an agentic system as a tuple $\Sigma = (\mathbf{M}, \mathbf{C}, \mathbf{H}, \mathbf{A}, \mathbf{N}, \mathbf{T})$, where $\mathbf{M}$ is the underlying model, $\mathbf{C}$ is the available context, $\mathbf{H}$ is the harness (such as ReAct, RLM (Zhang et al., 2025a), and inference policies (Alomrani et al., 2025; OpenAI, 2026)), $\mathbf{A}$ refers to non-neural assets like databases, $\mathbf{N}$ includes neural auxiliaries (such as modular adapters (Pfeiffer et al., 2023) and prefix-tuning states (Li & Liang, 2021)), and $\mathbf{T}$ is the tool set.

Study procedures. Let $\mathbf{D}$ be a declarative corpus that exceeds the system's high-quality context window and contains no labels, rewards, or explicit task distribution. A study procedure is any modification that the agentic system applies to any component of itself before evaluation, using $\mathbf{D}$ without prior knowledge of the test. For a procedure π, the post-study system is denoted as $\Sigma^{\pi}_{\mathbf{D}} = \pi(\Sigma, \mathbf{D})$.

Open-book hidden-task evaluation. Expertise is measured in a hidden downstream environment induced by $\mathbf{D}$. Importantly, $\mathbf{D}$ remains accessible during inference.

Expertise. Concretely, we measure expertise as the weighted area under the agent's performance curve as inference compute grows,

$$\mathcal{E}(\Sigma; \mathbf{D}) \;=\; \int p_{\Sigma,\mathbf{D}}(x)\, w(x)\, dx,$$

where $p_{\Sigma,\mathbf{D}}(x)$ is performance at an inference budget (in log scale) x and w says which budgets matter. In this paper, we anchor $x = 0$ at the 3k tokens an agent needs to produce one complete answer and halve w with every doubling of inference compute. A studying algorithm π maps an agent and a corpus to a new agent, and it works if $\mathcal{E}(\pi(\Sigma, \mathbf{D}); \mathbf{D}) > \mathcal{E}(\Sigma; \mathbf{D})$. Defining expertise as a scalar lets us define studying intelligence as the efficiency with which an agent acquires expertise across new domains, aligning with Chollet (2019)'s view of capability intelligence as skill-acquisition efficiency, which we leave to future work to measure. Appendix C describes the calculation used to estimate expertise and intelligence.

4. StudyBench

StudyBench comprises three tasks. Study-DSPy consists of 30 coding questions on DSPy (Khattab et al., 2023), a framework for programming language models. DSPy has existed since December 2022, so many recent models are aware of DSPy, but their knowledge is often incomplete or outdated. Study-OpenClaw consists of 20 coding questions on OpenClaw,[2] an open-source framework for self-hosted personal AI assistants released at the end of 2025, which places it beyond the training cutoff for most models in our experiments. For both, agents may access the codebase using grep and glob, and may read specific files. Study-Literature draws from approximately 50k full-text machine learning papers published between 2018 and 2025 from ICLR, CVPR, ICML, and NeurIPS. The agent queries the corpus using BM25 for up to 20 turns, retrieving 20 unique papers per query, then selects 100 papers and writes a review. We evaluate the selection against a must-cite set constructed using the MasterSet labeling procedure (Ratul et al., 2026).

Each coding exam is produced semi-automatically, with significant information asymmetry and expert human oversight. The exam writer is GPT-5.4 in Codex (OpenAI, 2025) at `xhigh` effort. It receives privileged materials including full documentation, a pool of real user questions, deterministic checkers, and a critique loop. Real user questions are sourced from the DSPy community and OpenClaw GitHub closed issues, which further ensures the diversity of the questions. In contrast, test takers receive only the codebase unless otherwise stated. Responses are evaluated using

[2] https://openclaw.ai/

Table 1. Accuracy (%, lenient grading) and mean generated tokens per question (thousands, rounded) across four inference budgets, with the resulting *expertise*. Only the self-written cheatsheet surpasses the raw model by a meaningful margin.

	Direct		k=5		k=20		Forced k=20		Expertise
System	Acc	Tok	Acc	Tok	Acc	Tok	Acc	Tok	(WAUC)
Study-DSPy									
Qwen3.5-9B (base)	3.3	4.1	8.6	7.9	9.6	8.6	29.4	34.6	6.49
+ cheatsheet	6.3	3.9	14.4	6.1	14.1	7.1	23.1	29.9	**9.65**
SFT + OPSD	9.4	9.2	7.4	15.2	8.5	16.6	21.0	155.8	3.29
CPT(code)	5.1	5.9	7.4	8.8	7.0	10.6	14.3	59.3	3.71
CPT(doc)	3.8	4.8	7.2	8.2	6.2	9.9	14.6	74.9	3.92
Study-OpenClaw									
Qwen3.5-9B (base)	2.3	4.1	6.9	4.6	15.8	9.7	17.6	24.3	7.64
+ cheatsheet	4.3	3.8	8.6	6.0	15.2	9.1	18.1	20.1	**8.18**
CPT(code)	2.2	3.4	8.8	4.7	15.3	11.6	14.1	32.1	7.82

weighted rubrics and deterministic checks for compilation. Failing the compilation check in standard grading results in an automatic zero, while lenient grading omits the compilation check. Appendix A gives the full pipeline, prompts, and examples.

5. Methods

Each agent is evaluated as a model within a ReAct harness. We vary the inference budget across four settings: the model may answer directly without tools, run up to 5 or 20 tool iterations and stop voluntarily, or perform exactly 20 iterations with no early stopping. For each budget, we sample three rollouts and average to obtain a point on the agent's quality-cost curve. Expertise is computed from these four points. We study three families of methods.

Continual pre-training (CPT) updates model weights by training LoRA adapters (Hu et al., 2022) through next-token prediction on the raw corpus. We test two variants: CPT(CODE) on the DSPy codebase and CPT(DOC) on its documentation. Documentation remains accessible at inference time for CPT(DOC). Since training a post-trained model on raw text can degrade abilities such as instruction following (Ibrahim et al., 2024), we also include self-sampled coding traces from MBPP (Austin et al., 2021) as an anchor. For OpenClaw, we use only the CPT(CODE) variant.

Supervised fine-tuning (SFT) trains the model on question-answer pairs using cross-entropy loss. We use a larger model, DeepSeek-V4-Flash (DeepSeek-AI, 2026), to generate high-quality synthetic questions, then sample answers from the model being studied by providing the gold context. Models are trained with reasoning disabled. A recovery stage of on-policy self-distillation (OPSD) over approximately 60k examples from Tulu3 (Lambert et al., 2024), OpenThoughts (Guha et al., 2025), and MBPP, following the Thinking Machines Lab recipe (Thinking Machines Lab, 2025), restores general behavior.

