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

Anonymous Authors[1]

Abstract

Humans study in various ways, like reading textbooks, working through examples, or thinking out loud. Underneath these activities is the capacity to convert preparation into expertise, and we argue that this belongs at the center of how agentic AI systems should adapt. We call the machine analogue of it *machine studying*: the work that an agentic system performs on itself, given a corpus and no preview of the downstream tasks, that pays off at test time. Potential approaches for machine studying are open-ended, ranging across gradient descent, harness design, context optimization, and combinations thereof. The payoff from studying manifests as fewer mistakes or faster answers. We instantiate the framework in an initial setting in which agents must teach themselves the concepts and usage of an extensive software library from nothing but a corpus of its internal code, with conceptual and coding tasks and study procedures that act on weights, context, and harness. We offer this as the new frame through which continual adaptation should be studied.

1. Introduction

A coherent pipeline for capable agentic AI is now visible (Shao et al., 2025; Shen et al., 2026; Gonzalez-Pumariega et al., 2026). Beyond the canonical recipe of pre-training and post-training, recent progress depends on inference-time compute (Wei et al., 2025), agentic scaffolds (Yao et al., 2022), and tool ecosystems (Anthropic, 2024). These systems reason, search, and complete long-horizon tasks across many domains, often at high levels of performance.

However, performance falters when a task depends on information that is absent from pretraining or buried in the tail. Take a shifted software API as an example, the evidence may be present in the repository or documentation, yet the agent still fails to search or apply it reliably (Liu et al., 2024). Even when the right evidence is encountered, the lesson often does not persist (Dupoux et al., 2026). This is often discussed as *continual learning* (Silver et al., 2013), but the phrase now covers too many distinct settings to serve as a precise problem definition and no longer cleanly specifies what should be learned or how success should be measured

Conversely, humans often become competent in a subject from a modest declarative corpus: textbooks, papers, documentation, worked examples, exercises, and notes (Chi et al., 1994; Renkl, 2014; Dunlosky et al., 2013; Fiorella, 2023; Renkl, 2014). The mechanisms vary. A student may memorize, annotate, summarize, practice, or look things up. What matters is not which of these mechanisms is privileged, but how preparation changes later use of the material.

We ask the analogous question for agentic AI systems. Suppose a system receives a declarative corpus before evaluation, but does not know what questions will later be asked. What should it do with the corpus before the task arrives? Should it rewrite its weights, prepare context, build an index, synthesize examples, change its harness, or simply wait and search at test time? These choices are all plausible; none should be treated as free.

We call this *machine studying*. The setting is deliberately narrower than the standard sequential-task view of continual learning. We do not begin with a stream of rewards or labels, nor do we ask whether task ten preserves task one. Those questions remain important, but they assume a more basic ability: that the system can extract usable competence from the first body of material. Machine studying isolates this first-exposure problem. A corpus may have internal order—chapters, examples, files, tests—and a study procedure may exploit that order, but the central unit is a *corpus of documents* rather than a task stream.

This focus changes the role of benchmark score. A corpus-grounded benchmark is rarely meaningful as a saturation test. A sufficiently aggressive system can often buy accuracy by spending more computation: inspect every file before answering question at inference time (Zhang et al., 2025a),

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the ICML 2026 Workshop "Continual Adaptation at Scale: Towards Sustainable AI". Do not distribute.

construct a corpus-specific retriever (Esfandiarpoor et al., 2025), or pre-train the model from scratch with additional data and doing the whole mid-train and post-train pipeline. These are legitimate systems, and in some settings they may be the right ones. But they occupy different points in the same accounting problem. The object of evaluation is therefore the tradeoff among study-time compute, test-time compute, and downstream performance.

We instantiate this framework in an initial MS-BENCH setting in which agents must teach themselves the concepts and usage of the latest version of the DSPy (Khattab et al., 2023) library from nothing but a corpus of its internal code. The system receives a real software corpus before evaluation and is tested on hidden conceptual and coding questions grounded in that corpus. We compare study procedures that act on weights, context, and harnesses, and evaluate each under multiple test-time budgets. The empirical question is not whether a stronger coding agent could eventually obtain a higher score. It is whether prior exposure to the corpus moves the compute–performance frontier relative to doing more work only after the question arrives.

We make three contributions. First, we formulate machine studying as an open-book, hidden-task adaptation problem for compound AI systems. Second, we instantiate the setting in MS-BENCH, a software-corpus benchmark that reports study-time compute, test-time compute, and downstream performance rather than a single score. Third, we provide an initial frontier analysis across representative study procedures, showing how weight updates, context-side preparation, and test-time search can be compared within the same accounting framework.

2. Motivation and Related Work

Classical continual learning derives its closed-book character from a simple restriction. After experience, only parameters persist. New data update weights, and later capability must be recovered from those weights, so catastrophic interference becomes the canonical failure mode (McCloskey & Cohen, 1989; Li & Hoiem, 2017; Kirkpatrick et al., 2017; Ratcliff, 1990). The success of in-context learning with LLMs showed that the input itself can induce new behavior without parameter updates (Brown et al., 2020). The immediate response has been to make context larger, up to millions of tokens (DeepSeek-AI, 2026) or even beyond (Tandon et al., 2025). Context rot shows the limit of length alone, since relevant material can remain in the window while reliability degrades (Hong et al., 2025). This motivates context management methods that make carried material easier to use (Jiang et al., 2023; 2024). A further line moves the effect of context into persistent model state. Context distillation trains behavior elicited with the relevant context present (Snell et al., 2022); other work learns a single-pass map from an input context to LoRA adapters, amortizing context distillation (Charakorn et al., 2026; Liu et al., 2026); and training procedures can pack corpus information into reusable KV caches (Eyuboglu et al., 2025; Zweiger et al., 2026). These methods broaden the machinery while preserving the same answer-time demand. The needed material must already be present or internalized before the system can use it. Under this framing, adaptation is driven toward perfect memorization followed by perfect generalization. Every fact that is misremembered or unsurfaced becomes a failure point. Machine studying starts from expertise settings where the corpus remains part of the world, and the central question becomes how prior exposure changes the cost of later use.

