LEARNING, FAST AND SLOW: TOWARDS LLMS THAT ADAPT CONTINUALLY

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are trained for downstream tasks by updating their parameters (e.g., via RL). However, updating model parameters forces them to absorb task-specific information, which can result in catastrophic forgetting and loss of plasticity. In contrast, in-context learning with fixed LLM parameters can cheaply and rapidly *adapt* to task-specific requirements (e.g., prompt optimization), but cannot by itself typically match the performance gains available through updating LLM parameters. Inspired by how human learning works at multiple time scales (e.g., System 1 vs 2), we propose a fast-slow learning framework for LLMs, instantiated with model parameters as "slow" weights and optimized context as "fast" weights. These fast "weights" can learn from textual feedback to absorb the task-specific information, while allowing slow weights to stay closer to the base model and persist general reasoning behaviors. **Fast-Slow Training** (FST) is up to $3\times$ more sample-efficient than only slow learning (RL) across reasoning tasks, while consistently reaching a higher performance asymptote. Moreover, FST-trained models remain closer to the base LLM (up to 70% less KL divergence), resulting in less catastrophic forgetting than RL-training. This reduced drift also preserves plasticity: after training on one task, FST trained models adapt more effectively to a subsequent task. In continual learning scenarios, where task domains change on the fly, FST continues to acquire each new task while RL stalls.

1 INTRODUCTION

Large language models (LLMs) are commonly adapted through supervised finetuning (SFT) or reinforcement learning (RL), both of which modify the model parameters, to specialized domains such as math and coding (Guo et al., 2025; Lambert et al., 2024; Shao et al., 2024; Guo et al., 2024a; Ouyang et al., 2022). However, treating model parameter updates as the sole mechanism of adaptation creates a fundamental bottleneck: every improvement, whether it be a reusable reasoning skill, a task-specific heuristic or a transient lesson from recent rollouts, must be written into the same persistent set of model weights. Since the entire policy is parameterized by these weights, an update that improves in-domain reward simultaneously moves the model away from its base behavior (Gao et al., 2023; Lin et al., 2024), reducing entropy (Cui et al., 2025; Li et al., 2026), hurting out-of-distribution generalization (Ouyang et al., 2022; Kalajdzievski, 2024; Luo et al., 2025; Phan et al., 2026), and degrading its ability to adapt to future tasks, known as plasticity loss (Frati et al., 2024; Lyle et al., 2022; Dohare et al., 2024; Tang et al., 2025).

LLM systems also possess another powerful adaptation mechanism: prompts, instructions, and contextual information (Brown et al., 2020; Wei et al., 2022). Unlike model parameters, these textual components can be modified cheaply, frequently, and per task. Prompt optimization methods demonstrate that substantial behavioral improvements can be obtained by improving the textual context under which the model operates (Zhou et al., 2022; Yang et al., 2024; Khattab et al., 2023; Opsahl-Ong et al., 2024; Agrawal et al., 2026).

We introduce **Fast-Slow Training (FST)**, a general framework that interleaves optimization of model parameters (*slow weights*) and adaptable textual components (*fast weights*) as shown in Figure 2. Fast weights can include prompts, context, tool definitions, or the text and code defining an agent harness, including its control flow. Slow weights can be updated through RL, SFT, or preference optimization. For the scope of this paper, we focus on an RL + GEPA instantiation, rather than all these possibilities.

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

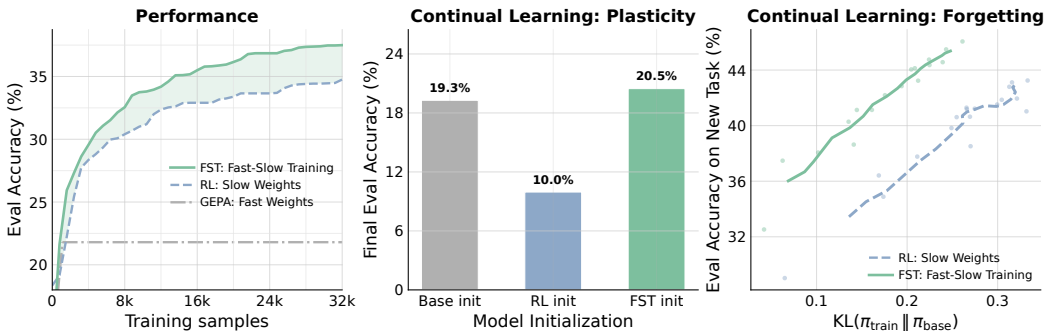

Figure 1: **Fast-Slow Training (FST) learns faster, stays adaptable, and forgets less.** Comparison of FST (slow-weight RL interleaved with fast-weight prompt optimization), RL alone (slow weights only), and GEPA alone (fast weights only), averaged over `CodeIO`, `Math (Polaris)`, and `HoVer-hard`. **Left:** Evaluation accuracy on the trained task as training samples accumulate. FST reaches RL's peak with substantially fewer samples and converges to a higher ceiling than either RL or GEPA alone. **Middle:** Plasticity, the model's remaining ability to learn a new skill. After training on a first task, we continue with a fresh round of RL on a second task and report the final accuracy from each initialization. The RL-trained checkpoint barely learns the new task, while the FST-trained checkpoint roughly matches the base model. **Right:** How far each training run drifts from the base model, measured by $\mathrm{KL}(\pi_{\text{train}} \,\|\, \pi_{\text{base}})$. Smaller drift correlates with less forgetting of prior abilities; at matched accuracy, FST sits well to the left of RL.

The distinction between fast and slow-moving weights has a long history in neural networks (Hinton & Plaut, 1987; Schmidhuber, 1992; Ba et al., 2016). A common motivation across this line of work is to separate different forms of learning: temporary, task-specific, or lightweight adaptations can be captured by fast weights, while more persistent and broadly useful behaviors are consolidated into slow weights. Motivated by this perspective, we adopt a fast-slow weight view of LLM post-training. In our setting, model parameters naturally serve as slow weights: they are expensive to update, persist across tasks, and encode long-lived behavior. In contrast, optimized textual context, such as prompts, instructions, and task-specific feedback, acts as a form of fast weight: it can be changed cheaply and frequently, influences model behavior immediately, and captures task-level adaptation without permanently modifying the model.

This division of labor has several consequences, which we evaluate in RL settings spanning math, code, general reasoning and non-verifiable rubric-reward tasks.

1. **Fast textual adaptation improves data efficiency.** Fast-slow training matches RL reward with up to $3\times$ fewer rollouts and consistently reaches a higher performance ceiling (Section 4 Advantages 1 and 2).

2. **Fast-slow training induces smaller slow-weight displacement.** With the textual channel carrying part of the adaptation, the model parameters need not move as far from the base policy. At matched reward, our models have up to 70% lower KL to the base policy than RL-only baselines. (Section 4 Advantage 3)

3. **Fast-slow training preserves plasticity.** After training on one task, FST models adapt effectively to a second task, while RL-only models collapse to near 0%, suggesting greater capacity for future learning with FST (Section 4 Advantage 4).

4. **Fast-slow training enables continual learning.** FST adapts more quickly to changing objectives when tasks change during training (Section 4 Advantage 5).

Our results support jointly training fast textual and slow parametric weights, rather than learning model parameters and then tuning context, to improve task performance while preserving generality and plasticity.

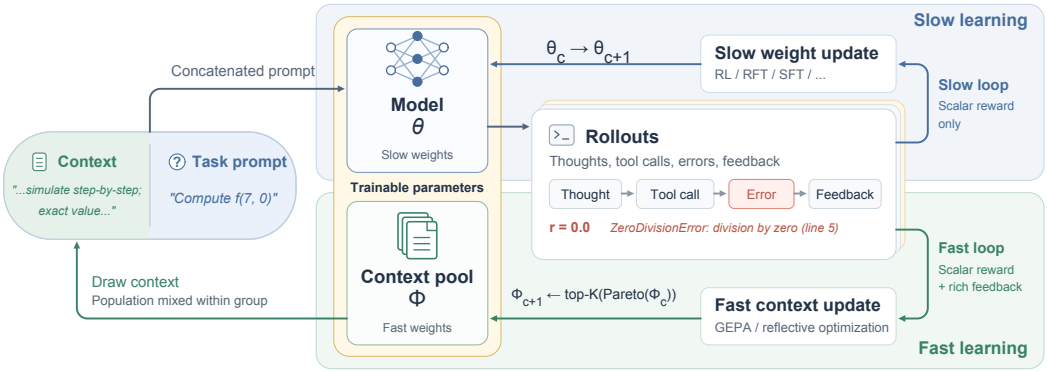

Figure 2: **Slow weights and fast weights co-evolve through interleaved updates.** The slow loop (top) updates θ from the scalar reward alone ($\theta_c \to \theta_{c+1}$). The fast loop (bottom) updates Φ via reflective optimization, additionally consuming the rollout's full text including thoughts, tool calls, errors, and rich feedback ($\Phi_c \to \Phi_{c+1}$). Maintaining Φ as a Pareto-frontier population (rather than a single best prompt) preserves diversity: different contexts specialize to different problem slices exposing the slow update rule to rich conditioning during training.

2 PRELIMINARIES

Fast and slow weights: a general framework We model *slow weights* (model parameters) as θ and *fast weights* (adaptable text or code) as $\phi \in \Sigma^*$. Optimizing ϕ can mean prompt or context optimization, tool optimization, or full harness optimization when the harness is represented as text or code. Given a query x, the system produces a response by sampling

$$y \sim \pi_\theta(\cdot \mid x, \phi), \tag{1}$$

where $\pi_\theta(y \mid x, \phi)$ denotes the response distribution of the system defined by θ and ϕ on query x. For a task distribution $\mathcal{D}$ and reward r, the natural joint objective is

$$\max_{\theta,\,\phi} \; J(\theta, \phi) \;=\; \mathbb{E}_{x\sim\mathcal{D},\; y\sim\pi_\theta(\cdot|x,\phi)}\big[\, r(x, y) \,\big]. \tag{2}$$

The framework is optimizer-agnostic. Slow weights can use SFT, preference optimization Rafailov et al. (2023), or policy-gradient methods such as PPO Schulman et al. (2017) and GRPO Shao et al. (2024), often with verifiable rewards Lambert et al. (2024). Fast weights can use prompt/harness optimizers such as APE Zhou et al. (2022), OPRO Yang et al. (2024), DSPy/MIPROv2 Khattab et al. (2023); Opsahl-Ong et al. (2024), Meta-Harness Lee et al. (2026), and GEPA Agrawal et al. (2026). We instantiate FST with RL with verifiable rewards (RLVR) and GEPA.