The third method, CHEATSHEET, has the agent explore the repository using the same three tools for 50 steps and write a reference document that is prepended to every future question. While people often handwrite such reference material as skill files, this would not qualify as studying because it outsources the process. Here, the agent writes its own guide. There is concurrent work exploring this direction of weight-free studying (Vogel et al., 2026). More details can be found in Section B.

6. Empirical results

We use four models across experiments: GPT-5.1 (knowledge cutoff September 30, 2024), GPT-5.4 mini (August 31, 2025), GPT-5.5 (December 1, 2025), and Qwen3.5-9B, released 2026 with no reported knowledge cutoff.

Models with comparable capabilities may demonstrate different levels of expertise. We analyze the brute-force studying approach used by frontier labs to illustrate the importance of expertise. When a new library becomes relevant, labs regenerate pre- and post-training data using an updated knowledge cutoff and retrain the next model to incorporate the new material. Retraining serves as a studying algorithm under our definition, and it is practically reliable but also the most expensive one.

We compare the performance of GPT-5.1 and GPT-5.4 mini on two coding benchmarks. Although GPT-5.4 mini is smaller, both models perform similarly on agentic benchmarks, with GPT-5.1 slightly outperforming GPT-5.4 mini in some public evaluations[3] of τ^2-telecom (Barres et al., 2025). As shown in Figure 1, GPT-5.4 mini consistently outperforms on DSPy, which gained popularity after 2024. However, on OpenClaw, released after both cutoffs, this

[3] https://llm-stats.com/

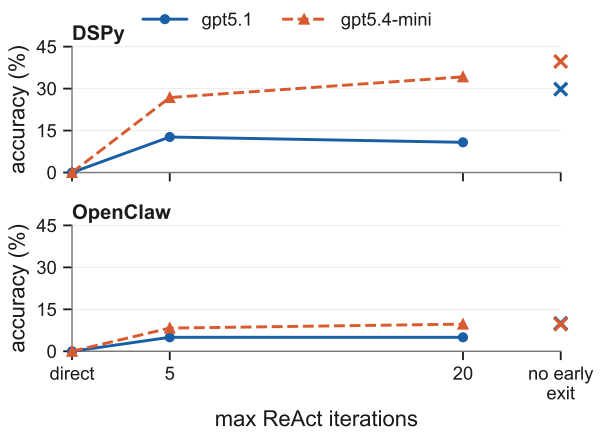


Figure 1. Results of two models close in agentic capability but a training cutoff apart. On DSPy the newer GPT-5.4 mini wins at every budget (except direct setting); on OpenClaw the advantage disappears.



Figure 2. STUDY-LITERATURE reach and recall@100 for two models a training cutoff apart. Reach is near 60% for both, so search delivers the right papers regardless of cutoff, while recall@100 separates the models by up to 20 points.

advantage disappears, and both models perform poorly regardless of longer search duration.

Memorization is no substitute for expertise. The Qwen3.5-9B model (Qwen Team, 2026) was used for the studying experiments. Table 1 presents its performance across both tasks. Results are based on lenient grading, since most answers would have otherwise received zero. The effectiveness of the model's study approach is subsequently analyzed. Table 1 includes performance, inference compute as generated tokens, and expertise scores. Under the budget of our runs, both CPT variants led to lower expertise than the untrained model. next-token prediction over the corpus did not improve expertise, and supervised fine-tuning with on-policy self-distillation increased performance in closed-book settings but did not compound with agentic search. Training also led to increased verbosity, with the shortest gradable answer now requiring about 9k tokens, and this higher cost per answer results in lower overall expertise. The cheatsheet is the only procedure that achieved a higher expertise score than the unmodified agent on STUDY-DSPY, with gains primarily at low inference budget, and with only marginal gains on STUDY-OPENCLAW. At the forced 20-iteration budget, the unmodified agent exceeds the cheatsheet, as more search amortizes away the cheatsheet's advantage.

Retrieval is no substitute for expertise. STUDY-LITERATURE is structured to distinguish the process of finding relevant papers from recognizing their significance and incorporating them into the review. The agent executes twenty BM25 queries per session, encountering on average 230 unique papers. The proportion of gold references in the must-cite set among these is measured as the agent's *reach*. At the end of each session, the agent selects 100 papers; *recall@100* indicates how many gold references remain in this final selection. Reach reflects the effectiveness of the search process, while recall@100 measures the model's ability to identify relevant papers. GPT-5.1 and GPT-5.5 are used for this evaluation.

As shown in Figure 2, both models reach approximately the same proportion of the gold references across years. However, GPT-5.5 retains up to 20% more gold references in its final selection, primarily on papers published recently. This finding indicates that while GPT-5.1 may find relevant 2025 papers through searches, it lacks the contextual knowledge to recognize their significance and retain them. Thus, although a stronger retriever could increase reach, it would not by itself be enough to close the gap unless the older agent develops contextual understanding from the corpus, similar to the expertise the newer agent acquired through additional training.

7. Conclusion

The gap between a frozen agent and a new domain is closed by hand. Machine studying seeks agents that can efficiently study a corpus to make themselves experts in novel domains. Our preliminary results suggest that current agents largely lack this ability. We are sharing STUDYBENCH, the benchmark we are actively developing, so that others can begin thinking about this with us.

Acknowledgements. This work is partially supported by the VMware University Research Fund (VMURF).

References

Alomrani, M. A., Zhang, Y., Li, D., Sun, Q., Pal, S., Zhang, Z., Hu, Y., Ajwani, R. D., Valkanas, A., Karimi, R., et al. Reasoning on a budget: A survey of adaptive and controllable test-time compute in llms. *arXiv preprint arXiv:2507.02076*, 2025.

Anthropic. Introducing the model context protocol. https://www.anthropic.com/news/model-context-protocol, November 2024. Accessed: 2026-04-23.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Barres, V., Dong, H., Ray, S., Si, X., and Narasimhan, K. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Campello, R. J., Moulavi, D., and Sander, J. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*, pp. 160–172. Springer, 2013.

Charakorn, R., Cetin, E., Uesaka, S., and Lange, R. T. Doc-to-lora: Learning to instantly internalize contexts. *arXiv preprint arXiv:2602.15902*, 2026.

Chi, M. T., De Leeuw, N., Chiu, M.-H., and LaVancher, C. Eliciting self-explanations improves understanding. *Cognitive science*, 18(3):439–477, 1994.

Chollet, F. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Dunlosky, J., Rawson, K. A., Marsh, E. J., Nathan, M. J., and Willingham, D. T. Improving students' learning with effective learning techniques: Promising directions from cognitive and educational psychology. *Psychological Science in the Public interest*, 14(1):4–58, 2013.