3. Machine Studying

We represent an agentic system as an M-CHANT tuple $\Sigma = (\mathbf{M}, \mathbf{C}, \mathbf{H}, \mathbf{A}, \mathbf{N}, \mathbf{T})$, where $\mathbf{M}$ is the underlying model, $\mathbf{C}$ the available context, $\mathbf{H}$ the harness (e.g., ReAct scaffolds (Yao et al., 2022), RLM control flow (Zhang et al., 2025a), language model programs such as DSPy (Khattab et al., 2023), and inference policies that regulate deliberation style (Alomrani et al., 2025; OpenAI, 2026)), $\mathbf{A}$ non-neural assets such as corpora and indices, $\mathbf{N}$ neural auxiliaries coupled to the base model but external to its core parameters (e.g., modular adapters (Pfeiffer et al., 2023) and prefix-tuning states (Li & Liang, 2021)), and $\mathbf{T}$ the tool set.

Study procedures. Let $\mathbf{D}$ denote a declarative corpus. We assume $\mathbf{D}$ exceeds the system's high-quality context window and lacks labels, rewards, or an explicit task distribution. A study procedure is what an agentic system does to itself before evaluation, with access to $\mathbf{D}$ but no advance knowledge of the test. For procedure π, we write

$$\Sigma^{\pi}_{\mathbf{D}} \ = \ \pi(\Sigma, \mathbf{D})$$

for the system after studying; the procedure may rewrite any component of the M-CHANT tuple. A degenerate no-study procedure leaves Σ unchanged and shifts all adaptation to test time.

Open-book hidden-task evaluation. Expertise is measured on a hidden downstream environment induced by $\mathbf{D}$. The system receives the material before evaluation but not the test, and may hypothesize what will matter only from the material itself. Crucially, $\mathbf{D}$ remains available at inference through $\mathbf{A}$ and $\mathbf{T}$.

4. Methods

We evaluate machine studying by fixing a corpus, an answering model, a study procedure, and a test-time harness. The study procedure receives the corpus before evaluation;

Table 1. Qwen3.5-4B on DSPy QA. **Code** and **Conceptual** each contain 30 questions, graded on a 0–100 scale by GPT-5.4 (OpenAI, 2026) and averaged over 3 rollouts at temperature 0.6. *direct* uses no tools. ReAct-5 and ReAct-20 allow up to 5 or 20 repository-tool iterations with early stopping. Scout pre-builds a cheatsheet during study and prepends it at inference.

	Code			Conceptual		
Method	*direct*	ReAct-5	ReAct-20	*direct*	ReAct-5	ReAct-20
Bare Qwen3.5-4B	9.8	23.2	31.0	20.3	50.6	56.8
Continued pretraining						
CPT-docs, 1 epoch	10.3	25.0	28.6	21.0	50.6	56.3
CPT-docs, 3 epochs	10.3	26.5	31.4	18.5	54.1	58.1
CPT-docs, 5 epochs	9.9	22.5	28.8	20.7	54.5	57.8
CPT-code, 1 epoch	10.0	27.1	27.5	20.5	53.7	57.2
CPT-code, 3 epochs	9.2	23.1	30.9	22.4	50.9	56.7
CPT-code, 5 epochs	10.2	24.8	28.4	20.5	52.4	56.8
On-policy context distillation						
SFT, 1 epoch	9.1	23.0	27.6	22.1	51.5	58.2
SFT, 2 epochs	9.0	24.7	30.6	20.0	51.8	54.9
SFT, 3 epochs	9.1	26.8	31.6	22.0	55.1	54.3
Scout						
Qwen3.5-4B cheatsheet	16.3	26.4	29.4	19.5	56.0	58.6
GPT-5.4 cheatsheet	17.4	23.1	31.2	35.1	60.8	58.6
Forced inference-time compute						
20 forced tool calls	—	—	39.0	—	—	60.2

the harness controls the work available after a question is revealed. Across the full sweep, we evaluate four model families (Qwen3.5 (Qwen Team, 2026), Qwen3 (Qwen Team, 2025), Gemma4 (Gemma Team, 2026), and GPT-5.4), three main harnesses (`direct`, ReAct-5, ReAct-20), and one forced-compute diagnostic. We compare four study-procedure families: no study, continued-pretraining LoRA (Hu et al., 2022), on-policy context distillation (Snell et al., 2022; Zhao et al., 2026), and Scout. The main paper reports the complete Qwen3.5-4B study, and addition results can be found in Appendix D; Appendix B reports the hyperparameters and training recipes. Experiments used approximately 500 NVIDIA H200 GPU-hours.

Harnesses. In `direct`, the model answers from the question alone. In ReAct-5 and ReAct-20, the model receives read-only repository tools, `glob_files`, `grep_code`, and `read_file`, scoped to the visible DSPy source and tests. The budget is 5 or 20 tool iterations, with early stopping allowed. The forced-compute diagnostic uses the same tools but requires 20 tool calls before answering.

Study procedures. No study leaves the system unchanged. Continued-pretraining LoRA trains adapters by next-token prediction on either source plus tests (CPT-CODE) or official documentation (CPT-DOCS), with checkpoints after 1, 3, and 5 epochs. On-policy context distillation uses DeepSeek-v4-Flash (DeepSeek-AI, 2026) to generate file-grounded questions, then runs the answering model with privileged access to the relevant source evidence and trains a LoRA student that sees only the question to match that continuation under prompt-masked cross-entropy. Scout is a context-side study procedure. During study, the builder is given the repository tools and a study prompt, explores the corpus for at least 50 tool iterations, and writes a cheatsheet. The cheatsheet is prepended to each model call, and the same repository tools remain available. We report a self-generated Qwen3.5-4B cheatsheet and a GPT-5.4-generated cheatsheet.

5. Benchmark

We instantiate MS-BENCH on a local snapshot of DSPy, totaling 426K tokens across source and tests. The benchmark contains 30 conceptual questions and 30 coding questions, anchored in DSPy user questions rather than synthetic topic selection. We filter, deduplicate, and cluster the user-question pool, then use the resulting clusters as seeds for benchmark construction. GPT-5.4, running in Codex (OpenAI, 2025) at `xhigh` reasoning effort, generates candidate questions, gold answers, and code evidence from the seeded distribution and repository context. A separate GPT-5.4 pass constructs private claim-level rubrics from the gold answers and evidence. A human DSPy expert verifies the final items for answer correctness and evidence support. Although we instantiate the pipeline on DSPy here, the construction is repository-agnostic: any codebase with real user questions, GitHub issues, or comparable community traces can run through the same pipeline. Appendix A gives the filtering and clustering procedure, generation and critique prompts, naming-discipline rules, and curation details; Appendix C gives the rubric construction and grading protocol.