Slow weights: RL with verifiable rewards We follow ScaleRL Khatri et al. (2026), using verifier rewards $r(x, y) \in [0, 1]$ Lambert et al. (2024), such as rule-based correctness for math, code, and science. For each query x, the current policy generates G rollouts $\{y_i\}_{i=1}^{G}$ under (θ, ϕ), with group-relative advantages Shao et al. (2024)

$$A_i \;=\; \frac{r(x, y_i) \;-\; \bar{r}_g}{\sigma_g \;+\; \varepsilon}, \qquad \bar{r}_g \;=\; \tfrac{1}{G}\sum_{j=1}^{G} r(x, y_j), \qquad \sigma_g^2 \;=\; \tfrac{1}{G}\sum_{j=1}^{G}\big(r(x, y_j) - \bar{r}_g\big)^2, \tag{3}$$

and normalized at the batch level. The policy is updated using the truncated importance-sampling REINFORCE objective CISPO MiniMax (2025); Khatri et al. (2026),

$$\mathcal{L}_{\text{CISPO}}(\theta) \;=\; -\,\mathbb{E}\big[\,\mathrm{sg}\big(\min(\rho_t, \tau)\big) \;\cdot\; A \;\cdot\; \log \pi_\theta(y_t \mid x, \phi, y_{<t})\,\big], \tag{4}$$

where $\rho_t = \pi_\theta(y_t \mid x, \phi, y_{<t}) / \pi_{\theta_{\text{old}}}(y_t \mid x, \phi, y_{<t})$ is the per-token importance ratio between the current and rollout policies, τ the truncation threshold, and $\mathrm{sg}(\cdot)$ the stop-gradient operator. Loss is aggregated at the prompt level. Conventional RLVR updates only θ, keeping ϕ fixed to a generic system prompt.

Fast weights: reflective prompt evolution. We optimize the fast weights ϕ using GEPA Agrawal et al. (2026), a reflective evolutionary procedure over textual prompts $\phi \in \Sigma^*$. For a fixed policy π_θ, the fitness of a prompt on instance x is its expected reward,

$$s(\phi; x) \;=\; \mathbb{E}_{y\sim\pi_\theta(\cdot|x,\phi)}[r(x,y)]\,. \tag{5}$$

GEPA maintains a population of prompts, uses rollouts to elicit natural-language critiques from a frozen reflection LM, and proposes textual mutations that improve performance on an anchor set from $\mathcal{D}$. Rather than returning a single prompt, GEPA retains a Pareto frontier of complementary prompts and returns the top-m candidates, which we use as fast weights. We defer the details of parent selection, mutation, pruning, and prompt examples to Appendix G.1.

3 FAST-SLOW TRAINING (FST)

We now describe the RL + GEPA instantiation of FST. It updates slow weights θ through RL and a population of fast weights Φ through GEPA. This instantiation maintains K textual prompts, $\Phi = \{\phi^{(1)}, \ldots, \phi^{(K)}\}$, and optimizes

$$\max_{\theta,\,\Phi}\;\; J(\theta,\Phi) = \mathbb{E}_{x\sim\mathcal{D},\;\phi\sim U(\Phi),\;y\sim\pi_\theta(\cdot|x,\phi)}[r(x,y)]\,, \tag{6}$$

where $U(\Phi)$ is uniform over the prompt population. We keep a population rather than one best prompt (examples of evolved prompts are in Appendix D.4). GEPA returns a Pareto frontier of complementary prompts: different prompts excel on different subsets of $\mathcal{D}$. Sampling across this frontier during RL gives the policy access to multiple conditioning behaviors and lets group-relative advantages compare both prompt-induced and sampling-induced variation on the same problem.

Training proceeds in cycles of T slow-weight updates. At the start of cycle c, we pre-fetch the next T RL batches and denote their union by the lookahead batch $\mathcal{L}_c$. We run GEPA with the current policy π_{θ_c} as the rollout model, a frozen reflection LM π_{ref} as the proposer, $\mathcal{L}_c$ or a fixed-size subset as the anchor set, and the previous population Φ_c as the seed. GEPA returns the top-K candidates from its Pareto frontier, yielding the fast weights Φ_{c+1}.

For the next T steps, we update θ on minibatches from $\mathcal{L}_c$ while holding Φ_{c+1} fixed. For each problem p, we form a rollout group of size G by sampling each prompt $\phi^{(k)} \in \Phi_{c+1}$ exactly G/K times. That is, in each group, G/K rollouts receive the same prompts and we have K such mini-groups. Cumulatively, they are treated as one group for p; rewards are normalized by the per-problem statistics $(\bar{r}_g, \sigma_g)$ as in eq 3, mixing prompt and sampling variation within the same advantage computation. We then apply the CISPO update in Eq. (4). After T updates, the procedure repeats with a new GEPA phase under the updated policy. Pseudocode of FST is given in Appendix G.2.

4 ADVANTAGES OF FAST-SLOW TRAINING

The textual fast weights Φ carry part of the task-level information that RL would otherwise force into θ, so the slow weights move less to reach the same reward. The downstream signature of this division of labor is consistent across our settings: training reaches matched reward more quickly, θ drifts less from the base policy at convergence, the model retains greater plasticity to adapt to subsequent tasks, and our method shows higher continual learning capability. We show each in the following sections.

Advantage 1: Fast-Slow Training Improves Data Efficiency We evaluate FST on three training families: code-output prediction (`CodeIO`) (Li et al., 2025b; Stojanovski et al., 2025), math (`Polaris`) (An et al., 2025), and multi-hop fact verification (`HoVer-hard`) (Jiang et al., 2020). All experiments use `Qwen3-8B` (Yang et al., 2025), except for the Math run, where we first SFT `Qwen3-8B-Base` on Nemotron data (Du et al., 2025) because `Qwen3-8B` is already saturated on math benchmarks (Qwen3-32B, Qwen3-4B, and OLMo-3-7B results are in Appendices D.2 and E.1). FST uses cycle length T=6 and $K \in \{4, 8\}$ candidate prompts per cycle. Training-time performance is measured on a held-out in-distribution validation set. RL is trained until step 1500 or in-distribution saturation (whichever comes first); FST is trained at least until it matches RL's running peak. Full hyperparameters and dataset details are deferred to Appendix G.4 and cost to Appendix C.1.

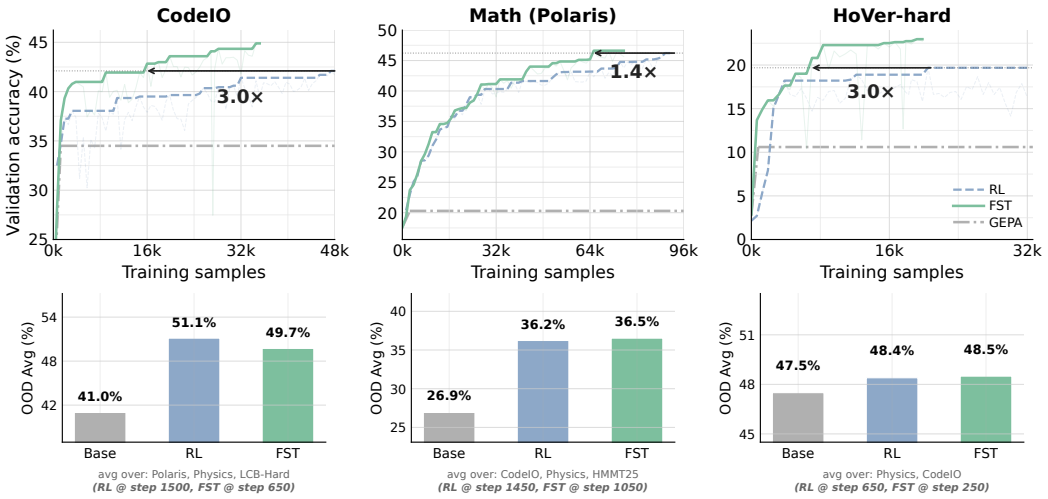

Figure 3: **Data efficiency across three training families. Top row**: matched-step validation accuracy (running max, mean@4); dash-dot GEPA-only reference rises from the step-0 base accuracy to the prompt-only ceiling within GEPA's inference budget. FST reaches RL's running peak in substantially fewer training steps ($3.0\times$ on `CodeIO`, $1.4\times$ on `Math` (Polaris), $3.0\times$ on `HoVer-hard`). **Bottom row**: out-of-distribution accuracy averaged across cross-domain (and easy→hard, where available) benchmarks for each family, evaluated with no GEPA prompt. FST matches RL on OOD averages while reaching the in-distribution peak with substantially fewer steps.

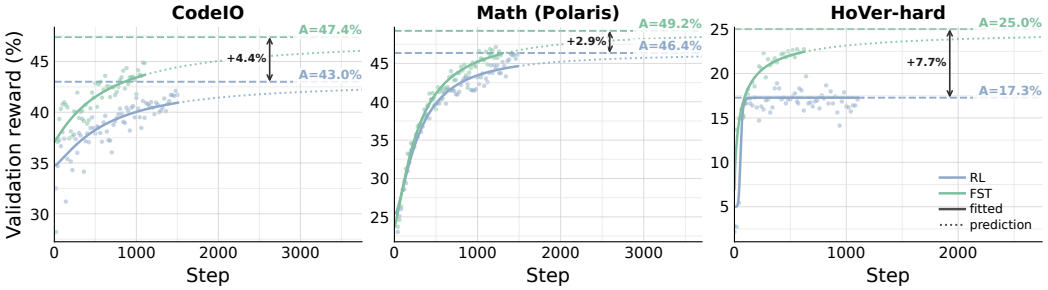

Figure 4: **Performance asymptote** on `CodeIO`, `Math` (Polaris), and `HoVer-hard`. For each run we fit a 4-parameter sigmoid $R - R_0 = \frac{A - R_0}{1 + (C_{mid}/C)^B}$ to the validation-accuracy trajectory and annotate the upper asymptote A. FST's asymptote (green) is higher than RL's (blue) on all three tasks. Solid curves cover the fit window; dotted curves are extrapolation past the last training step.

The matched-step training curves (Figure 3 Top) show that FST reaches RL's running peak in substantially fewer steps: $\mathbf{3.0}\times$ fewer on `CodeIO` , $\mathbf{1.4}\times$ on `Math` , and $\mathbf{3.0}\times$ on `HoVer-hard`. Continuing past the crossover, FST's running peak also exceeds RL's on all three tasks .