Eyuboglu, S., Ehrlich, R., Arora, S., Guha, N., Zinsley, D., Liu, E., Tennien, W., Rudra, A., Zou, J., Mirhoseini, A., et al. Cartridges: Lightweight and general-purpose long context representations via self-study. *arXiv preprint arXiv:2506.06266*, 2025.

Fiorella, L. Making sense of generative learning. *Educational Psychology Review*, 35(2):50, 2023.

Gonzalez-Pumariega, G., Tu, V., Lee, C.-L., Yang, J., Li, A., and Wang, X. E. Scaling agents for computer use, 2026. URL https://arxiv.org/abs/2510.02250.

Guha, E., Marten, R., Keh, S., Raoof, N., Smyrnis, G., Bansal, H., Nezhurina, M., Mercat, J., Vu, T., Sprague, Z., et al. Openthoughts: Data recipes for reasoning models. *arXiv preprint arXiv:2506.04178*, 2025.

Hu, E. J., yelong shen, Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Ibrahim, A., Thérien, B., Gupta, K., Richter, M. L., Anthony, Q., Lesort, T., Belilovsky, E., and Rish, I. Simple and scalable strategies to continually pre-train large language models. *arXiv preprint arXiv:2403.08763*, 2024.

Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

Lambert, N., Morrison, J., Pyatkin, V., Huang, S., Ivison, H., Brahman, F., Miranda, L. J. V., Liu, A., Dziri, N., Lyu, S., et al. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

Li, X. L. and Liang, P. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 4582–4597, 2021.

Liu, Y., Wang, X., Mao, Y., Gelberg, Y., Maron, H., and Zhang, M. Shine: A scalable in-context hypernetwork for mapping context to lora in a single pass. *arXiv preprint arXiv:2602.06358*, 2026.

McCloskey, M. and Cohen, N. J. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pp. 109–165. Elsevier, 1989.

McInnes, L., Healy, J., and Melville, J. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

OpenAI. Introducing Codex. https://openai.com/index/introducing-codex/, May 2025. Accessed: 2026-05-05.

OpenAI. Reasoning models. https://developers.openai.com/api/docs/guides/reasoning, 2026. Accessed: 2026-04-23.

Pfeiffer, J., Ruder, S., Vulić, I., and Ponti, E. M. Modular deep learning. *arXiv preprint arXiv:2302.11529*, 2023.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Ratul, M. T. R., Chen, Z., Fu, K., Ji, T., and Zhang, L. Masterset: A large-scale benchmark for must-cite citation recommendation in the ai/ml literature. *arXiv preprint arXiv:2604.17680*, 2026.

Shao, R., Asai, A., Shen, S. Z., Ivison, H., Kishore, V., Zhuo, J., Zhao, X., Park, M., Finlayson, S. G., Sontag, D., et al. Dr tulu: Reinforcement learning with evolving rubrics for deep research. *arXiv preprint arXiv:2511.19399*, 2025.

Shen, E., Tormoen, D., Shah, S., Farhadi, A., and Dettmers, T. Sera: Soft-verified efficient repository agents. *arXiv preprint arXiv:2601.20789*, 2026.

Snell, C., Klein, D., and Zhong, R. Learning by distilling context. *arXiv preprint arXiv:2209.15189*, 2022.

Tandon, A., Dalal, K., Li, X., Koceja, D., Rød, M., Buchanan, S., Wang, X., Leskovec, J., Koyejo, S., Hashimoto, T., et al. End-to-end test-time training for long context. *arXiv preprint arXiv:2512.23675*, 2025.

Thinking Machines Lab. Tinker, 2025. URL https://thinkingmachines.ai/tinker/.

Vogel, M., Meyer-Eschenbach, F., Kohler, S., Grünewald, E., and Balzer, F. Codebase-memory: Tree-sitter-based knowledge graphs for llm code exploration via mcp. *arXiv preprint arXiv:2603.27277*, 2026.

Wei, J., Sun, Z., Papay, S., McKinney, S., Han, J., Fulford, I., Chung, H. W., Passos, A. T., Fedus, W., and Glaese, A. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*, 2025.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Zhang, A. L., Kraska, T., and Khattab, O. Recursive language models. *arXiv preprint arXiv:2512.24601*, 2025a.

Zhang, Y., Li, M., Long, D., Zhang, X., Lin, H., Yang, B., Xie, P., Yang, A., Liu, D., Lin, J., Huang, F., and Zhou, J. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025b.

Zweiger, A., Fu, X., Guo, H., and Kim, Y. Fast kv compaction via attention matching, 2026. URL https://arxiv.org/abs/2602.16284.