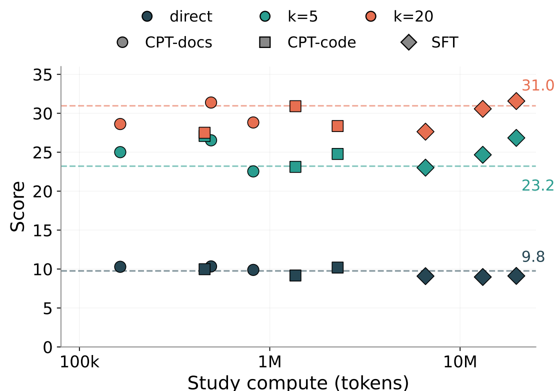

Figure 1. Colors denote the three inference-time compute budgets. Marker shapes denote different forms of studying effort via gradient updates of different epochs. The gradient-update variants mostly fluctuate at a fixed inference budget. The dominant movement comes from increasing inference-time compute rather than from switching among the tested gradient-update procedures.

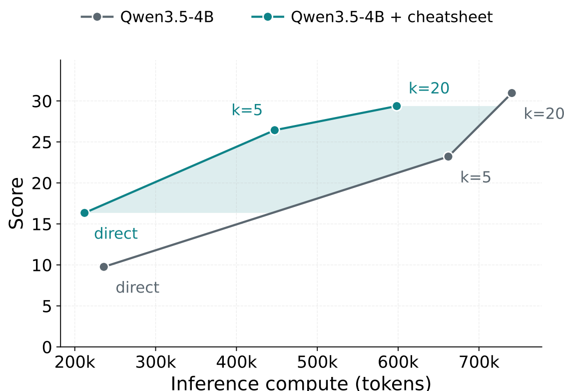

Figure 2. Self-generated Scout shifts the coding compute–accuracy frontier. The x-axis is total completion tokens over 30 questions and 3 rollouts. Reusing a study-time cheatsheet improves low-budget inference and reaches a competitive high-budget region with less evaluation-time output.

6. Empirical Pattern

Test-time search dominates the observed weight updates. Table 1 shows the main pattern. For bare Qwen3.5-4B, moving from direct answering to ReAct-20 raises code score from 9.8 to 31.0 and conceptual score from 20.3 to 56.8. The largest gains from weight-side study are much smaller at the same harness budget. Relative to the corresponding bare cell, the best code gain is +3.9 from CPT-code at ReAct-5, and the best conceptual gain is +4.5 from SFT at ReAct-5. Direct and ReAct-20 settings are nearly flat. In this regime, the largest movement comes from allowing the system to search at test time rather than from the studying.

The weight-update result should be read against the density of synthetic study signal available. Although our context-distillation run is substantial in absolute terms—roughly 25K file-grounded questions generated from about 240 eligible Python files—it covers roughly 426K visible source/test tokens, yielding only 0.06 synthetic questions per corpus token, or about one question per 17 corpus tokens. We call this quantity *synthetic-practice density*: the number of corpus-conditioned synthetic questions generated per token of material to be studied. Table 2 compares this density with related corpus-conditioned training regimes. The comparison is a scale calibration rather than an outcome comparison, but it shows that our gradient-study point is low-density relative to regimes that report stronger internalization effects.

Table 2. Synthetic-practice density. We normalize generated corpus-conditioned synthetic questions by size of study corpus.

Setting	Corpus tokens	Questions	Q / token
DSPy QA, ours	426K	25K	0.06
LongHealth[1]	100K	320K	3.2
MTOB[1]	60K	179K	3.0
RuleArena NBA-L2[2]	20K	16K	0.80

Context-side study shifts the frontier. Scout is the clearest positive study result in this setting. During study, the system explores the corpus upfront and writes a compact artifact. During evaluation, that artifact is reused inside the prompt for every question while the repository tools remain available. The cheatsheet does not need to replace search; it just allows the system to approach each question with a reusable map of the corpus rather than beginning cold. This produces the expected frontier movement. The gains are largest when the inference budget is small, where prior preparation gives direct answering and ReAct-5 a better route through the corpus and lowers inference-time output size. With a larger ReAct budget, test-time search can recover more information on its own, so the accuracy gap narrows while Scout still uses less inference work. In general, a useful study procedure should be credited when prior corpus exposure produces a reusable system change that improves the tradeoff between study compute, test compute, and performance. We are not claiming that a cheatsheet is the final study mechanism but that a simple, auditable context-management procedure produces the most visible frontier shift in our tests. This is an empirical reason to evaluate studying at the level of the whole compound system.

[1] LongHealth and MTOB are cited as underlying task sources (Adams et al., 2025; Tanzer et al., 2023). The densities shown here use the task variants and corpus-conditioned training schedules reported in (Eyuboglu et al., 2025), not necessarily the original benchmark configurations.

[2] RuleArena NBA-L2 is a task setting from (Zhou et al., 2025) used in SIEVE (Asawa et al., 2026). The table reports unique generated questions. SIEVE trains for two epochs over the 16K questions, giving 1.6 question exposures per corpus token.

220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274

References

Adams, L., Busch, F., Han, T., Excoffier, J.-B., Ortala, M., Löser, A., Aerts, H. J., Kather, J. N., Truhn, D., and Bressem, K. Longhealth: A question answering benchmark with long clinical documents. *Journal of Healthcare Informatics Research*, 9(3):280–296, 2025.

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.

Asawa, P., Dimakis, A. G., and Zaharia, M. Sieve: Sample-efficient parametric learning from natural language. *arXiv preprint arXiv:2604.02339*, 2026.

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.

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.

Dupoux, E., LeCun, Y., and Malik, J. Why ai systems don't learn and what to do about it: Lessons on autonomous learning from cognitive science. *arXiv preprint arXiv:2603.15381*, 2026.