To check that the in-distribution data efficiency does not come at the cost of out-of-distribution behavior, for each training task we compare Base, RL's final checkpoint, and FST's validation matched-performance checkpoint on a family of cross-domain and easy-to-hard generalization datasets, with no GEPA prompt at inference (Figure 3 bottom). Across all three training families, FST reaches RL's in-distribution peak with $1.4\times$ to $3.0\times$ fewer training samples while retaining comparable OOD performance. Thus, FST's faster in-distribution learning is accompanied by preserved cross-domain and easy-to-hard generalization.

Advantage 2: Fast-Slow Training Raises the Performance Asymptote Following Khatri et al. (2026), we compare RL and FST by the saturation level of their validation-accuracy curves rather than at any single training step. Unlike final-step or matched-step accuracy, which depends on where

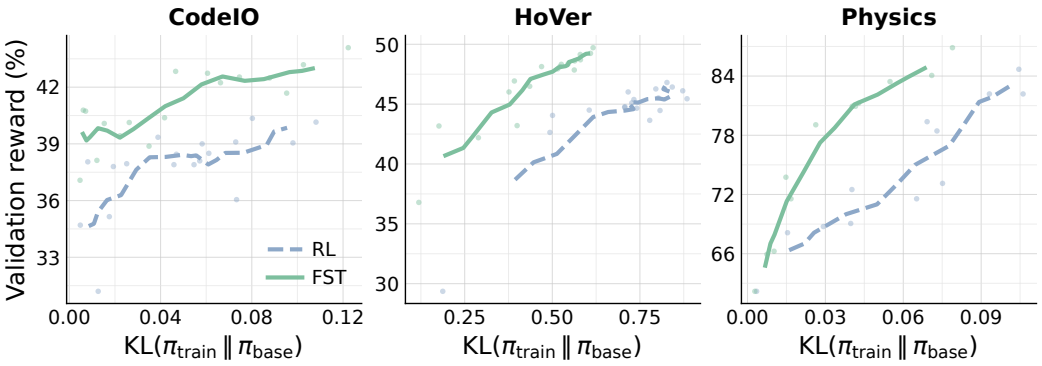

Figure 5: **Validation reward versus** $\mathrm{KL}(\pi_{\textbf{train}} \| \pi_{\textbf{base}})$ **trajectories** on `CodeIO`, `HoVer`, and `Physics`. Translucent markers are per-checkpoint measurements; the line is a rolling-mean smoothing along training step. At matched reward, FST (green) sits to the left of RL (blue) on every task, reaching the same reward at a significantly lower KL from the base policy. Full figure in Appendix D.1.

each run was stopped, the asymptote of a fitted curve reads off the level the run is converging to. For each (task, method) we fit a sigmoid curve

$$\Delta R = \frac{A - R_0}{1 + (C_{mid}/C)^B} \tag{7}$$

to the validation-accuracy trajectory, where A is the upper asymptote, B a scaling exponent, C_{mid} the midpoint of the performance, and R_0 is the initial reward at step 0.

Across all three tasks (Figure 4), FST's fitted asymptote exceeds RL's: A=47.4% vs 43.0% on `CodeIO` (+4.4pp), 49.2% vs 46.4% on `Math` (Polaris) (+2.9pp), and 25.0% vs 17.3% on `HoVer-hard` (+7.7pp). Pushing part of the task adaptation into the textual fast-weight channel Φ in addition to the slow weights θ helps the overall method converge to a higher accuracy ceiling than RL alone reaches; the same fit at Qwen3-32B is in Appendix D.2.

Advantage 3: Fast-Slow Training Remains Close to the Base Model The KL divergence $\mathrm{KL}(\pi_{\text{train}} \| \pi_{\text{base}})$ between the post-trained policy and the base measures how far the slow weights have moved away from their base configuration; larger displacement is associated with reduced entropy, weaker OOD generalization, and lower plasticity for future tasks (Frati et al., 2024; Lyle et al., 2022; Dohare et al., 2024; Tang et al., 2025). We track this directly - at each training checkpoint we compute token-level KL from the base on the held-out validation prompts and plot it against the same checkpoint's validation accuracy, for both FST and RL across `Physics`, `Math` (Polaris), `HoVer`, and `CodeIO`.

Across all three tasks (Figure 5), and on a rubric-scored task without a verifier (Appendix E.1), FST achieves higher performance at lower KL than RL. Shenfeld et al. (2026) recently showed that on-policy RL is already biased toward KL-minimal solutions on a new task, and that the size of this shift correlates with how much prior knowledge is forgotten. Even relative to this strong baseline, FST shifts the accuracy/KL frontier further left. We next show that models trained with FST preserve plasticity and learn better continually due to reduced displacement and continually adapted fast weights (Appendix F.1).

Advantage 4: Fast-Slow Training Preserves Plasticity Continued post-training has been observed to hamper a model's ability to learn future tasks, a phenomenon commonly called *plasticity loss* (Dohare et al., 2024; Lyle et al., 2022; Frati et al., 2024; Tang et al., 2025): the slow weights become specialized to the trained task and lose responsiveness to gradient signals from new ones. We probe this directly in two phases. *Phase 1* trains a base model on task X using either standard RL or FST. *Phase 2* takes the Phase-1 checkpoint, at the same training step for both arms, as initialization and runs standard RL on a different task Y. Throughout Phase 2 we track validation accuracy on

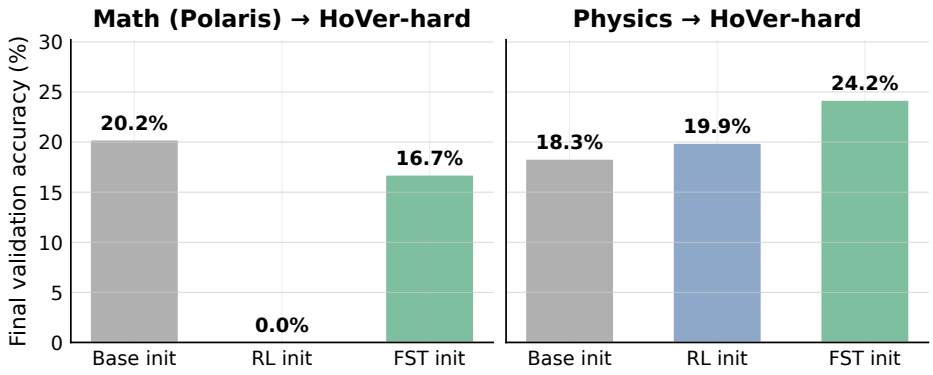

Figure 6: **Plasticity probe**: starting from a Math (left) or Physics (right) checkpoint trained with either RL or FST, we run a *fresh* RL pass on `HoVer-hard` and plot HoVer validation accuracy over 400 steps. Base init (dotted) is the no-prior-training reference. FST-init (green) preserves more capacity for the new task than RL-init (blue) on both arms; on the Math arm, prior RL collapses `HoVer-hard` learnability to near-zero.

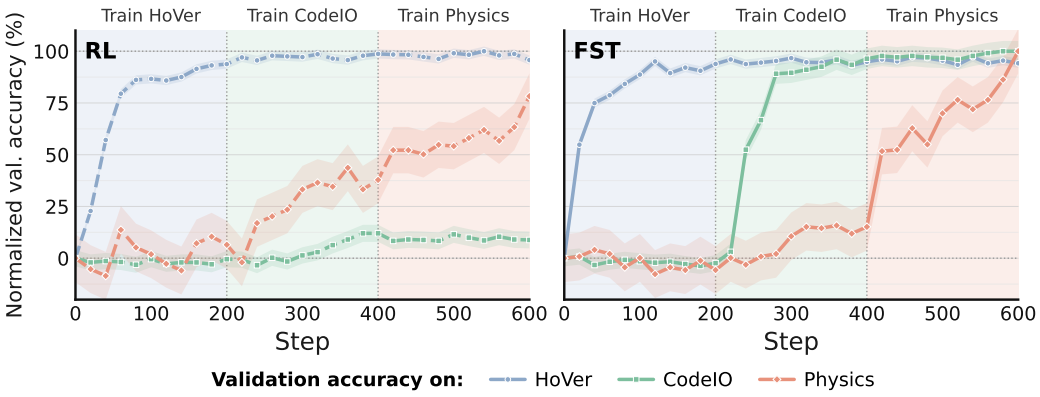

Figure 7: **Continual learning across `HoVer` → `CodeIO` → `Physics`**: a single uninterrupted training run that switches task every 200 steps. The y-axis is per-task validation accuracy, normalized so that 0 is the base model's accuracy on the task and 100 is the peak reached on that task by either method; bands are ± 1 standard error over the validation rollouts. FST (solid) reaches near-peak on every stage; RL (dashed) acquires `HoVer` but completely stalls on `CodeIO` and only partially recovers on `Physics`.

Y. As a no-prior-training reference, we also run Phase 2 starting from the base model. We test `Math` → `HoVer-hard` and `Physics` → `HoVer-hard`.

Figure 6 shows that in Phase-2, FST-init outperforms RL-init through the 400-step probe in both settings. The contrast is sharpest in `Math` → `HoVer-hard`: prior RL collapses `HoVer-hard` learnability to near-zero, the RL-init curve drops to $\sim 0\%$ within 40 steps and stays flat for the rest of the run. In contrast, FST-init reaches performance close to the base-init reference. On `Physics` → `HoVer-hard`, FST-init finishes at 24.2% and is still climbing, versus RL-init's 19.9% at step 400 . This indicates that, unlike RL, FST does not over-specialize the slow weights to task X: the resulting checkpoint retains capacity to learn a new task Y, exhibiting higher plasticity.

Advantage 5: Fast-Slow Training Improves Continual Learning A continual learning algorithm must keep absorbing new tasks as training proceeds, without losing the capacity to absorb later ones (Dohare et al., 2024; Tang et al., 2025; Lyle et al., 2022). To test this we run a single uninterrupted training pass over three tasks, sequentially swapping the task every 200 steps - first 200 steps with `HoVer` (multi-hop fact verification), then `CodeIO` (code-output prediction), and finally `Physics` (multiple-choice from `sciknoweval`). In this setting, the same live training trajectory must absorb

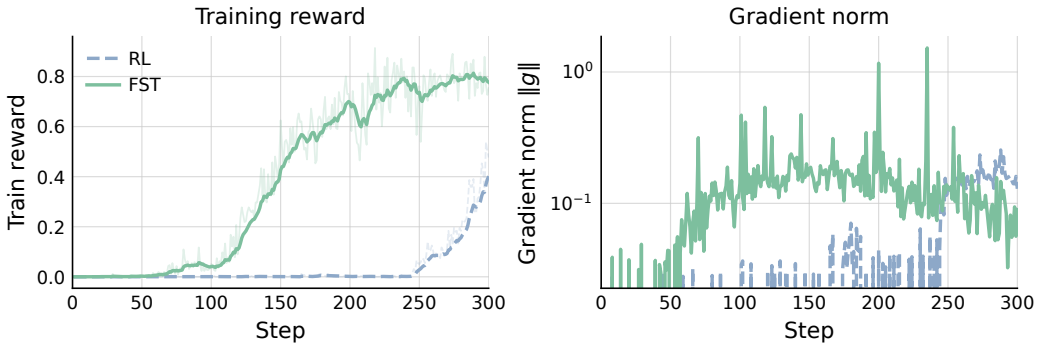

Figure 8: **Star Graph Search Task. Left:** Training reward improves around step 50 for FST, versus 250 for RL. **Right:** Gradient norm $\|g\|$ (log scale) shows earlier sustained gradient signal for FST's slow-weight updates; RL's gradients remain mostly small until reward improves.

three task changes back-to-back, mirroring how a deployed model would actually be trained on a stream of incoming tasks.