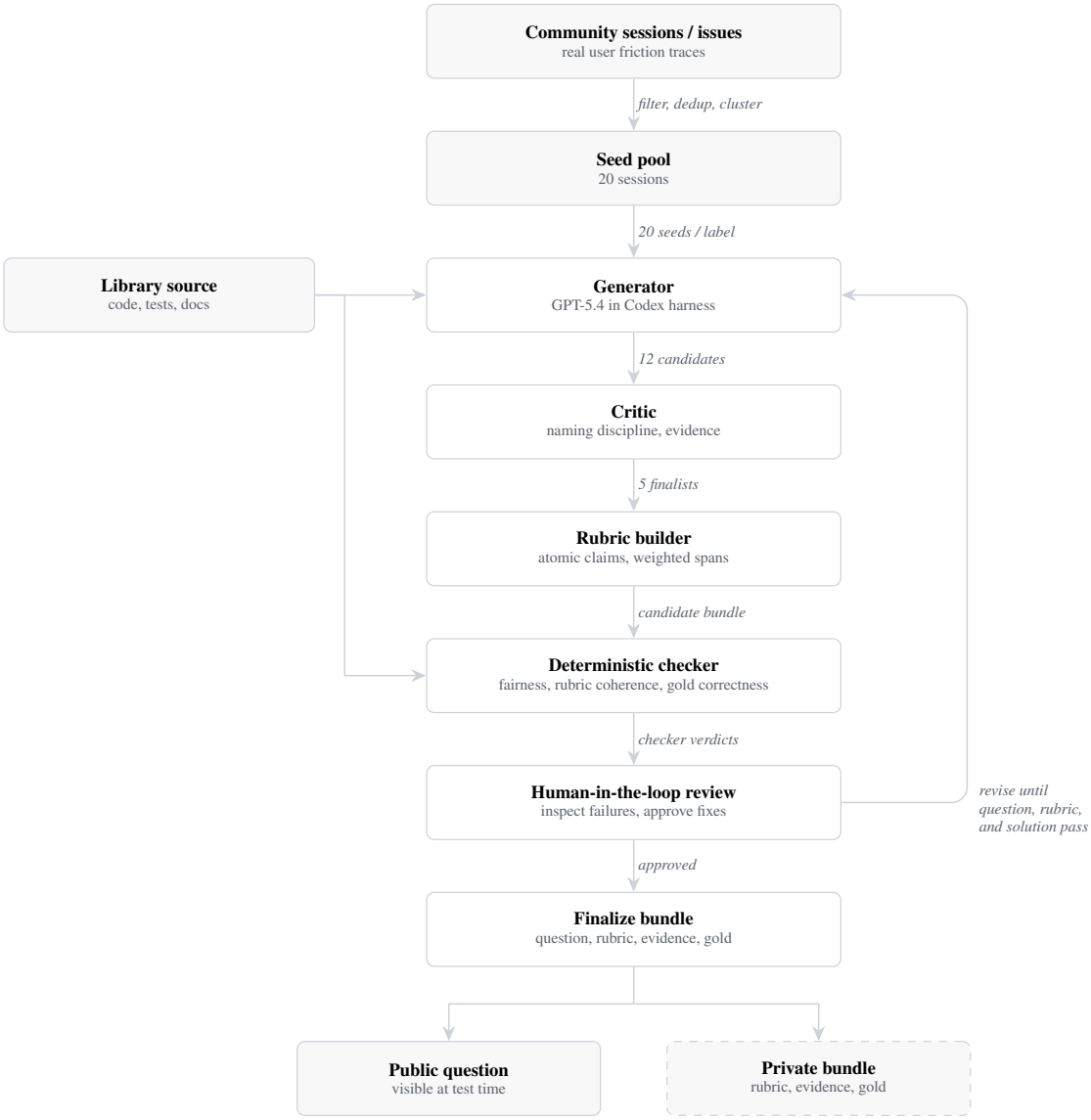


Figure 3. The STUDYBENCH coding-suite construction pipeline. Questions (DSPy) or GitHub issues (OpenClaw) are filtered and clustered into a per-label seed pool that anchors the question distribution. GPT-5.4 in a Codex-style harness, with read access to the full repository, lifts each cluster's seeds into candidate items through a generator and critic pair, and documentation is available at generation time but not at test time. A rubric builder converts each finalist into an atomic-claim grading bundle. A deterministic checker verifies rubric coherence, question fairness, and gold-answer correctness, while a senior library user reviews failures and sends items back for revision until the question, rubric, and solution jointly pass. The finalized bundle is then split: the public question is exposed to the answering system, while the rubric, evidence spans, and gold answer stay private.

A. StudyBench coding tasks: construction and validation

This appendix details the construction pipeline for STUDYBENCH coding tasks, including the prompt templates, library-specific values, and examples from the human-verified benchmark. The same pipeline is used for both STUDY-DSPY and STUDY-OPENCLAW.

A.1. Construction pipeline

Figure 3 presents the end-to-end process, and the following stages describe each step. Table 2 lists the codebase snapshots used in the benchmark.

Codebase	Repository	Commit	License
DSPy	`stanfordnlp/dspy`	`9cdb0aac28b2a04b064e40697ccd301872cf6a43`	MIT
OpenClaw	`openclaw/openclaw`	`da228660306b55a9cce3b973946f3aacfc515848`	MIT

Table 2. Codebase snapshots used for STUDYBENCH.

Codebase	Selected behavioral labels

DSPy	`gepa_optimizer_usage`
	`prompt_optimization_workflows`
	`rag_and_retrieval_pipelines`
	`react_agents_and_tools`
	`signature_schema_and_pydantic_types`
	`evaluation_metrics_and_custom_eval`

OpenClaw	`model_fallback_and_failover_logic`
	`cross_session_channel_context_and_session_behavior_requests`
	`memory_core_dreaming_and_promotion_pipeline`
	`new_plugin_provider_and_channel_integration_requests`

Table 3. Behavioral clusters selected for task generation.

Stage 1: source distribution. The process begins with a snapshot of real user-question sessions for each library. Conversations are filtered by length, language (English only), and question form: the first substantive turn must begin with an interrogative or imperative, such as *how*, *what*, *why*, *can*, *does*, *explain*, *show*, or *help*. We then deduplicate questions by text, and near-deduplicate using MinHash with $num_perm = 128$ and a Jaccard threshold of 0.7 over question shingles.

Stage 2: clustering. Each session is represented by its first substantive user question, embedded using Qwen3-Embedding-8B (Zhang et al., 2025b) with a domain-aware prefix prompt. Embeddings are projected to ten dimensions with UMAP (McInnes et al., 2018) ($n_neighbors = 15$) and clustered using HDBSCAN (Campello et al., 2013) (`min_cluster_size`=30, `min_samples`=5). GPT-5.4 assigns a behavioral label to each cluster after reviewing 30 representative sessions. Six clusters are selected for DSPy and four for OpenClaw, as shown in Table 3.

Stage 3a: candidate generation. For each label, GPT-5.4 at reasoning effort `xhigh`, running within Codex (OpenAI, 2025) with access to the full repository and documentation, generates 12 candidate (question, gold answer, `code_evidence`) triples. The generator is conditioned on 20 sampled seed sessions, the label description, and the library description. The complete template is provided in §A.2.

Stage 3b: critic selection. In a second pass, the same model and harness act as a critic. It reviews the 12 candidates and the seed sessions, then selects five finalists per label.

Stage 4: private rubric construction. For each selected item, GPT-5.4 converts the gold answer into atomic grading claims. Each claim is classified as `core` or `supporting` and cited to specific line spans in the evidence files. Numbered file dumps are provided to the rubric builder. Claim weights total 100, with most weight assigned to `core` claims.

Stage 5: bundle optimization. A Codex agent uses a deterministic syntax checker and a sandbox to run and verify answers. The questions, answers, and rubrics are optimized together. Human experts confirm that the questions and rubrics are fair and that the answers are correct. Answers and rubrics are further refined using the syntax checker and sandbox as debugging tools, iterating until the reference answers achieve full scores under the private rubric.