Esfandiarpoor, R., Suryanarayanan, V., Bach, S. H., Chowdhary, V., and Aue, A. Themcpcompany: Creating general-purpose agents with task-specific tools, 2025. URL https://arxiv.org/abs/2510.19286.

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.

Gemma Team. Gemma 4, 2026. URL https://deepmind.google/models/gemma/.

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.

Hong, K., Troynikov, A., and Huber, J. Context rot: How increasing input tokens impacts llm performance. Technical report, Chroma, July 2025. URL https://trychroma.com/research/context-rot.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al. Lora: Low-rank adaptation of large language models. *Iclr*, 1(2):3, 2022.

Jiang, H., Wu, Q., Lin, C.-Y., Yang, Y., and Qiu, L. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 13358–13376, 2023.

Jiang, H., Wu, Q., Luo, X., Li, D., Lin, C.-Y., Yang, Y., and Qiu, L. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1658–1677, 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.

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.

Li, Z. and Hoiem, D. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2017.

Liu, Y., Wang, X., Mao, Y., Gelbery, 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.

Liu, Z. L., Pandit, S., Ye, X., Choi, E., and Durrett, G. Codeupdatearena: Benchmarking knowledge editing on api updates. *arXiv preprint arXiv:2407.06249*, 2024.

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 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

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

Ratcliff, R. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review*, 97(2):285, 1990.

Renkl, A. Toward an instructionally oriented theory of example-based learning. *Cognitive Science*, 38(1):1–37, 2014.

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.

Silver, D. L., Yang, Q., and Li, L. Lifelong machine learning systems: Beyond learning algorithms. In *AAAI Spring Symposium: Lifelong Machine Learning*, volume 13, 2013.

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.

Tanzer, G., Suzgun, M., Visser, E., Jurafsky, D., and Melas-Kyriazi, L. A benchmark for learning to translate a new language from one grammar book. *arXiv preprint arXiv:2309.16575*, 2023.

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

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*, 2022.

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.

Zhao, S., Xie, Z., Liu, M., Huang, J., Pang, G., Chen, F., and Grover, A. Self-distilled reasoner: On-policy self-distillation for large language models, 2026. URL https://arxiv.org/abs/2601.18734.

Zhou, R., Hua, W., Pan, L., Cheng, S., Wu, X., Yu, E., and Wang, W. Y. Rulearena: A benchmark for rule-guided reasoning with llms in real-world scenarios. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 550–572, 2025.

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

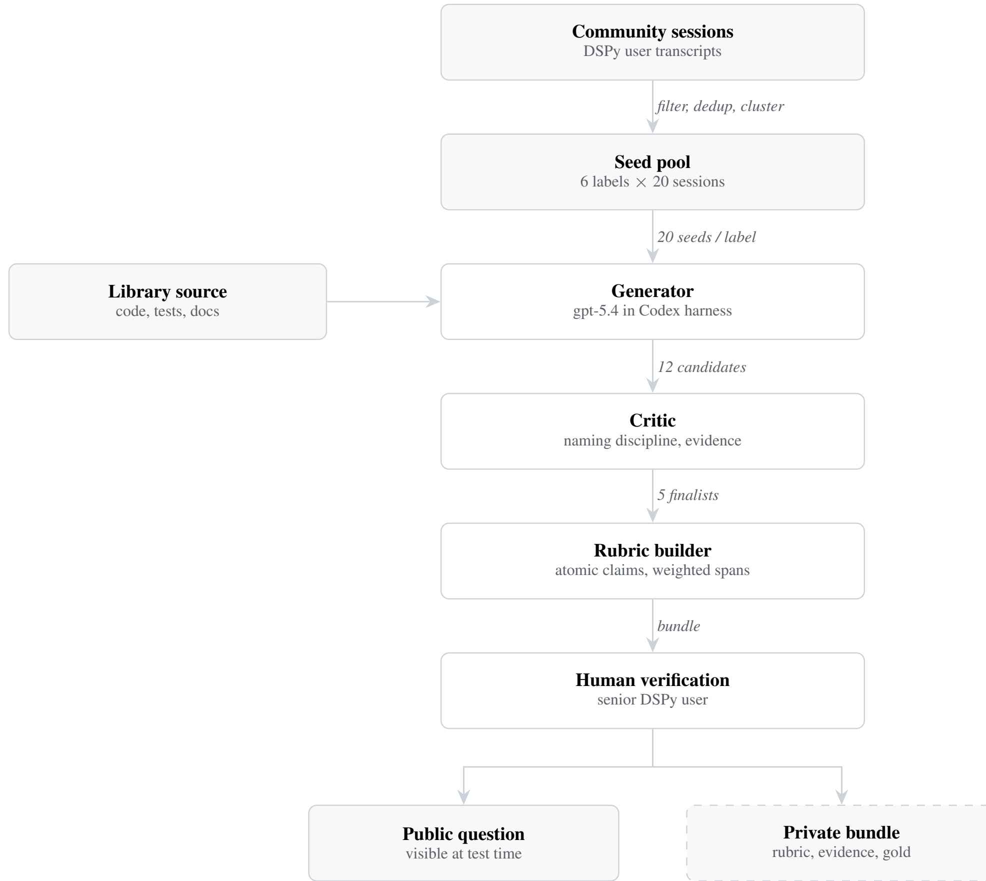

Figure 3. DSPy QA construction pipeline. Community sessions are filtered and clustered into a per-label seed pool that anchors the question distribution. gpt-5.4 in a Codex-style harness, with read access to the full repository, lifts each cluster's seeds into candidate items via a generator/critic pair; documentation is available at generation time but not at test time. A rubric builder converts each finalist into an atomic-claim grading bundle, and a senior DSPy user verifies every item before release. The public question is exposed to the answering system; the rubric, evidence spans, and gold answer stay private.

A. DSPy QA: Construction and Validation

This appendix specifies the construction pipeline behind the DSPy QA suite, the prompt templates that drive it, the DSPy-specific values plugged into those templates, and worked examples from the final human-verified benchmark. The same pipeline produces both conceptual and coding suites; the only modality-specific differences are in the generator and critic contracts.

A.1. Construction pipeline

The DSPy QA suite is constructed by a five-stage pipeline that preserves a real user-friction distribution at every step and ends with a human verification gate. Figure 3 shows the end-to-end flow; the stages below describe each step.