Figure 7 shows evaluation on all three tasks at different points across the full 600-step training run, normalized per task so that 0 is the base model's accuracy and 100% is the peak reached on the task by either method. FST reaches near-peak in every stage while learning faster within each stage, mirroring the data-efficiency gap of Section 4 Advantage 1. The contrast is sharpest in the second stage, `CodeIO`: across the full 200-step budget, RL barely lifts off its starting accuracy, peaking at 21.1% mean@4 (a +2.6pp gain over its 18.5% stage-start), while FST climbs to near-peak in about 80 steps (less than half the budget) and finishes the stage at 39.8%, a +21.3pp gain (a $\sim 8\times$ within-stage acquisition rate over RL, and a +18.7pp absolute lead at step 400). This demonstrates that FST is a promising continual-learning algorithm for LLMs: by routing task-level adaptation through both the textual fast-weight channel Φ in addition to θ, the method remains capable of acquiring later tasks under continued optimization. In Appendix F, we compare FST with replay, KL-penalty, and GEPA-once-per-task baselines, repeat the experiment under a second task ordering and run a longer stream of smaller tasks.

5 Why Does Fast-Slow Training Work?

The empirical benefits in Section 4 raise the questions: where do the benefits come from exactly and which component is doing the majority of the work in which setting? The two studies below isolate these questions.

Observation 1: Fast Weights Acquire Task Signal Faster Than Slow Weights We compare FST and RL on a synthetic star-graph task with near-zero base-model reward: finding a path between two labeled nodes (Appendix G.3).

Parameter-only RL remains near zero for roughly 250 steps, whereas FST reaches measurable reward around step 50 (Figure 8), driven primarily by the first few GEPA cycles. Textual feedback identifies where a submitted path fails, allowing GEPA to extract task structure from a handful of rollouts and encode it in Φ before substantial slow-weight adaptation. Although these early prompts solve only a few problems, they provide an initial reward signal that enables subsequent RL updates to improve rapidly.

Observation 2: Fast and Slow Weights Both Optimizing for Reward Raise Performance Ceiling Figure 9 decomposes the in-distribution gain on each training task into slow-weight and fast-weight contributions, evaluating every combination of {base, FST-trained weights} with {original prompt, FST-evolved prompt}. On `HoVer-hard`, combining both channels yields 21.2% pass@1, versus 11.6% with trained model weights alone and 10.6% with evolved prompts alone. `CodeIO` shows the same complementarity. In contrast, almost all the `Math (Polaris)` gain comes from slow weights (20.0% $\rightarrow$ 47.2%), consistent with the custom SFT base's weaker instruction-following (Appendix D.1). Thus, FST allows a task-dependent division of labor rather than requiring either channel

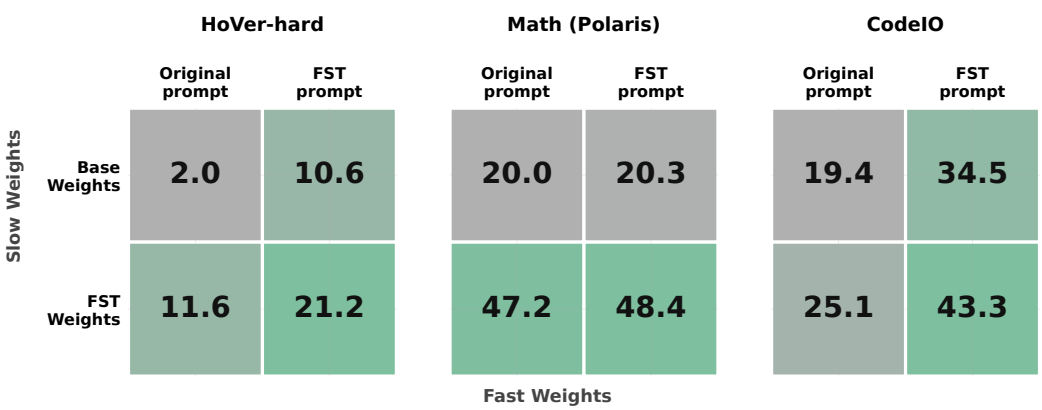

Figure 9: **In-distribution gain decomposed into slow- and fast-weight contributions** (pass@1, %). For each task, we evaluate four combinations: base or FST-trained model weights, with the original prompt or the FST-evolved prompt. On `HoVer-hard` and `CodeIO`, both channels contribute and the joint cell (FST weights + FST prompt) dominates. On `Math (Polaris)`, almost all of the gain is carried by the slow weights.

to dominate. Further decomposition and design ablations appear in Figure 19 and Appendix A.1. Naive fast-to-slow distillation also plateaus below FST (Appendix D.3), supporting direct reward optimization of both channels rather than replacing slow-weight RL with distillation.

6 Related Work

We compare FST with methods that combine prompt optimization and model weight updates below. We compare against relevant baselines in Appendix F.1 and review the broader literature on RL post-training, prompt optimization, and complementary learning systems in Appendix H.

Modern fast–slow methods for LLM RL. A small but growing body of work combines textual feedback with reward-driven weight updates. BetterTogether (Soylu et al., 2024) alternates SFT with prompt optimization over a DSPy pipeline, albeit instantiated with prompt optimizers that don't use textual feedback; we extend this to a new training paradigm instantiated in verifiable-reward RL in which a Pareto-frontier prompt population created with textual feedback co-evolves with the policy. LANPO (Li et al., 2025a) interleaves language and numerical feedback via per-instance reflections; we instead maintain a cross-problem Pareto-frontier population. Recent work E-SPL (Zhang et al., 2026a) explores prompt optimization and RL at a smaller scale with focus on performance rather than adaptation. mmGRPO (Ziems et al., 2026) runs prompt optimization once and then RL on a DSPy program; we interleave the two across cycles. Appendix B.1 compares FST with this sequential pipeline and with LANPO on CodeIO and HoVer-hard. POPE (Qu et al., 2026) prefixes hard prompts with partial reference solutions. By contrast, FST learns task-level prompts that condition any rollout, and combining the two is a natural direction for future work.

7 Conclusion

We present a fast-slow framework for LLM post-training that jointly optimizes the slow model parameters θ via RL and a fast textual-context population Φ via reflective prompt evolution, interleaving the two channels. Across CodeIO, Math, and HoVer-hard, this co-optimization reaches matched performance with 1.4–$3\times$ fewer optimizer steps, attains a higher asymptote, and incurs lower KL displacement than RL alone, which in turn translates into preserved plasticity and stronger continual-learning behavior on new tasks. More broadly, our results suggest that effective post-training should not ask model parameters to absorb all forms of adaptation. Fast textual weights can capture task-specific and rapidly evolving improvements, while slow weights can focus on consolidating persistent behavior. This division of labor offers a path toward post-training methods that are more data-efficient, less destructive, and more amenable to continual learning.

AI USE STATEMENT

We used generative AI tools to help polish the writing and prepare graphs and figures. We have reviewed all AI-assisted work. The authors are responsible for the final text, claims, graphs, and figures.

REPRODUCIBILITY STATEMENT

The method is described in Section 3, with pseudocode in Appendix G.2. Appendix G.4 provides training and prompt-optimization settings. Appendix G.3 describes the star-graph data construction and scoring, and Appendix D.4 includes seed and optimized prompts.

REFERENCES

Eshaan Agarwal, Joykirat Singh, Vivek Dani, Raghav Magazine, Tanuja Ganu, and Akshay Nambi. Promptwizard: Task-aware prompt optimization framework. *arXiv preprint arXiv:2405.18369*, 2024.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

Chenxin An, Zhihui Xie, Xiaonan Li, Lei Li, Jun Zhang, Shansan Gong, Ming Zhong, Jingjing Xu, Xipeng Qiu, Mingxuan Wang, and Lingpeng Kong. Polaris: A post-training recipe for scaling reinforcement learning on advanced reasoning models, 2025. URL `https://hkunlp.github.io/blog/2025/Polaris`.

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *Advances in neural information processing systems*, 30, 2017.

Jimmy Ba, Geoffrey E Hinton, Volodymyr Mnih, Joel Z Leibo, and Catalin Ionescu. Using fast weights to attend to the recent past. *Advances in neural information processing systems*, 29, 2016.

Gregor Bachmann and Vaishnavh Nagarajan. The pitfalls of next-token prediction. *arXiv preprint arXiv:2403.06963*, 2024.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms. *Advances in Neural Information Processing Systems*, 37:71596–71642, 2024.

Ganqu Cui, Yuchen Zhang, Jiacheng Chen, Lifan Yuan, Zhi Wang, Yuxin Zuo, Haozhan Li, Yuchen Fan, Huayu Chen, Weize Chen, et al. The entropy mechanism of reinforcement learning for reasoning language models. *arXiv preprint arXiv:2505.22617*, 2025.

Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric Xing, and Zhiting Hu. Rlprompt: Optimizing discrete text prompts with reinforcement learning. In *Proceedings of the 2022 conference on empirical methods in natural language processing*, pp. 3369–3391, 2022.

Shibhansh Dohare, J Fernando Hernandez-Garcia, Qingfeng Lan, Parash Rahman, A Rupam Mahmood, and Richard S Sutton. Loss of plasticity in deep continual learning. *Nature*, 632(8026): 768–774, 2024.