A.2. Generator prompt

```
You are generating benchmark-grade {library_name} expert QA inside the official {library_name} repository.

## About {library_name}
{library_description}

## Mission
- Target primary label: `{label}`
- Label description: `{label_description}`
- Produce exactly `{num_candidates}` candidate QA pairs.
- The final benchmark will evaluate an answering agent that has access to the {library_name}.
- You may use both code and docs right now to understand {library_name} deeply, but every final answer must be
↪ recoverable from the code roots listed below alone (no docs at answer time).

## Available context
- {library_name} code roots the answering agent will also see:
{code_roots_bullets}
- Documentation under `{repo_docs_subpath}/` for privileged generation-time orientation only (the answering
↪ agent might or might not have these)
- Sampled community QA sessions below, which represent realistic user-question distribution for the target label

## How to use the sampled community QA
- The sampled community sessions are the **distribution we want to match**, not just a tone reference. Real
↪ {library_name} users hit real friction --- that is the question gold mine. Anchor each generated question in
↪ what a user in the seeds was actually trying to do or observing.
- Real community questions are often vague, mis-framed, or mixed with multiple issues. Your job is to **sharpen,
↪ not imitate**: keep the user framing ("I'm trying to X and I see Y"), drop the noise, and commit to one crisp
↪ locator-hard question.
- **Do not trust the community answer as ground truth.** The answers in the seed sessions were written by a
↪ weaker assistant and are frequently wrong, incomplete, or out of date. Treat them only as hints about what
↪ the user was confused about. Your gold `answer` must be re-derived from the actual {library_name} source and
↪ docs --- read the code, verify the behavior, and cite `code_evidence` by file and symbol. If the community
↪ answer contradicts the code, the code wins.
- Do **not** copy or lightly paraphrase a sampled question; upscale by sharpening the behavioral framing.

## Naming discipline (critical --- the locator is the challenge)
The answering agent has grep/glob/read over the full {library_name} repo and tests. If the question text names
↪ the specific class, method, file, or internal helper that **is** the answer, you've given away the page
↪ number and turned this into a trivia question. The whole point of the benchmark is that *locating* the right
↪ code is half the work.

**OK to name (brand-level, user-facing concepts that a real user would type):**
{ok_to_name_bullets}
- Anything the community user in the seed session typed first, at the same granularity they typed it.

**Not OK to name (these attach the question to a specific implementation and leak the locator):**
- The method or attribute on a branded class that contains the answer --- name the behavior, not the method.
- Internal adapter / handler / parser / helper classes.
- Internal helper functions, file paths, test-file names, private config flags, snake_case function names with
↪ dot-paths.
- **Do not refer to "the X example / tutorial / notebook / walkthrough / demo / README / guide".** These
↪ phrasings are awkward and underspecified --- they point at an artifact as if a shared referent exists ("in
↪ the repo's multihop RAG example, ..."). A strong question stands on its own: describe the *scenario* or
↪ *setup* itself ("in a multi-hop retrieval pipeline where the model refines its query across hops, ..."), not
↪ the artifact that demonstrates it.
- Examples for this codebase:
{not_ok_examples_bullets}

Rule of thumb: if a reader can `grep -R "<token>"` and land within a few files of the answer, the token belongs
↪ in the gold `answer` and `code_evidence`, not in the question.

**Bad (names the attachment point):**
{bad_examples_block}

**Good (forces the agent to locate):**
{good_examples_block}

Walk this line carefully: a good question is **specific enough** that a careful reader of the repo converges on
↪ one well-defined answer, and **general enough** in wording that no symbol name gives the answer away. If the
↪ question could match a dozen unrelated places in the repo, it's too generic; tighten by adding behavioral
↪ constraints, not by naming the class.

## Required quality bar
- Questions should read like a thoughtful senior user describing **what they observed or what they want to
↪ accomplish**, not like an exam asking about a specific symbol.
- Questions must be difficult enough that they require synthesis across files, abstractions, behavior, tests,
↪ edge cases, or design tradeoffs.
- Prefer questions whose answers require reading implementation and tests together.
- Gold answers should be concise but precise, and must be supported by `code_evidence` pointing into actual
↪ {library_name} source (not just paraphrased from the seed's community answer).
```

```
- `code_evidence` must cite only real files under the code roots ({code_roots_inline}), and each cited filename
↪ must match the pattern `{file_glob}` (files with other extensions are out of scope and will be rejected).
- Provide at least two evidence items per candidate.
- Difficulty must be either `hard` or `very_hard`.

## Hard bans
- No documentation-only questions.
- No questions about exact wording from docs, tutorials, README, notebooks, or guides.
- No "in the X example / tutorial / notebook / walkthrough / demo / README / guide" phrasings. Describe the
↪ scenario itself.
- No trivial "does {library_name} have X?" or single-symbol existence questions.
- No one-grep questions with an obvious single-line answer.
- No ambiguous or underspecified questions.
- No questions whose answers depend on privileged docs rather than code/tests.
- No questions that violate the naming discipline above (no internal class/method/helper names, no file paths,
↪ no `Class.method` attachment points).
- No questions whose gold answer rests on "the seed said so" rather than on verified code behavior.

## Sampling anchors
The JSON block below contains the sampled community sessions for the target label. Use it to preserve the
↪ real-world distribution while raising the quality bar sharply.

{sampled_sources_json}

Return JSON that matches the provided schema and nothing else.
```

A.3. Critic prompt

```
You are the final critic and selector for benchmark-grade {library_name} expert QA.

## About {library_name}
{library_description}

## Benchmark reality
- Target primary label: `{label}`
- Label description: `{label_description}`
- You are selecting the final `{num_final}` items from a larger candidate set.
- Treat docs as potentially unavailable at answer time even though you may have seen them during generation.

## Selection criteria
- Keep only candidates that are clearly answerable from the code roots alone ({code_roots_inline}) --- no docs
↪ at answer time.
- Reject anything based on exact wording from documentation.
- Reject questions that are too easy, one-grep, or single-symbol lookups.
- **Reject questions that give away the locator.** The benchmark tests an agent with grep/glob/read over the
↪ repo --- locating the right code is half the challenge. If the question names a method/attribute on a class,
↪ an internal handler/adapter/parser/helper class, an internal helper, a file path, a test-file name, or a
↪ `snake_case` dotted function, it's a closed-book question. Rewrite to describe the behavior / symptom / user
↪ goal, keeping the specific symbol only in the gold `answer` and `code_evidence`. If rewriting would require
↪ fabricating a question unsupported by the seed or code, reject outright.
- **Reject "in the X example / tutorial / notebook / walkthrough / demo / README / guide" phrasings.** These are
↪ awkward and underspecified --- they point at an artifact as if a shared referent exists ("in the repo's
↪ multihop RAG example, ..."). A strong question stands on its own: rewrite to describe the *scenario* or
↪ *setup* itself (e.g., "in a multi-hop retrieval pipeline where the model refines its query across hops,
↪ ..."), or reject.
- **OK to keep**: branded user-facing concept names at the granularity a user would type:
{ok_to_name_bullets}
  The rule is "named the concept, not the attachment point." A branded class named as a concept is fine; the
↪ same class with a `.method` suffix is not.
- **Reject questions that are too generic to have one locator** (e.g., "how does {library_name} handle errors?"
↪ or "how does {library_name} do retries?"). A valid question is one where, after reading the repo, a careful
↪ expert would converge on the same specific file/symbol as the answer. Tighten generic questions by adding
↪ behavioral constraints, not by naming the class.
- **Reject questions whose gold answer rests on the seed's community answer as truth.** The community answers
↪ come from a weaker assistant and are frequently wrong. The gold answer must be supported by `code_evidence`
↪ pointing into actual {library_name} source. If the only support is "the seed said so," reject.
- Reject questions that copy or closely paraphrase sampled community questions.
- Reject anything outside the target label or overly similar to another candidate.
- Prefer diversity across subtopics within the label.
- You may rewrite the question, answer, difficulty, evidence, and note to improve quality.
- Keep final answers concise and well-grounded.
- `code_evidence` must contain real repo files under one of the code roots ({code_roots_inline}), and each
↪ filename must match the pattern `{file_glob}`; reject candidates that cite files with other extensions.

## Sampled community anchors
These are the same sampled sessions used during generation. They are for distribution anchoring only.
```

```
{sampled_sources_json}

## Candidate set to review
{candidate_json}

Return JSON that matches the provided schema and nothing else. If fewer than `{num_final}` candidates truly
↪ qualify, return fewer and explain the shortage in `selection_notes`.
```