Stage 1: source distribution. We start from a snapshot of real DSPy user-question sessions provided directly by the DSPy community. Each session is filtered by length, detected language (English only), and question-shape (the first substantive turn must begin with an interrogative or imperative form, e.g. *how*, *what*, *why*, *can*, *does*, *explain*, *show*, *help*). We then deduplicate exactly on the representative question text, and near-deduplicate using MinHash with $num_perm = 128$ and a Jaccard threshold of 0.7 over question shingles. Only sessions created strictly after April 2025 are kept.

Stage 2: clustering. Each surviving session is represented by its first substantive user question, embedded with Qwen3-Embedding-8B (Zhang et al., 2025b) under a domain-aware prefix prompt, then projected to ten dimensions with UMAP (McInnes et al., 2018) ($n_neighbors = 15$) and clustered with HDBSCAN (Campello et al., 2013)

(`min_cluster_size`=30, `min_samples`=5). GPT-5.4 names each cluster into a behavioral label by reading 30 representative sessions per cluster. We retain six labels per mode and sample 20 seed sessions from each. The conceptual-suite labels are `custom_llm_provider_setup`, `prompt_optimization_workflows`, `rag_and_retrieval_pipelines`, `react_agents_and_tools`, `gepa_optimizer_usage`, and `core_modules_signatures_and_program_structure`; the coding-suite labels share four of these and substitute `signature_schema_and_pydantic_types` and `evaluation_metrics_and_custom_eval` for the two non-coding-friendly labels.

Stage 3a: candidate generation. For each label, gpt-5.4 at reasoning effort `xhigh`, running inside Codex (OpenAI, 2025) with read access to the full repository and documentation, proposes 12 candidate (question, gold answer, `code_evidence`) triples. The generator is conditioned on the 20 sampled seed sessions for the label, the label description, the library description, and the per-task naming discipline. The full template is in §A.2.

Naming discipline. Both the generator and the critic enforce a strict *naming discipline* on the question text. The question may mention only brand-level concepts a real user would type (e.g., `dspy.LM`, `GEPA`, `Predict`), but must not name the internal class, method, helper, or file that constitutes the answer. The discipline is necessary because at test time the answering system has `glob_files`, `grep_code`, and `read_file` tools over $\mathbf{D}$; if the question leaks the locator, the question collapses into a one-grep lookup and stops measuring expertise. The critic rejects or rewrites questions that introduce internal symbols, file paths, or attachment-point phrasings such as "in the X tutorial / example / notebook". The gold answer is exempt from the discipline and is expected to name the precise symbols, files, and lines.

Stage 3b: critic selection. A second pass under the same model and harness acts as a critic. It sees the 12 candidates and the same seed sessions, selects five finalists per label, and may rewrite for compliance. The critic enforces naming discipline, validates that `code_evidence` cites real files under the configured code roots and matching the configured `file_glob`, and in coding mode additionally enforces that the gold answer contains a fenced code block, is not a stub or one-liner, and composes multiple public APIs. The full template is in §A.3.

Stage 4: private rubric construction. For each kept item, gpt-5.4 converts the gold answer into 2–8 atomic grading claims. Each claim is typed as `core` or `supporting` and cited to specific line spans in the evidence files (numbered file dumps are passed to the rubric builder). Claim weights sum to exactly 100, with most weight carried by `core` claims. Spans are constrained to 1–300 lines (preferred 1–40). The (claim rubric, evidence spans, gold answer) triple is the *private grading bundle*; only the question text is public. The full template is in §A.4.

Stage 5: human verification. Every kept item is then reviewed by a senior DSPy user before entering the final suite. The reviewer confirms three properties: (i) the gold answer is correct against the live source at the fixed commit; (ii) the cited evidence spans actually support each rubric claim; (iii) the question respects naming discipline and is not solvable by a single grep on a leaked symbol. Items failing any check are either rewritten by the reviewer or returned to the critic for re-selection. The human gate is what lets us claim that the benchmark measures expertise rather than the model-author's biases; the rest of the pipeline is what makes the gate tractable, since the reviewer reads pre-graded candidates rather than authoring from scratch.

Test-time harness. At test time the answering system Σ has $\mathbf{A} = \mathbf{D}$ restricted to the source and test roots, $\mathbf{T} = \{\texttt{glob_files}, \texttt{grep_code}, \texttt{read_file}\}$ scoped to those roots (per-call output truncated to 20,000 characters), and $\mathbf{H}$ varied across four configurations of fixed budget: `direct` (no tools), `react-5`, `react-20`, and `react-20-enforced` ($\geq$ 20 non-finish tool calls before the model is allowed to halt) (Yao et al., 2022). Coding answers must wrap code in a fenced ` ```python ` block; un-fenced answers are scored zero before the judge is invoked. A separate gpt-5.4 invocation grades each (question, model answer) pair against the private rubric, scoring each claim in $\{0, 0.5, 1\}$ and returning the weighted sum on $[0, 100]$. The full grader template is in §A.5.

A.2. Generator prompt

The conceptual-mode generator template is rendered with task-specific values substituted for the bracketed placeholders. The DSPy values are in §A.6. The coding-mode variant is structurally identical, with the "Required quality bar" and "Hard bans" sections replaced by a buildable-spec contract.

```
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} codebase and tests,
↪ but **not** the documentation.
- 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 will 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

The conceptual-mode critic template. The coding-mode variant adds a short "Coding-mode additional bars" section that requires the gold answer to be a non-stub fenced code block composing multiple public APIs.

```
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.
- The evaluation-time answering agent will have access to the {library_name} codebase and tests, but **not** the
↪ documentation.
- Treat docs as 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 doc-dependent, tutorial-dependent, or 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.
```

550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604

```
- `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}`
- Difficulty: `{difficulty}`
- 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.
```

605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659

```
- 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. Library-agnostic instantiation

The pipeline is library-agnostic. The generator, critic, rubric-builder, and grader prompts above contain only generic contracts about question quality, naming discipline, evidence, and grading; they do not mention DSPy. Porting the pipeline to any other library with a buildable code root and a stream of community-question text reduces to filling in a small adapter, which we call the *library profile*. Once the profile is specified, the rest of the pipeline runs unchanged.