Wei Du, Shubham Toshniwal, Branislav Kisacanin, Sadegh Mahdavi, Ivan Moshkov, George Armstrong, Stephen Ge, Edgar Minasyan, Feng Chen, and Igor Gitman. Nemotron-math: Efficient long-context distillation of mathematical reasoning from multi-mode supervision. *arXiv preprint arXiv:2512.15489*, 2025.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

Lapo Frati, Neil Traft, Jeff Clune, and Nick Cheney. *Reset It and Forget It: Relearning Last-Layer Weights Improves Continual and Transfer Learning*. IOS Press, October 2024. ISBN 9781643685489. doi: 10.3233/faia240840. URL http://dx.doi.org/10.3233/FAIA240840.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *International conference on machine learning*, pp. 10835–10866. PMLR, 2023.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains. In *International Conference on Learning Representations*, volume 2026, pp. 127924–127945, 2026.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024a. URL https://arxiv.org/abs/2401.14196.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 34133–34156, 2024b.

Geoffrey E Hinton and David C Plaut. Using fast weights to deblur old memories. In *Proceedings of the ninth annual conference of the Cognitive Science Society*, pp. 177–186, 1987.

Jonas Hübotter, Frederike Lübeck, Lejs Behric, Anton Baumann, Marco Bagatella, Daniel Marta, Ido Hakimi, Idan Shenfeld, Thomas Kleine Buening, Carlos Guestrin, et al. Reinforcement learning via self-distillation. *arXiv preprint arXiv:2601.20802*, 2026.

Yichen Jiang, Shikha Bordia, Zheng Zhong, Charles Dognin, Maneesh Singh, and Mohit Bansal. Hover: A dataset for many-hop fact extraction and claim verification. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 3441–3460, 2020.

Damjan Kalajdzievski. Scaling laws for forgetting when fine-tuning large language models. *arXiv preprint arXiv:2401.05605*, 2024.

Devvrit Khatri, Lovish Madaan, Rishabh Tiwari, Rachit Bansal, Venkata Sai Surya Subramanyam Duvvuri, Manzil Zaheer, Inderjit Dhillon, David Brandfonbrener, and Rishabh Agarwal. The art of scaling reinforcement learning compute for llms. In *International Conference on Learning Representations*, volume 2026, pp. 72438–72467, 2026.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Dharshan Kumaran, Demis Hassabis, and James L McClelland. What learning systems do intelligent agents need? complementary learning systems theory updated. *Trends in cognitive sciences*, 20(7): 512–534, 2016.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Ang Li, Yifei Wang, Zhihang Yuan, Stefanie Jegelka, and Yisen Wang. Lanpo: Bootstrapping language and numerical feedback for reinforcement learning in llms. *arXiv preprint arXiv:2510.16552*, 2025a.

Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. Codei/o: Condensing reasoning patterns via code input-output prediction. *arXiv preprint arXiv:2502.07316*, 2025b.

Long Li, Zhijian Zhou, Jiaran Hao, Jason Liu, Yanting Miao, Wei Pang, Xiaoyu Tan, Wei Chu, Zhe Wang, Shirui Pan, et al. The choice of divergence: A neglected key to mitigating diversity collapse in reinforcement learning with verifiable reward. In *International Conference on Learning Representations*, volume 2026, pp. 64742–64768, 2026.

Yong Lin, Hangyu Lin, Wei Xiong, Shizhe Diao, Jianmeng Liu, Jipeng Zhang, Rui Pan, Haoxiang Wang, Wenbin Hu, Hanning Zhang, et al. Mitigating the alignment tax of rlhf. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pp. 580–606, 2024.

Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, and Yue Zhang. An empirical study of catastrophic forgetting in large language models during continual fine-tuning. *IEEE Transactions on Audio, Speech and Language Processing*, 2025.

Clare Lyle, Mark Rowland, and Will Dabney. Understanding and preventing capacity loss in reinforcement learning. *arXiv preprint arXiv:2204.09560*, 2022.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

James L McClelland, Bruce L McNaughton, and Randall C O'Reilly. Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. *Psychological review*, 102(3):419, 1995.

MiniMax. Minimax-m1: Scaling test-time compute efficiently with lightning attention, 2025. URL https://arxiv.org/abs/2506.13585.

Team Olmo, Allyson Ettinger, Amanda Bertsch, Bailey Kuehl, David Graham, David Heineman, Dirk Groeneveld, Faeze Brahman, Finbarr Timbers, Hamish Ivison, et al. Olmo 3. *arXiv preprint arXiv:2512.13961*, 2025.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, 2024.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Quang Pham, Chenghao Liu, and Steven Hoi. Dualnet: Continual learning, fast and slow. *Advances in neural information processing systems*, 34:16131–16144, 2021.

Hoang Phan, Xianjun Yang, Yuanshun Yao, Jingyu Zhang, Shengjie Bi, Xiaocheng Tang, Madian Khabsa, Lijuan Liu, and Deren Lei. Beyond reasoning gains: Mitigating general-capability forgetting in large reasoning models. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 34368–34398, 2026.

Jatin Prakash and Anirudh Buvanesh. What can you do when you have zero rewards during rl? *arXiv preprint arXiv:2510.03971*, 2025.

Archiki Prasad, Peter Hase, Xiang Zhou, and Mohit Bansal. Grips: Gradient-free, edit-based instruction search for prompting large language models. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 3845–3864, 2023.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 7957–7968, 2023.

Yuxiao Qu, Amrith Setlur, Virginia Smith, Ruslan Salakhutdinov, and Aviral Kumar. Pope: Learning to reason on hard problems via privileged on-policy exploration. *arXiv preprint arXiv:2601.18779*, 2026.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: your language model is secretly a reward model. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International conference on machine learning*, pp. 9355–9366. PMLR, 2021.

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Idan Shenfeld, Jyothish Pari, and Pulkit Agrawal. Rl's razor: Why online reinforcement learning forgets less. In *International Conference on Learning Representations*, volume 2026, pp. 59839–59864, 2026.

Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 4222–4235, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.346. URL https://aclanthology.org/2020.emnlp-main.346/.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Alessandro Sordoni, Eric Yuan, Marc-Alexandre Côté, Matheus Pereira, Adam Trischler, Ziang Xiao, Arian Hosseini, Friederike Niedtner, and Nicolas Le Roux. Joint prompt optimization of stacked llms using variational inference. *Advances in Neural Information Processing Systems*, 36: 58128–58151, 2023.

Dilara Soylu, Christopher Potts, and Omar Khattab. Fine-tuning and prompt optimization: Two great steps that work better together. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 10696–10710, 2024.

Zafir Stojanovski, Oliver Stanley, Joe Sharratt, Richard Jones, Abdulhakeem Adefioye, Jean Kaddour, and Andreas Köpf. Reasoning gym: Reasoning environments for reinforcement learning with verifiable rewards, 2025. URL https://arxiv.org/abs/2505.24760.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, 2026.

Hongyao Tang, Johan Obando-Ceron, Pablo Samuel Castro, Aaron Courville, and Glen Berseth. Mitigating plasticity loss in continual reinforcement learning by reducing churn. *arXiv preprint arXiv:2506.00592*, 2025.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Yuxin Wen, Neel Jain, John Kirchenbauer, Micah Goldblum, Jonas Geiping, and Tom Goldstein. Hard prompts made easy: Gradient-based discrete optimization for prompt tuning and discovery. *Advances in Neural Information Processing Systems*, 36:51008–51025, 2023.

Shirley Wu, Parth Sarthi, Shiyu Zhao, Aaron Lee, Herumb Shandilya, Adrian Mladenic Grobelnik, Nurendra Choudhary, Edward Huang, Karthik Subbian, Linjun Zhang, et al. Optimas: Optimizing compound ai systems with globally aligned local rewards. In *International Conference on Learning Representations*, volume 2026, pp. 149685–149706, 2026.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Lunjun Zhang, Ryan Chen, and Bradly C Stadie. Evolutionary system prompt learning for reinforcement learning in llms. *arXiv preprint arXiv:2602.14697*, 2026a.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026b.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*, 2022.

Noah Ziems, Dilara Soylu, Lakshya A Agrawal, Isaac Miller, Liheng Lai, Chen Qian, Kaiqiang Song, Meng Jiang, Dan Klein, Matei Zaharia, et al. Composing policy gradients and prompt optimization for language model programs. In *Proceedings of the ACM Conference on AI and Agentic Systems*, pp. 197–213, 2026.

APPENDIX CONTENTS

A Ablations **17**

A.1 CodeIO design ablations 17

A.2 A wider cycle-length sweep 18

A.3 Replication on HoVer-hard 18

B Baselines **19**

B.1 Prompt optimization once, then RL 19

B.2 More prompt-optimization budget for the static pipeline 20

B.3 Prompt optimization after RL 20

B.4 LANPO 21

C Efficiency **21**

C.1 Rollout reuse 21

C.2 Total rollout and reflection cost 22

D Additional results **22**

D.1 KL-vs-reward 22

D.2 Scaling the base model 23

D.3 Explicit fast-to-slow distillation 23

D.4 Evolved GEPA prompts during FST training 25

E Non-verifiable tasks **30**

E.1 RaR-Medicine: task and setup 30

E.2 Displacement versus reward 30

E.3 Training curves 31

F Continual learning **31**

F.1 Additional baselines 31

F.2 A second task ordering 32

F.3 A longer stream of smaller tasks 32

G Experimental setup **33**

G.1 GEPA 33

G.2 Algorithm pseudocode 33

G.3 Star-graph dataset construction 33

G.4 Hyperparameters and compute 35

H Additional related work **36**

I Discussion **36**

A ABLATIONS

A.1 CODEIO DESIGN ABLATIONS

In this section, we ablate the design choices for FST. We run sweeps on CodeIO (Qwen3-8B thinking, light-recipe defaults from Appendix G.4) and reported at a matched RL step (500) on the held-out CodeIO mean@4. The unmodified RL-only baseline at the same step (**39.65**%) is included in every panel for reference.