A.4. Rubric-builder prompt

```
You are building a private grading rubric for one {library_name} expert QA benchmark question.

Your output is confidential and will only be used by the evaluator.

## Goal
- Turn the gold answer into 2-8 atomic grading claims.
- Claims should be small enough to score independently.
- Together, the claims should capture what a strong code-grounded answer must say.

## Rules
- Use only the provided question, gold answer, evidence references, and evidence file contents.
- Make every claim judgeable from code and tests alone.
- Use `core` for essential mechanisms or facts that define correctness.
- Use `supporting` for narrower detail, nuance, edge cases, or examples.
- Claims should be minimally overlapping.
- The claim weights must sum to exactly 100.
- `core` claims should carry most of the total weight.
- Every claim must cite 1-3 evidence spans.
- Every evidence span must come from the provided files only.
- Use exact line numbers from the numbered file dumps.
- Keep spans focused. Prefer 1-40 lines when possible, and never exceed 300 lines.
- Reuse spans across claims when that is the cleanest grounding.
- Do not include any public-release wording, benchmarking commentary, or grading instructions in the claim text.

## Inputs
- Question ID: `{question_id}`
- Label: `{label}`
- Question: `{question}`
- Gold answer:
{gold_answer}

## Evidence references
{evidence_references_json}

## Full evidence files
{evidence_files_text}

Return JSON that matches the schema exactly.
```

A.5. Grader prompt

```
You are grading one model answer for a private {library_name} expert QA benchmark.

## Scoring model
- The question gets one final continuous 0-100 score.
- Claims are only the internal rubric used to compute that question's score.
- Score each claim as:
  - `0` = wrong or missing
  - `0.5` = partially correct but incomplete, vague, or only partly grounded
  - `1` = fully correct
- Do not give extra credit for material outside the rubric.
- If an answer is polished but misses essential content, score the missing claims low.
- Use the evidence spans and gold answer to resolve ambiguity.

## Output rules
- Score every rubric claim exactly once.
- `question_score` must equal the weighted sum of the claim scores.
- Set `needs_regrade` to `true` only if the rubric or evidence is genuinely insufficient to judge the answer
↪ confidently.
- Keep rationales concise and specific.

## Inputs
- Question ID: `{question_id}`
- Label: `{label}`
```

```
- Question: `{question}`
- Model answer:
{model_answer}

## Gold answer
{gold_answer}

## Claim rubric
{claim_rubric_json}

## Evidence spans
{evidence_spans_json}

## Whole evidence files
{whole_evidence_text}

Return JSON that matches the schema exactly.
```

A.6. Worked examples

Below, we provide one example each for DSPy and OpenClaw. The official dataset is available at https://huggingface.co/datasets/jacobli/studybench.

Example 1: Study-DSPy

ID `dspy_3a5e956e4421` LABEL `evaluation_metrics_and_custom_eval`

QUESTION

> I've got a `dspy.ReAct` agent that answers arithmetic word problems and is given a Python `add` function as a tool. I want an evaluation harness that gives an example credit ONLY when the final answer is correct AND the agent genuinely used the calculator to get there. The reason I care: some of these agents just blurt out the right number and immediately finish without ever calling the tool, and those runs must score zero — a correct answer alone is not enough.
>
> My current metric pulls the tool calls off the prediction (I look at `pred.tool_calls` like I would with native function-calling / `dspy.ToolCalls`) and checks whether `add` is among them, but it never awards credit — even on runs where I can see from the logs that the agent clearly did call `add`. So my whole devset scores 0%.
>
> Give me a small, runnable harness (devset of a couple of arithmetic examples, a custom metric, and a run over the devset that prints an overall percentage plus per-example scores) that scores this correctly. Use `DummyLM` so it runs offline with no API key.

GOLD REFERENCE IMPLEMENTATION

```python
import dspy
from dspy.evaluate import Evaluate
from dspy.utils.dummies import DummyLM


def add(a: int, b: int) -> int:
    return a + b


def tool_use_metric(example, pred, trace=None):
    # Credit only if the agent actually invoked the `add` tool. Note that ReAct always
    # records a `finish` step in the trajectory, so we must match the specific tool name,
    # not merely "some tool was called".
    used_add = any(
        key.startswith('tool_name_') and value == 'add'
        for key, value in pred.trajectory.items()
    )
    correct = str(example.answer) == str(pred.answer)
    return float(used_add and correct)


devset = [
    dspy.Example(question='What is 2 + 2?', answer='4').with_inputs('question'),
```

```
    dspy.Example(question='What is 3 + 4?', answer='7').with_inputs('question'),
]


dspy.configure(
    lm=DummyLM(
        [
            {'next_thought': 'I should add the numbers.', 'next_tool_name': 'add', 'next_tool_args': {'a': 2, 'b': 2}},
            {'next_thought': 'I have the result, so I can finish.', 'next_tool_name': 'finish', 'next_tool_args': {}},
            {'reasoning': 'The tool returned 4.', 'answer': '4'},
            {'next_thought': 'I already know the answer, so I can finish.', 'next_tool_name': 'finish', 'next_tool_args': {}},
            {'reasoning': 'It is 7.', 'answer': '7'},
        ]
    )
)

agent = dspy.ReAct('question -> answer', tools=[add], max_iters=2)
result = Evaluate(
    devset=devset,
    metric=tool_use_metric,
    num_threads=1,
    display_progress=False,
)(agent)

per_example = [
    {'question': example.question, 'tool_use_score': score}
    for example, _, score in result.results
]

print({'overall_percent': result.score, 'per_example': per_example})
```

EVIDENCE FILES
`dspy/predict/react.py`,
`dspy/evaluate/evaluate.py`,
`tests/predict/test_react.py`.