A library profile consists of seven fields. The schema below gives the field name, its role, and the per-task value used for DSPy QA in this paper. Producing the analogous values for a new library is the only library-specific work the pipeline requires.

`library_name`
The display name used wherever the prompts mention the library.

```
DSPy
```

`library_description`
One-paragraph summary plus a layout map of the code roots the answering agent will see at test time. Used to orient the generator and critic at generation time.

```
DSPy is a Python framework for programming with language models using composable, self-optimizing modules. The
↪ answering agent has access to the DSPy codebase and tests, but not the documentation.

Repo layout:
  dspy/          core library --- predict/, retrieve/, teleprompt/, evaluate/, clients/, primitives/,
↪ signatures/, utils/
  tests/         unit and integration tests
  docs/          documentation (not visible to answering agent)
```

`ok_to_name`
Brand-level concepts a real user would type. Naming these in the question is allowed because they appear in user sessions and documentation.

```
- The library (`dspy`) and top-level tutorial primitives (`Signature`, `Module`, `Predict`, `LM`,
↪ `dspy.configure`, `dspy.settings`)
- Branded, user-facing optimizers and utilities that appear in docs and user sessions (e.g. `GEPA`, `MIPRO`,
↪ `BootstrapFewShot`, `Refine`, `BestOfN`) --- naming them as a concept is fine, because users do
- Anything the community user in the seed session typed first, at the same granularity they typed it
```

660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714

not_ok_examples

Library-specific cases of locator-leaking phrasings. The generic naming-discipline rules are encoded in the generator and critic prompts; this field supplies concrete bad patterns for the target library.

```
- The method or attribute on a branded class that contains the answer (`BootstrapFewShot.compile`,
↪ `GEPA.propose`, `LM.__call__`) --- name the behavior, not the method
- Internal adapter / predictor / parser classes (`JSONAdapter`, `TwoStepAdapter`, `ChatAdapter`,
↪ `ChainOfThoughtWithHint`, etc.)
- Internal helper functions, `snake_case` function names with dot-paths, private config flags, file paths,
↪ test-file names
```

bad_question_examples

Concrete examples of questions the critic should reject or rewrite, drawn from the target library's symbol surface.

```
- "When does `JSONAdapter` send a structured response schema vs `{type: json_object}`?"
- "What does `dspy.Refine` add beyond `dspy.BestOfN`? (names both internal sides --- reframe as a user goal)"
- "Does `TwoStepAdapter` reuse the globally configured adapter for its extraction pass?"
- "How does `BootstrapFewShot.compile` decide when to stop bootstrapping?"
```

good_question_examples

Concrete examples of well-formed questions the critic should accept, demonstrating the locator-hard, behavior-anchored shape the pipeline targets.

```
- "When my signature uses a Pydantic model as the output type, how does DSPy decide whether to ask the provider
↪ for strict JSON schema enforcement, a loose JSON object, or plain-text parsing --- and where in the code is
↪ that decision made?"
- "If I want retries that feed the previous failure back into the next attempt as guidance (rather than just
↪ re-rolling), what's available in DSPy, and how does the feedback actually get surfaced to the LM?"
- "When an adapter needs a second LM call to extract structured fields from a free-form answer, does that second
↪ call inherit whatever adapter I configured globally, or is it pinned?"
- "Using `BootstrapFewShot`, what actually determines when bootstrapping stops --- a fixed budget, a success
↪ criterion, or something else --- and where is that controlled?"
```

code_roots, file_glob, evidence_path_prefixes

Filesystem scope of the corpus at generation and test time.

`repo_code_roots` are `dspy/` and `tests/`; the answering harness's `file_glob` is `*.py`; `evidence_path_prefixes` restrict citations to `dspy/` and `tests/`. Per-call tool output is truncated to 20,000 characters.

A.7. Worked examples

The three examples below are drawn from the final, human-verified suite. Each shows the public question text alongside the private grading bundle (gold answer, evidence files, claim rubric) and a locator-leak audit confirming that the question text alone does not give away the answer's location in the source.

Example 1: conceptual mode (very-hard)

ID `dspy_00130f02bbbb` LABEL `custom_llm_provider_setup` DIFFICULTY `very_hard`

QUESTION

I'm wiring a private OpenAI-compatible endpoint and I'm unsure whether the model should be named like `my-model`, `openai/my-model`, or `groq/my-model`. Inside DSPy, what actually changes when I change that prefix or omit it?

GOLD ANSWER The prefix matters in a few different code paths. DSPy treats the text before the first / as the provider name, and if there is no slash it defaults to `openai`; that provider name is used for response-schema and supported-parameter lookups, and

on the text-completion path it is also used to derive <provider>_API_KEY and <provider>_API_BASE. Function-calling and reasoning checks still use the full model string. Separately, provider inference only special-cases openai/... and ft:; other names stay on the generic provider unless you pass one explicitly, which mainly affects provider-managed operations like launch, kill, and finetune, not normal generation.

EVIDENCE FILES dspy/clients/lm.py, dspy/clients/openai.py.

CLAIM RUBRIC

- **c1** (core, weight 30, span s1 = dspy/clients/lm.py:115-121).
  DSPy parses the provider as the text before the first / in the model string, and if there is no slash it defaults that provider name to openai.
- **c2** (core, weight 20, span s2 = dspy/clients/lm.py:130-137).
  That parsed provider name is what DSPy passes into LiteLLM for response-schema support checks and supported-parameter lookup.
- **c3** (core, weight 20, span s3 = dspy/clients/lm.py:403-430).
  On the text-completion path, the parsed/defaulted provider name is also used to choose provider-specific environment variables named <provider>_API_KEY and <provider>_API_BASE, while the completion model sent becomes text-completion-openai/<model-name>.
- **c4** (core, weight 15, span s4 = dspy/clients/lm.py:122-128).
  Function-calling support and reasoning support are checked against the full model string, not the parsed provider prefix alone.
- **c5** (core, weight 10, spans s5 = dspy/clients/openai.py:48-55 and s6 = dspy/clients/lm.py:307-310).
  Provider inference only recognizes models starting with openai/ or ft: as OpenAI; otherwise LM.infer_provider() leaves the model on the generic Provider unless a provider is passed explicitly.
- **c6** (supporting, weight 5, spans s6 = dspy/clients/lm.py:307-310 and s7 = dspy/clients/lm.py:241-277).
  The inferred or explicit Provider mainly affects provider-managed operations such as launch, kill, and finetune rather than ordinary generation calls.