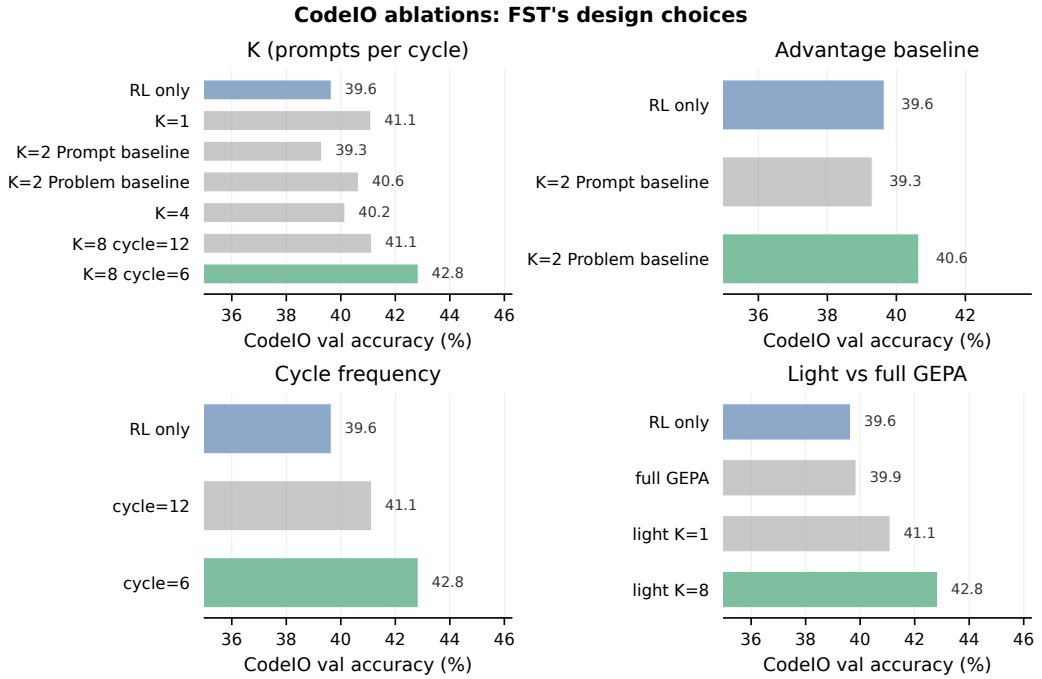

Figure 10: CodeIO design ablations (val mean@4 at training step 500). Sage bars mark the headline configuration in each panel; gray bars are the alternatives swept; the dashed line indicates RL-only at the same matched step. (**a**) Population size K. (**b**) Advantage baseline at $K{=}2$: per-prompt (Prompt baseline) vs. per-problem (Problem baseline). (**c**) Cycle length T at $K{=}8$, Problem baseline. (**d**) Light vs. full GEPA recipe.

Population size K (Fig. 10a). We sweep $K \in \{1, 2, 4, 8\}$ holding the rest of the recipe at light / Problem baseline / cycle $T{=}6$. Every $K \geq 1$ improves on RL-only (39.65%), indicating that even a single optimized prompt carries useful task signal into RL. Performance is non-monotonic in K: $K{=}1$ already buys $+1.5$ pp (41.10%), $K{=}2$ Problem baseline and $K{=}4$ are roughly tied at $\sim$40.4%, and the gain saturates at $K{=}8$ with **42.84**% (cycle 6), the headline configuration. Larger K widens the within-group prompt distribution and gives GRPO advantages a richer signal to compare against, but only when paired with a tight cycle (see panel c). $K{=}8$ with cycle$=12$ recovers most but not all of the gain (41.13%).

Advantage baseline at $K{=}2$ (Fig. 10b). With a fixed population of $K{=}2$ prompts, the choice of GRPO advantage baseline matters. The *Prompt baseline* (per-prompt: rollouts under each $\phi^{(k)}$ are normalized against their own group mean and std) yields 39.30%, slightly *below* RL-only. The *Problem baseline* (per-problem: all G rollouts under both prompts share a single group statistic) yields **40.65**%, $+1.4$ pp over the Prompt baseline and $+1.0$ pp over RL-only. Our intuition for the problem baseline working well is as follows. The prompt baseline makes prompts compete only with themselves and discards the cross-prompt comparison entirely, while the problem baseline exposes the policy gradient to which prompt a stronger response came from on the same problem. The problem baseline is the default for all main-text experiments.

Cycle length T at $K{=}8$ (Fig. 10c). Holding $K{=}8$ Problem-baseline fixed, we compare $T{=}6$ vs. $T{=}12$ RL steps between successive GEPA optimizations. Cycle $T{=}6$ reaches $\mathbf{42.84}\%$; doubling the cycle to $T{=}12$ drops mean@4 by 1.7 pp to 41.13%, giving up more than half of the advantage over RL-only. This is the expected effect of staleness: as θ moves between GEPA cycles, the prompts in Φ become increasingly mistuned to the current policy, and the per-question rollout-group signal degrades. $T{=}6$ is short enough that the population Φ_c remains close to optimal across the cycle, though it requires twice as many GEPA optimizations as cycle 12.

Light vs. full GEPA recipe (Fig. 10d). The "light" recipe uses $K{=}4$ candidates, a smaller per-cycle GEPA budget (`num_eval_examples=` 192, `max_metric_calls=` 960), and a proposer prompt that asks `gpt-5.2` for incremental edits to the current best prompt rather than rewrites from scratch. The "full" recipe is the original configuration from Agrawal et al. (2026): $K{=}1$, doubled metric budget (`max_metric_calls=` 1922), and an open-ended proposer that allows full rewrites. Within our setup the full recipe gives essentially no lift over RL-only (39.85% vs. 39.65%); the light recipe at the same $K{=}1$ yields $+1.5$ pp (41.10%); and scaling light to $K{=}8$ gives a further $+1.7$ pp ($\mathbf{42.84}\%$). We believe two factors contribute. First, full's $K{=}1$ pinches the population channel that turns out to matter most in panel a. Second, the open-ended rewrite proposer is more prone to drift on a moving policy: incremental edits keep the population's induction biases coherent across cycles, while wholesale rewrites tend to discard structure each round. The combined effect is a recipe that runs on roughly half the GEPA budget and outperforms the original by $\sim$3 pp on this task.

A.2 A WIDER CYCLE-LENGTH SWEEP

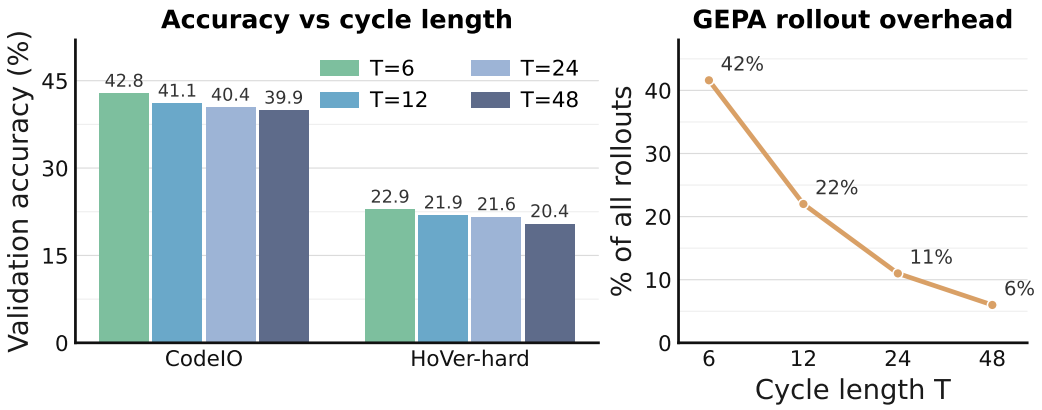

Figure 11: **Accuracy is weakly sensitive to cycle length T; rollout cost is strongly sensitive.** *Left:* held-out accuracy for $T \in \{6, 12, 24, 48\}$ on CodeIO (mean@4, 700 training steps) and HoVer-hard (mean@4, 300 training steps). *Right:* GEPA rollouts as a share of all rollouts generated during training, for the same runs.

Panel c compares $T{=}6$ against $T{=}12$ on a single task. To check that longer cycles do not eventually pay off, we extend the sweep to $T \in \{6, 12, 24, 48\}$ and repeat it on HoVer-hard. Accuracy declines monotonically but slowly with T: from 42.8% at $T{=}6$ to 39.9% at $T{=}48$ on CodeIO, and from 22.9% to 20.4% on HoVer-hard. Cost moves far more over the same range: the share of rollouts spent by GEPA falls from 41.6% at $T{=}6$ to 6.0% at $T{=}48$. Short cycles keep the prompt population tightly coupled to the current policy and buy a small, consistent accuracy edge; long cycles give most of that edge back in exchange for a several-fold reduction in prompt-optimization overhead. In practice we therefore pick T by rollout budget: $T{=}6$ when rollouts are cheap, larger T when they are not, with the expectation of losing roughly a point of accuracy per doubling.

A.3 REPLICATION ON HOVER-HARD

The CodeIO sweeps above are single-task, so we repeated the three component ablations on HoVer-hard. The population-size trend replicates: $K{=}8$ reaches 21.9% against 20.8% for a single prompt

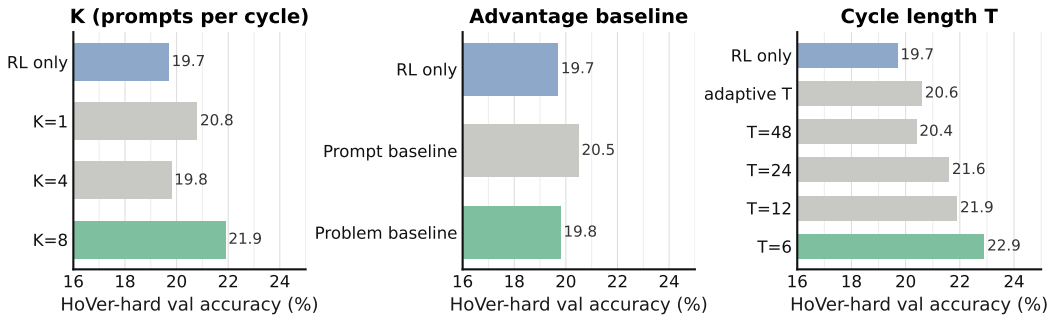

Figure 12: **Design ablations replicated on HoVer-hard** (held-out mean@4 at the best checkpoint of a 300-step run). Same layout as Figure 10: sage marks the headline configuration, grey the alternatives, blue the RL-only run. (**Left**) Population size K. (**Middle**) Advantage baseline. (**Right**) Cycle length T, including an adaptive rule that starts a GEPA cycle only when the training reward stalls.

and 19.8% for K=4. The cycle-length trend replicates too, and it is the largest effect on this task: accuracy falls from 22.9% at T=6 to 20.4% at T=48. The advantage baseline is the one component whose effect does not carry over: on HoVer-hard the per-prompt baseline (20.5%) and the per-problem baseline (19.8%) are within a point of each other, whereas on CodeIO the per-prompt baseline fell below RL-only. We keep the per-problem baseline throughout the paper because it never hurts and is decisive on CodeIO. Finally, to check that T need not be hand-tuned, we ran a variant that starts a GEPA cycle only when an exponential moving average of the training reward has stopped improving. It reaches 20.6%, between the T=24 and T=48 runs, at a fraction of their prompt-optimization cost.