CLAIM RUBRIC

- **c1** (`core`, weight 70, spans s1, s2, s3).
  The custom metric awards credit only when BOTH (a) the prediction's final answer matches the example answer AND (b) the agent actually invoked the arithmetic tool, and it detects (b) by reading the ReAct trajectory: it iterates `pred.trajectory` (the dict ReAct returns) and checks for a `tool_name_*` entry whose VALUE equals the specific tool name `'add'`. It must NOT read tool usage from `pred.tool_calls` / `pred.trace.tool_calls` / any `dspy.ToolCalls`-style native-function-calling field (ReAct does not populate those), and it must NOT pass on the mere presence of any `tool_name_*` entry or a non-empty trajectory — because ReAct ALWAYS records a `finish` step (`tool_name_* == 'finish'`), a correct answer whose trajectory contains only a `finish` step (no `add` call) must score zero.
- **c2** (`core`, weight 10, spans s1, s2).
  The evaluated program is a `dspy.ReAct` agent constructed with the arithmetic function passed in `tools=[...]`, so each prediction carries a `trajectory` attribute recording the per-step `tool_name_*`/`tool_args_*`/`observation_*` entries that the metric inspects (rather than an OpenAI-style `tool_calls` field on the prediction).
- **c3** (`core`, weight 12, span s4).
  The harness runs `dspy.evaluate.Evaluate` over a devset of `dspy.Example`s (declared runnable, i.e. the input is marked with `.with_inputs('question')`) using the custom metric; `Evaluate` invokes the program per example as `program(**example.inputs())`, scores it via `metric(example, prediction)`, and returns an overall percentage plus per-example `(example, prediction, score)` results.
- **c4** (`supporting`, weight 8, span s4).
  The example reports results by reading from the `Evaluate` return value: it extracts the per-example tool-use scores from `result.results` (the `(example, prediction, score)` tuples) and prints them alongside the overall percentage from `result.score`.

Example 2: Studying-OpenClaw

ID `openclaw_53957e5b2e33` LABEL `model_fallback_and_failover_logic`

QUESTION

I've got a set of isolated cron jobs, each scheduled independently. Most should use my usual model-fallback chain, but a couple of high-priority jobs I want to pin to a single model with no fallbacks at all — if that model is down, just fail rather than silently drift onto a cheaper backup.

So on those jobs I cleared out the fallback list (saved it as an empty list on the job) and left the rest untouched. My understanding of how the per-job resolution is supposed to work: if a job doesn't carry its own fallback list, it inherits the agent's configured fallbacks; if it does carry one, we use that. Since "empty" is the natural way to express "no list here," I expected the empty-list jobs to behave like the un-pinned ones and just inherit — but I actually want the opposite for them, and I'm second-guessing whether clearing the list is even the right signal.

Write the per-job resolver that produces the effective fallback chain for one of these scheduled jobs, so that a job which deliberately specifies "no backups" is honored as exactly that, while jobs that simply never set a list keep getting the normal agent-level chain. Match how the rest of this codebase already distinguishes those two states.

GOLD REFERENCE IMPLEMENTATION

```ts
import type { OpenClawConfig } from "../../config/types.openclaw.js";
import type { CronJob } from "../types.js";
import { resolveEffectiveModelFallbacks } from "./run-execution.runtime.js";

export function resolveCronFallbacksOverride(params: {
  cfg: OpenClawConfig;
  job: CronJob;
  agentId: string;
}): string[] | undefined {
  const payload = params.job.payload.kind === "agentTurn" ? params.job.payload : undefined;
  const payloadFallbacks = Array.isArray(payload?.fallbacks) ? payload.fallbacks : undefined;
  const hasCronPayloadModelOverride =
    typeof payload?.model === "string" && payload.model.trim().length > 0;

  return (
    payloadFallbacks ??
    resolveEffectiveModelFallbacks({
      cfg: params.cfg,
      agentId: params.agentId,
      hasSessionModelOverride: hasCronPayloadModelOverride,
    })
  );
}
```

EVIDENCE FILES

`src/cron/isolated-agent/run-fallback-policy.ts`,
`src/cron/isolated-agent/run.payload-fallbacks.test.ts`.

CLAIM RUBRIC

- **c1** (`core`, weight 42, spans s1, s3).
  (outcome) Honors an explicit empty per-job list as "no backups": when the agentTurn payload's `fallbacks` is an empty array it is returned as-is and the agent chain is NOT inherited; the ONLY input that inherits is a truly absent (`undefined`) list. It must operate on the cron agentTurn payload's own `fallbacks` field (not an invented job field), and must NOT treat `[]` as unset (no `.length`, `||`, or other truthiness collapse that would route an empty list to inheritance).
- **c2** (`core`, weight 28, spans s1, s3).
  (mechanism) Decides unset-vs-set purely by array PRESENCE, matching this repo's convention — `Array.isArray(payload.fallbacks)` (or an equivalent `=== undefined` presence check) selects the per-job list, and falls through to the inherited resolver with `??`, so the empty array survives as a real value. It must NOT branch on `.length`, truthiness, or `||`, which would silently merge `[]` into the unset case.
- **c3** (`core`, weight 15, span s1).
  When the per-job list is absent (`undefined`), it inherits by delegating to the repo's effective fallback resolver `resolveEffectiveModelFallbacks` called with `cfg` and `agentId` — not by a hand-rolled agent/global default lookup or an invented `agent.defaultFallbackChain`.

- **c4** (`supporting`, weight 9, spans s1, s2).
  Returns the job's own `payload.fallbacks` array verbatim (overriding the inherited chain) when the payload kind is `agentTurn` and `payload.fallbacks` is a defined non-empty array.
- **c5** (`supporting`, weight 6, span s1).
  Computes `hasSessionModelOverride` from a non-empty trimmed `payload.model` and passes it into the inherited resolver, and gates the whole resolution on `payload.kind === "agentTurn"` so non-agentTurn payloads carry no fallback override.

B. Hyperparameters and sampling parameters

Table 4. Study-procedure recipes for the reported STUDYBENCH runs on Qwen3.5-9B.

Procedure	Study material	Recipe
No study	none	none
CPT(code)	DSPy source and tests, roughly 459k tokens, tokens are separated by files; files longer than 6k tokens are truncated	LoRA rank 128, alpha 32, dropout 0; learning rate 10^{-4}; batch size 4, grad accumulation 8. Self-sampled MBPP traces mixed in as an anchor. For OpenClaw the same recipe runs over the OpenClaw codebase.
CPT(doc)	DSPy documentation, roughly 160k tokens	Same LoRA and optimizer recipe as CPT(code).
Synthetic fine-tuning	DeepSeek-V4-Flash question-answer pairs plus a recovery mixture of roughly 60k examples	Supervised training on the generated pairs with reasoning disabled, followed by on-policy distillation recovery. We adopt the hyperparameters from `tinker_cookbook` (Thinking Machines Lab, 2025). LoRA rank 256, alpha 32, dropout 0; max sequence length 8192; learning rate 4.9×10^{-4}; effective batch size 32.
Cheatsheet	DSPy source and tests through the same three tools as evaluation	The agent runs a forced ReAct study loop with at least 50 no-early-return tool calls and writes a cheatsheet, which is prepended to every later question while the repository tools remain available. The same scaffold is used to produce the OpenClaw cheatsheet.