LOCATOR-LEAK AUDIT The question text mentions only the user-facing model-name fragments (my-model, openai/my-model, groq/my-model) and the user-facing concept of a "private OpenAI-compatible endpoint". It names no class, method, helper, or file. The answering system must locate dspy/clients/lm.py, dspy/clients/openai.py, and the relevant line ranges from the question's behavioral framing alone.

Example 2: conceptual mode (very-hard)

ID dspy_b132c197d5fa   LABEL prompt_optimization_workflows   DIFFICULTY very_hard

QUESTION

> *If I want MIPROv2 to do prompt-only search with no few-shot examples in the deployed prompt, what setting combination actually enforces that, and does DSPy still generate demos internally anyway?*

GOLD ANSWER Set both max_bootstrapped_demos=0 and max_labeled_demos=0. That makes MIPROv2 zero-shot at search time: it still runs the demo-generation step first, but before prompt-parameter search it discards the demo candidates, so the compiled program ends without demos. In that zero-shot branch, the internal proposal context still uses up to 3 bootstrapped demos and 0 labeled demos. In auto mode, zero-shot runs also use the full auto candidate count for instruction proposals instead of halving it.

EVIDENCE FILES dspy/teleprompt/mipro_optimizer_v2.py, dspy/teleprompt/utils.py.

CLAIM RUBRIC

- **c1** (core, weight 35, span s1).
  MIPROv2 enters its zero-shot optimization path only when both max_bootstrapped_demos and max_labeled_demos are set to 0.
- **c2** (core, weight 30, spans s2, s5).
  Even in that zero-shot configuration, MIPROv2 still runs the few-shot demo bootstrapping step before instruction proposal and prompt optimization.

770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824

- **c3** (`core`, weight 20, spans s2, s3).
  After generating instruction candidates, the zero-shot path discards `demo_candidates`, so the final prompt-parameter search and resulting compiled program do not include demos.
- **c4** (`supporting`, weight 10, span s4).
  In the zero-shot branch, internal demo generation uses 3 bootstrapped demos in context and 0 labeled demos.
- **c5** (`supporting`, weight 5, span s6).
  When `auto` mode is enabled, zero-shot runs use the full auto `n` for instruction candidates, whereas non-zero-shot runs halve that instruction-candidate count.

LOCATOR-LEAK AUDIT The question names only the brand-level concept `MIPROv2` and the brand-level user goal ("prompt-only search with no few-shot examples in the deployed prompt"). It does not name any internal flag, helper, or file path. The answering system must reach the zero-shot branch in `mipro_optimizer_v2.py`, observe the demo-bootstrapping precondition, and read the instruction-proposal helpers in `utils.py` to recover the rubric content.

Example 3: coding mode (hard)

ID `dspy_db6c58a4b861` LABEL `gepa_optimizer_usage` DIFFICULTY `hard`

QUESTION

> *I'm debugging a tiny GEPA run on a one-step QA module. Build a harness that runs a small optimization, uses separate task and reflection LMs, and prints the exact reflection-model request, the raw reflection-model response, the parsed reflection output, and the final instruction text on the optimized predictor.*

GOLD REFERENCE IMPLEMENTATION

```
1  import json
2  import dspy
3  from dspy.utils.dummies import DummyLM
4
5
6  class QAProgram(dspy.Module):
7      def __init__(self):
8          super().__init__()
9          self.predictor = dspy.Predict('question -> answer')
10
11     def forward(self, question):
12         return self.predictor(question=question)
13
14
15 def metric(gold, pred, trace=None, pred_name=None, pred_trace=None):
16     score = float(pred.answer.strip().lower() == gold.answer.lower())
17     feedback = 'Return the exact short answer and nothing else.' if not score else 'Good.'
18     return dspy.Prediction(score=score, feedback=feedback)
19
20
21 task_lm = DummyLM({'capital of France': {'answer': 'Lyon'}})
22 reflection_lm = DummyLM([
23     {'improved_instruction': 'Return the exact short answer and nothing else.'},
24 ] * 8)
25
26 dspy.configure(lm=task_lm)
27 trainset = [
28     dspy.Example(question='What is the capital of France?', answer='Paris').with_inputs('question'),
29 ]
30
31 optimized = dspy.GEPA(
32     metric=metric,
33     reflection_lm=reflection_lm,
34     max_metric_calls=4,
35     seed=0,
36 ).compile(QAProgram(), trainset=trainset, valset=trainset)
37
38 last_reflection = reflection_lm.history[-1]
39 print(json.dumps({
```

825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879

```
40     'reflection_request': last_reflection['messages'],
41     'reflection_raw_response': last_reflection['response'],
42     'reflection_outputs': last_reflection['outputs'],
43     'optimized_instruction': optimized.predictor.signature.instructions,
44 }, indent=2, default=str))
```

EVIDENCE FILES `dspy/teleprompt/gepa/gepa.py`, `dspy/clients/base_lm.py`, `dspy/utils/dummies.py`.

CLAIM RUBRIC

- **c1** (`core`, weight 28, spans s1, s2).
  The harness must instantiate `GEPA` with a separate `reflection_lm`, and provide exactly one budget control such as `max_metric_calls` plus an optional `seed`, before calling `.compile(...)` on the QA module.
- **c2** (`core`, weight 24, spans s2, s6).
  The optimization setup must use a task LM distinct from the reflection LM by configuring the default DSPy LM for task execution and passing the reflection model separately to GEPA.
- **c3** (`core`, weight 20, span s3).
  The metric supplied to GEPA must accept five arguments `(gold, pred, trace, pred_name, pred_trace)` and may return score-with-feedback, enabling reflection feedback during optimization.
- **c4** (`core`, weight 18, spans s4, s5).
  To inspect the reflection-model interaction, the harness should read the final entry from `reflection_lm.history` and use its recorded `messages`, raw `response`, and processed `outputs`.
- **c5** (`supporting`, weight 10, span s7).
  The harness should print the optimized predictor instruction from `optimized.predictor.signature.instructions`, which GEPA seeds from predictor instructions and replaces with the best candidate after compile.