B Baselines

B.1 Prompt optimization once, then RL

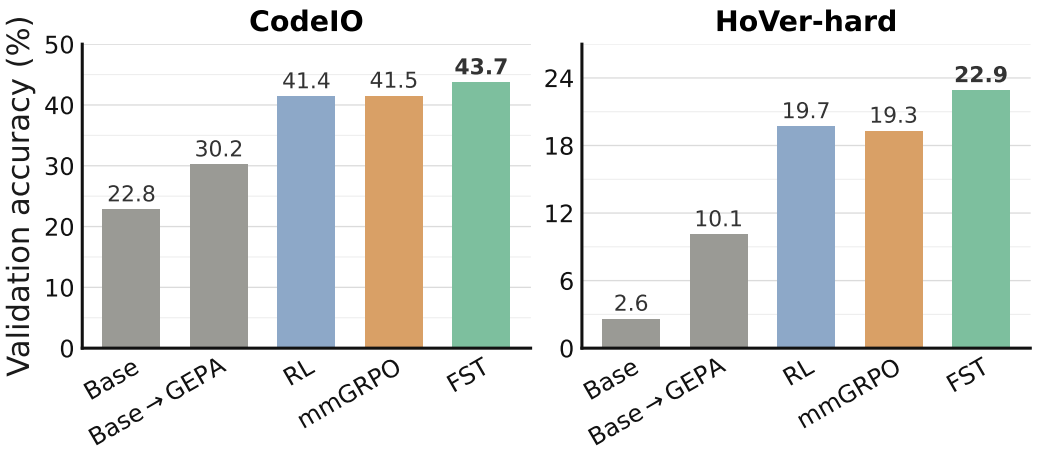

Figure 13: **A strong static prompt does not reproduce FST.** Held-out accuracy (mean@4) for the base model, the base model with a GEPA-optimized prompt, RL, the static pipeline (GEPA once on the base model, prompt frozen, then RL; the recipe of mmGRPO (Ziems et al., 2026)), and FST. CodeIO values are compared at a matched training step (1100); HoVer-hard values are the best checkpoint per arm.

FST interleaves prompt optimization with RL. A natural alternative, and the recipe of mmGRPO (Ziems et al., 2026), is to optimize the prompt once on the base model, freeze it, and run RL on top. Comparing the two separates the benefit of *co-evolving* the prompt with the policy

from the benefit of merely having a strong prompt available during training. We run this pipeline on CodeIO and HoVer-hard: GEPA on the base model, then the unmodified RL baseline with the best resulting prompt held fixed throughout training.

Figure 13 shows the result. On both tasks the static pipeline lands on the RL baseline rather than between RL and FST: 41.5% versus 41.4% on CodeIO, and 19.3% versus 19.7% on HoVer-hard, against 43.7% and 22.9% for FST. The GEPA prompt with used with the base model lifts CodeIO from 22.8% to 30.2% and HoVer-hard from 2.6% to 10.1%. However, through training RL eventually learns the same task knowledge and both arms arrive at the same place. Optimizing the prompt once and training cannot reproduce is the co-adaptation of prompt and weights during training, which is where FST's remaining 2–3 pp comes from.

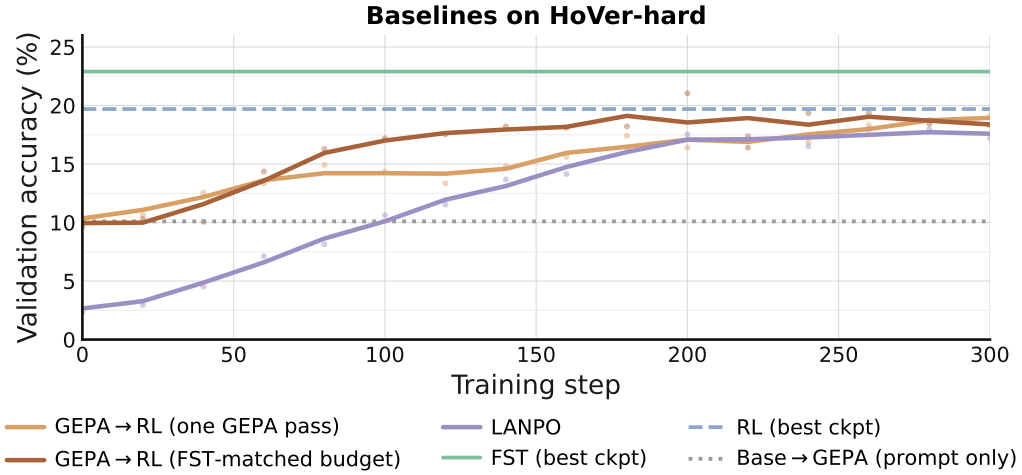

Figure 14: **Baselines on HoVer-hard** (held-out mean@4, evaluated every 20 steps). Two GEPA→RL arms differ only in the budget of the GEPA stage: one GEPA pass, or the metric-call budget that FST spends over a whole 300-step run. Horizontal lines mark the best checkpoints of RL and FST from Figure 13 and the prompt-only level. Both static-prompt arms and LANPO (Li et al., 2025a) finish at or below RL and well below FST.

B.2 More prompt-optimization budget for the static pipeline

One objection to the comparison above is that FST runs GEPA every T steps while the static pipeline runs it once, so the static prompt might simply be under-optimized. We therefore reran the pipeline on HoVer-hard with the GEPA stage given the same metric-call budget that FST consumes over a full 300-step ru. Figure 14 shows RL on top of the two prompts, alongside LANPO (next paragraph). At step 300 the standard-budget arm is at 18.7% and the matched-budget arm at 18.5%, against 22.9% for FST. The two arms differ only in how much prompt optimization is done before RL, and they end in the same place, so the gap to FST comes from *when* the prompt optimization is spent, during RL against the current policy, rather than from how much of it there is.

B.3 Prompt optimization after RL

The other sequential order is RL first and GEPA afterwards on the trained checkpoint. Evaluating all arms under one protocol (pass@1 over 8 samples on the 221 HoVer-hard test problems, the protocol of Figure 9), RL alone reaches 16.4%, and running GEPA on the RL checkpoint at evaluation raises it to 19.3%. FST, evaluated with its own evolved prompt, reaches 21.2%. Handing the model weights from FST a prompt that was instead optimized for the base model *lowers* their accuracy to 17.9%: the weights and the prompt were adapted to each other during training, and neither order of sequential optimization reproduces that pairing.

B.4 LANPO

LANPO (Li et al., 2025a) also feeds language back into RL, but through a different channel: before each rollout, the policy is shown a reflection over its own earlier attempts at the same problem, drawn from a per-problem experience pool, with no prompt optimizer and no external reflection model. We implemented its intra-sample variant (context assigned per problem so that GRPO groups stay comparable, p=0.75 probability of including the reflection, and the format bonus of the original) and trained it with the HoVer-hard RL recipe for 300 steps. On HoVer-hard, LANPO learns steadily from the base model's 2.4% and reaches a best checkpoint of 18.0% (17.2% at step 300), below RL-only (19.7%) and 5 pp below FST (Figure 14). On CodeIO it reaches 37.9%, against 41.4% for RL-only and 43.7% for FST. Reflecting on the policy's own rollouts therefore does not substitute for optimizing the prompt against reward.

C EFFICIENCY

C.1 ROLLOUT REUSE

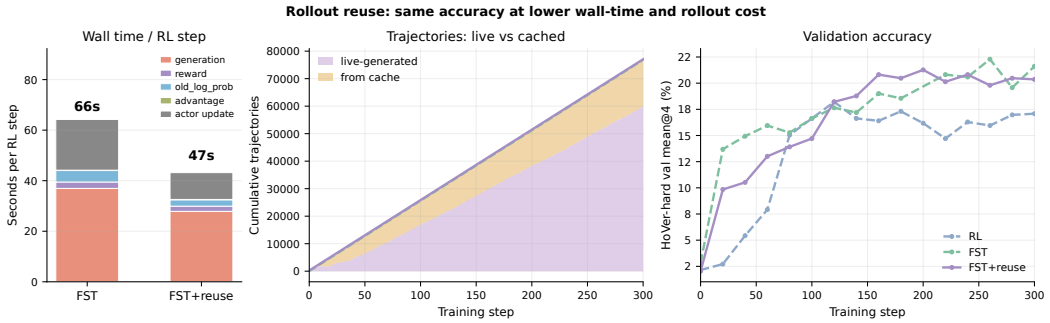

Figure 15: Rollout reuse on HoVer-hard, training step ≤300. **Left:** mean wall-time per RL step. FST+reuse drops from ∼66 s to ∼47 s (about 30% faster); the saving comes almost entirely from the generation phase. **Middle:** cumulative RL trajectories. The shaded region splits each step's 256 trajectories into live-generated (lavender) and from-cache (amber). By step 300 about ∼17,000 of the ∼77,000 RL trajectories were served from cache. **Right:** HoVer-hard val accuracy mean@4. FST+reuse tracks FST; the no-prompt RL baseline lags both.

Reusing GEPA rollouts for RL. FST interleaves GEPA prompt optimization with RL updates: every T RL steps GEPA runs a cycle that scores each candidate prompt $\phi^{(k)}$ on a small evaluation pool drawn from the same training distribution the next RL phase will see. Each evaluation entry is a tuple $(p, \phi^{(k)}, y, r)$ – a problem p, the prompt $\phi^{(k)}$ used, the sampled response y, and its scalar reward r. Without reuse, those tuples are thrown away once GEPA picks a new population Φ_{c+1}; the next RL step then re-rolls G fresh trajectories per problem from scratch under the new prompts. But because the population Φ_{c+1} is a Pareto-frontier subset of the prompts GEPA *already evaluated*, many of those discarded tuples are perfectly valid samples from the current policy under one of the current prompts. The natural optimization is to splice them into the RL group instead of regenerating them.

Cache mechanics. We maintain a per-(problem, prompt) cache of GEPA evaluation tuples produced during the most recent cycle. When the RL phase forms its rollout group of size batch $\times$ G across the prompts in Φ_{c+1}, it first claims any cached (p, ϕ, y, r) matching a (problem, prompt) slot and only generates the remaining slots live with vLLM. Cached and live trajectories are then concatenated into a single GRPO group; the policy gradient does not distinguish the two sources because both were sampled under prompts in the active population. The cache is cleared when GEPA produces the next Φ_{c+2}, so reused trajectories are at most T RL steps old (T=6 in our headline configuration).

Empirical effect. Figure 15 measures the impact on HoVer-hard. We compare two otherwise identical FST runs – one with reuse enabled and one without – through the first 300 training steps,

with a no-prompt RL baseline as a third reference. Wall-time per RL step (left panel) drops from $\sim$66 s to $\sim$47 s, a $\sim$29% speedup that comes almost entirely out of the generation phase: GEPA's own per-cycle cost is unchanged, but a substantial fraction of the following RL phase's rollouts no longer need to be sampled. The middle panel decomposes the cumulative trajectory budget into the live and from-cache components: by step 300, roughly 17k of the 77k total trajectories ($\sim$22%) were served from cache, with the cache hit rate concentrated in the first few RL steps after each GEPA cycle and tapering as the live group fills in problems not in GEPA's evaluation pool. The right panel shows that this comes at no accuracy cost: the reuse and non-reuse FST curves are within sampling noise of each other on HoVer-hard val mean@4, and both lead the no-prompt RL baseline throughout.