During evaluation, all Qwen3.5-9B experiments use the official sampling parameters for thinking mode and general tasks [4]. Specifically, we set the temperature to 1.0, top-p to 0.95, top-k to 20, min-p to 0, presence penalty to 1.5, and repetition penalty to 1.0, with a maximum generation length per ReAct turn of $32,768$.

C. Calculating Expertise

We define an expert in domain $\mathbf{D}$ as *an agent that can efficiently convert inference compute into accurate results*. While a capable novice may eventually succeed on an open-book exam through brute force, only an expert produces high-quality answers efficiently and achieves more as additional time is provided. We therefore assess expertise as the *weighted area* under the agent's performance curve as inference compute increases,

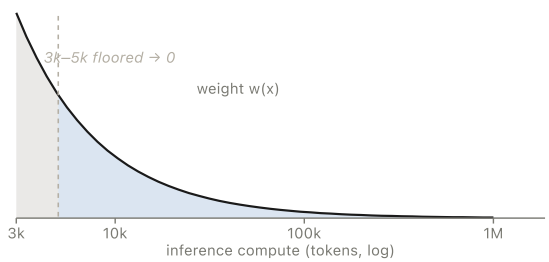
$$\text{Expertise} \;=\; \int \underbrace{\text{performance at an inference budget}}_{\text{quality}} \times \underbrace{\text{importance of that budget}}_{\text{weight}}. \tag{1}$$

Concretely, writing $p_{\Sigma,\mathbf{D}}(x)$ for the performance of an agent Σ on domain $\mathbf{D}$ at position x of a log-token axis, expertise is the weighted average of performance over log-compute,

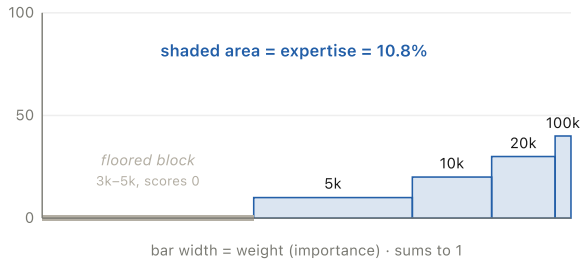
$$\mathcal{E}(\Sigma;\mathbf{D}) \;=\; \int p_{\Sigma,\mathbf{D}}(x)\, w(x)\, dx, \qquad w(x)\geq 0, \quad \int w(x)\, dx = 1. \tag{2}$$

The weight w encodes the relative importance of each budget, and normalizing it ($\int w = 1$) keeps expertise on the same scale as performance. We anchor the axis so that $x = 0$ corresponds to 3k generated tokens (about the minimum a modern reasoning model needs to read a question and give a complete answer) and each increment of $+1$ represents a tenfold

[4] https://huggingface.co/Qwen/Qwen3.5-9B



(a) Where the weight lives. Importance halves every $2\times$ compute, crowding the cheap end; here the 3k–5k block is floored to 0, so that weight is wasted.



(b) The expertise rectangles. Each budget is a rectangle of *width* its weight and *height* its score; the shaded staircase area is the expertise (10.8%). The floored 3k–5k block ($\approx 40\%$ of the weight) sits at zero height.

Figure 4. Expertise as a weighted area under the performance curve, at $\lambda = \ln 10$ (weight halves every $2\times$ compute) with a 3k anchor. Numbers match the worked example.

increase in tokens. Also note that since x is non-negative as it reflects inference compute, the integration $\int w(x)\,dx = 1$ is well-defined. We use the exponential decay

$$w(x) = (\ln 10)\, 10^{-x}, \tag{3}$$

so that each doubling of compute halves a budget's weight; importance therefore crowds the cheap end of the axis (Figure 4a).

Estimating $\mathcal{E}$ from a few budgets. In practice we evaluate a limited set of budgets and read $p_{\Sigma,\mathbf{D}}(x)$ as the best score the agent achieves using *at most* x tokens, (a step function). For budgets below the first measured point the score defaults to the minimum (zero, or chance for multiple choice), reflecting that the agent simply cannot answer with fewer tokens; beyond the last measured budget we carry the final score forward, since the tail holds little weight anyway.

Worked example. Suppose budgets of 5k, 10k, 20k, 100k tokens earn scores of $10\%, 20\%, 30\%, 40\%$. Integrating the importance density (3) over each budget's catchment region on the log-token axis gives weights of $0.30,\ 0.15,\ 0.12,\ 0.03$, so that

$$\widehat{\mathcal{E}} = 0.30 \cdot 10 + 0.15 \cdot 20 + 0.12 \cdot 30 + 0.03 \cdot 40 \;\approx\; 10.8\%. \tag{4}$$

These weights sum to 0.60, and the remaining 0.40 falls in the region below 5k tokens, where the score is floored to zero (Figure 4b). Note that raw per-budget accuracy is not necessarily monotone as compute grows in practice. We therefore empirically take $p_{\Sigma,\mathbf{D}}(x)$ to be the best score achieved at or below cost x.

D. Studying Intelligence

For the purposes of machine studying, an *intelligent* agent is one that can *quickly acquire expertise* in totally new domains. Thus intelligence measures how well a studying algorithm converts *study* compute into expertise.

Concretely, we apply the construction of Appendix C one level up. Fix an agent Σ, a corpus $\mathbf{D}$, and a studying algorithm π. For each amount of study compute s, we let π study $\mathbf{D}$ with budget s, plot the resulting agent's full performance curve, and reduce it to an expertise score

$$\mathcal{E}_\pi(s) \;:=\; \mathcal{E}\big(\pi_s(\Sigma, \mathbf{D});\, \mathbf{D}\big), \tag{5}$$

where π_s denotes studying under budget s. Reading $\mathcal{E}_\pi(s)$ as a function of study compute gives an *expertise-vs-study-compute* curve, and the weighted area under *this* curve is the (studying) intelligence:

$$\mathcal{I}(\pi, \Sigma; \mathbf{D}) \;=\; \int \underbrace{\mathcal{E}_\pi(s)}_{\text{expertise after a study budget}} \omega(s)\, ds, \qquad \omega(s) \geq 0, \quad \int \omega(s)\, ds = 1, \tag{6}$$

where s runs along a log–study-compute axis and ω is an importance weight of the same exponential-decay form as (3), so that cheaper studying is emphasized, although a different, potentially slower decay rate might be used to calculate the intelligence score as study-compute is conceptually cheaper than inference-compute. Intelligence is thus a weighted average of expertise, on the same scale as expertise.