LOCATOR-LEAK AUDIT The question text mentions only the brand-level concept `GEPA`, the brand-level user goal ("runs a small optimization, uses separate task and reflection LMs"), and the four required artifacts (request, raw response, parsed output, final instruction). It does not name `dspy.Predict`, `DummyLM`, the `(gold, pred, trace, pred_name, pred_trace)` metric signature, the `history` attribute, or `signature.instructions`. The answering system must compose all five from the source.

B. Methods: Hyperparameters and Training Recipes

This appendix records the method recipes used for the Qwen3.5-4B DSPy matrix in Table 1. All parameter-updating methods are served as LoRA modules on top of the same `Qwen/Qwen3.5-4B` base model, and parameters can be found in Table 3

At evaluation time, all Qwen3.5-4B cells use temperature 0.6, top-$p = 0.95$, top-$k = 20$, min-$p = 0$, presence penalty 0, repetition penalty 1.0, and maximum generation length 32768. Coding questions receive an additional suffix telling the model that it may use `grep_code`, `read_file`, and `glob_files`, and requiring the final answer to be executable Python in a fenced `python` block.

C. Grading Pipeline

The grader is a separate GPT-5.4 structured-output call. For each model answer, it scores every claim as 0, 0.5, or 1 and emits a confidence score plus a `needs_regrade` flag. The reported question score is not the judge's free-form total; it is recomputed as the weighted sum of claim scores, and the pipeline records the mismatch between this computed score and the judge-reported score. Coding answers without a complete fenced code block are auto-scored zero before the judge call.

Aggregation uses three rollouts per question. For each rollout, the pipeline averages over all selected questions; the point estimate is the mean of rollout-level means, and the standard error is the sample standard deviation over rollout-level means divided by $\sqrt{3}$. Per-label statistics use the same rollout-level convention. The current result files also record mean confidence, number of `needs_regrade` flags, and maximum score mismatch for each cell. A second-judge agreement study has not yet been run for the reported matrix, so this draft does not claim inter-judge agreement.

880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934

Table 3. Study-procedure recipes for the reported DSPy runs.

Procedure	Study material	Recipe
Bare	none	No pre-evaluation transformation. Test-time harness is either direct prediction or ReAct with the repository tools.
CPT-code	237 files, 426k included tokens; 111 packed sequences	LoRA rank 16, alpha 32, dropout 0.05; targets `q,k,v,o,up,down,gate` projections; seq len 4096; LR 10^{-4}; batch size 1, grad accumulation 8
CPT-docs	32 markdown files, 165,922 included tokens; 40 packed sequences	Same LoRA and optimizer recipe as CPT-code;
SFT / context distillation	25,198 DSPy traces plus 2,508 OpenThoughts-style anchor traces replayed six times	We adopted the hyperparameters from `tinker_cookbook` (Thinking Machines Lab, 2025). LoRA rank 32, alpha 32, dropout 0; max sequence length 8192; LR 4.9×10^{-4}; AdamW fused, warmup ratio 0.03, weight decay 0; batch size 2, grad accumulation 8, four DDP processes.
Scout self-built	DSPy source/tests through the same three tools as evaluation	Qwen3.5-4B runs a forced ReAct study loop with at least 50 non-finish tool calls and writes a cheatsheet. The $K = 50$ artifact contains a 13,216-character cheatsheet, 56 tool steps, 44,195 completion tokens, 2.12M prompt tokens.
Scout GPT-5.4-built	Same as Scout self-built	Same scaffold, but the studying model is GPT-5.4. The $K = 50$ artifact contains a 34,811-character cheatsheet, 52 tool steps, 45,755 completion tokens, 2.88M prompt tokens.

D. Additional Results

This appendix reports the remaining result sweeps in the same format as Table 1. All scores are on the same 0–100 grading scale. To keep the appendix compact, we report means rounded to one decimal place and omit standard errors. Overall, the additional results support the main empirical pattern: at a fixed inference budget, the tested weight-side study procedures move scores only modestly, while stronger models and forced tool use can substantially raise absolute coding performance.

The larger-model coding results show the same qualitative accounting issue from a higher absolute-performance regime. GPT-5.4 already obtains substantially stronger direct coding performance than the smaller open models, and Scout further improves every reported GPT-5.4 inference budget. The forced-compute cells are also informative: requiring the agent to spend the full tool budget can improve coding performance substantially, suggesting that early stopping or insufficient search can leave useful repository evidence unused.

935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989

Table 4. Qwen3 on DSPy QA. Scores are means rounded to one decimal. The conceptual suite is the same suite labeled "trivia" in the raw result files. ReAct-5 and ReAct-20 allow up to 5 or 20 repository-tool iterations with early stopping.

	Code			**Conceptual**		
Method	*direct*	ReAct-5	ReAct-20	*direct*	ReAct-5	ReAct-20
Bare Qwen3	6.7	5.2	6.0	22.6	23.4	22.7
Continued pretraining						
CPT, 1 epoch	6.0	6.0	5.7	24.9	20.4	22.1
CPT, 3 epochs	7.9	5.0	4.6	22.3	24.7	22.1
CPT, 5 epochs	7.1	5.9	5.0	23.1	22.5	21.8
CPT, 10 epochs	8.8	4.6	4.1	25.7	24.2	24.7
On-policy context distillation						
SFT, 1 epoch	8.5	4.8	3.8	21.1	21.3	20.8
SFT, 2 epochs	8.9	4.7	4.6	28.4	21.0	24.9
SFT, 3 epochs	9.7	4.6	3.8	24.7	22.3	27.6

Table 5. Additional coding-only results for larger or stronger answering models. Scores are means on the 0–100 coding scale. ReAct-20-enforced uses the same repository tools as ReAct-20 but requires 20 tool calls before the final answer.

Method	*direct*	ReAct-5	ReAct-20	ReAct-20-enforced
GPT-5.4 bare	32.9	51.2	50.9	—
GPT-5.4 + Scout	46.8	57.6	63.4	71.9
Gemma4-26B-A4B-it bare	23.4	46.2	44.2	57.2