C.2 Total rollout and reflection cost

The step counts in Section 4 compare data consumed: one RL step is one batch of 32 problems with G=8 rollouts each for both FST and RL, and GEPA draws its reflection examples from problems the RL loop has already consumed, so a step is the same unit for both methods. GEPA does, however, generate additional rollouts of its own, mostly to score candidate prompts on its validation pool, and we account for those here. Table 1 gives the per-step averages over a full run at T=6. GEPA adds 92–123 rollouts per RL step, an overhead of 18–48%, and about one to three reflection-LM calls per step (under \$10 of API cost per training run). Because FST reaches RL's peak in 1.4–3$\times$ fewer steps, its total rollout count at matched accuracy is still lower than RL-only's on every task: 51% fewer on CodeIO and HoVer-hard and 16% fewer on Math. Wall-clock per step is higher and depends on T: on CodeIO, RL-only takes 163 s per step, FST takes 329 s at T=6 (2.0$\times$) and 249 s at T=12 (1.5$\times$), before the rollout reuse described above, which on HoVer-hard brings FST's per-step wall-clock below RL-only's.

Table 1: Rollout and reflection cost of FST at T=6. Rollouts and reflection-LM calls are averaged per RL step over a full run (a GEPA cycle fires every 6 steps). The last column compares the total number of rollouts each method has generated when it first reaches RL's peak accuracy.

	Rollouts / step		Reflection	Step	Rollouts at
Task	RL	GEPA (overhead)	calls / step	reduction	matched acc.
CodeIO	256	116 (45%)	0.96	3.0$\times$	$-$51%
HoVer-hard	256	123 (48%)	2.72	3.0$\times$	$-$51%
Math (Polaris)	512	92 (18%)	1.70	1.4$\times$	$-$16%

D Additional results

D.1 KL-vs-reward

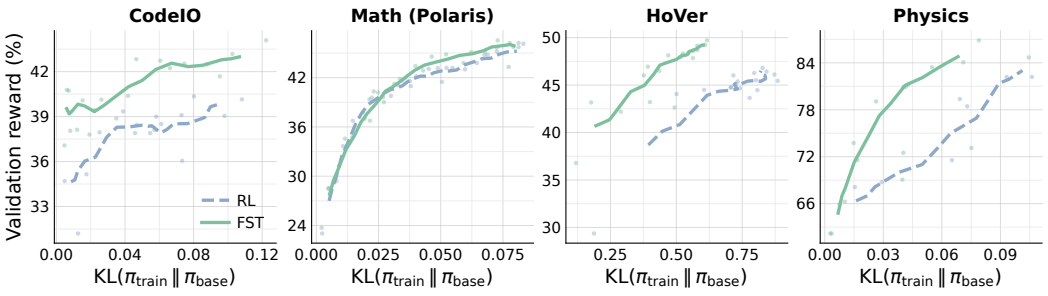

Figure 16: Validation reward versus $\mathrm{KL}(\pi_{\text{train}} \,\|\, \pi_{\text{base}})$ on all four training tasks: `CodeIO`, `Math` (Polaris), `HoVer`, and `Physics`. Same axes, smoothing, and conventions as Figure 5 in the main text. The three-task variant in the main text drops the Polaris panel for the reasons discussed below.

The Polaris (math) trajectories in Figure 16 look qualitatively different from the other three tasks: RL and FST sit on top of each other in $(\mathrm{KL}, \text{reward})$-space rather than FST pulling the frontier to the

left. We attribute this to the base model used for the Polaris runs. Unlike the other three tasks, which start from `Qwen3-8B` (an instruction-tuned model with strong format following), Polaris was trained on top of a model built by SFT'ing `Qwen3-8B-Base` on Nemotron, since the public `Qwen3-8B` is already saturated on Polaris. That custom SFT base has noticeably weaker instruction-following than the public Instruct checkpoint: the GEPA-evolved prompts that drive FST's KL gain on the other three tasks rely on the policy actually following format and self-checking instructions in the prompt, and on this base much of that signal is lost. The model still learns the math task from RL reward, so the reward axis behaves normally. Tt simply does not benefit from the prompt channel as strongly, and the two trajectories collapse onto each other. We expect the Polaris curve to look more like the other three with a stronger instruction-tuned base, but isolating that requires retraining and is left for future work.

D.2 Scaling the base model

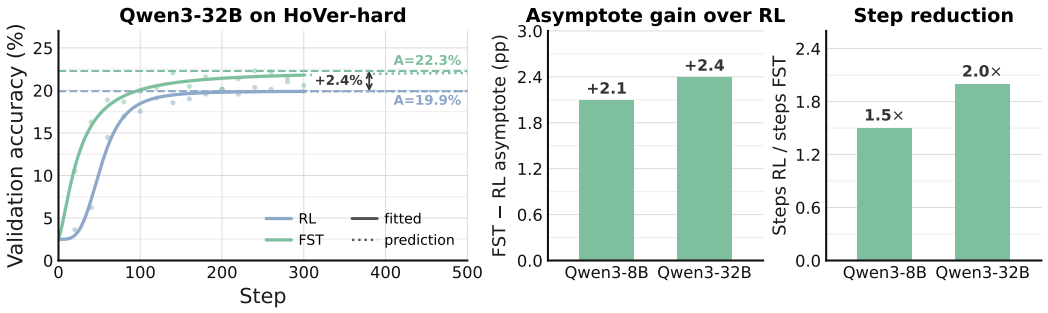

Figure 17: **FST scaling: two model sizes on HoVer-hard** (300 training steps, T=12, otherwise the settings of Appendix G.4). **Left:** Qwen3-32B validation curves for RL-only and FST, in the conventions of Figure 4: points are the per-evaluation values, the solid line is the fitted sigmoid of Section 4, dotted is its extrapolation, and dashed lines mark the fitted asymptotes A. **Middle:** the asymptote of FST minus that of RL-only at each size. **Right:** the number of steps RL-only needs to reach its own peak reward divided by the number FST needs.

Our results in the main paper use Qwen3-8B. To check that the gain survives a stronger base model, we trained Qwen3-32B (thinking mode) on HoVer-hard with RL-only and with FST. The 32B runs are shorter than the 8B headline runs (300 steps) and use T=12, so we compare them against 8B runs with the same budget and cycle length, and we read the comparison off the fitted asymptote of each arm rather than off the best checkpoint, which also removes the effect of where each run was stopped. Figure 17 shows the 32B curves and the two quantities that carry the paper's argument. At 32B the fitted asymptotes are 19.9% for RL-only and 22.3% for FST. The FST curve is still rising at step 300, so its fit is floored at the observed maximum and its asymptote is a lower bound. FST first reaches RL-only's peak (21.0%) at step 140 against RL-only's step 280. The asymptote gap between FST and RL-only is +2.1 pp at 8B and +2.4 pp at 32B, and the step reduction grows from 1.5× at 8B to 2.0× at 32B. At matched reward the 32B FST checkpoint also sits at 78% lower KL to the base model than the RL-only checkpoint. The raw best-checkpoint accuracies at 32B are 21.0% for RL-only and 22.3% for FST, from a base of 2.6%, the same base accuracy as at 8B, so the task does not become easy with scale. Appendix E.1 covers the other direction, a smaller model (Qwen3-4B) and a second family (OLMo-3-7B), on a rubric-scored task.

D.3 Explicit fast-to-slow distillation

The ceiling finding in Section 5 raises a natural question: can the gains from the fast textual channel be folded into model parameters explicitly, without ever doing RL on the slow weights? We test this by replacing the slow-weight policy-gradient update with an on-policy reverse-KL distillation loss

$$\mathcal{L}_{\text{distill}}(\theta) \;=\; \mathbb{E}_{x\sim\mathcal{D},\; y\sim\pi_\theta(\cdot|x)}\left[\sum_{t=1}^{|y|}\mathrm{KL}\Big(\pi_\theta(\cdot \mid x, y_{<t}) \,\Big\|\, \pi_{\bar\theta}(\cdot \mid x, \phi, y_{<t})\Big)\right], \tag{8}$$

where the teacher $\pi_{\bar{\theta}}(\cdot \mid x, \phi, \cdot)$ is the same model evaluated with a FST-evolved fast-weight prompt ϕ and fixed model weights $\bar{\theta}$, and the student $\pi_\theta(\cdot \mid x, \cdot)$ is conditioned only on the problem x. Sampling y on-policy from the student and minimizing the per-token reverse KL toward the teacher follows recent work on self-distillation (Hübotter et al., 2026). We call this variant FST-distill: the slow weights move only by chasing the teacher distribution induced by the fast-weight prompt, with no direct exposure to scalar reward.

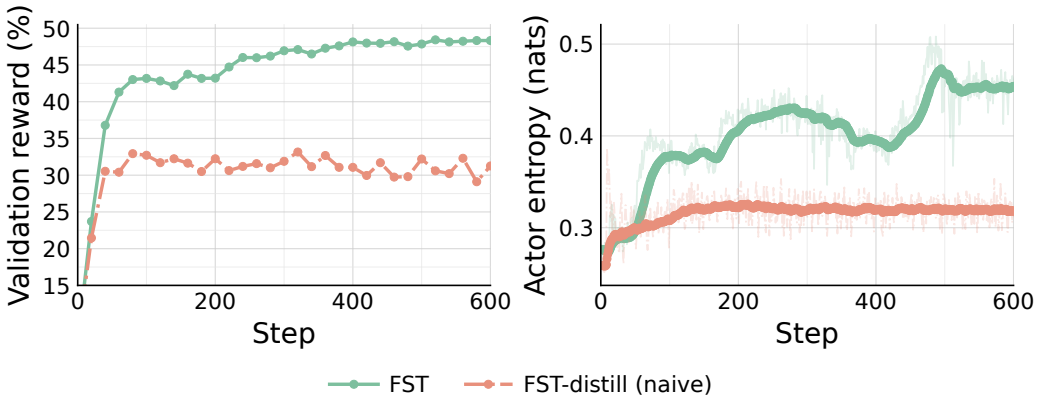

Figure 18: **Explicit fast-to-slow distillation on `HoVer`.** FST (green) is compared to FST-distill (orange), which updates θ only via the on-policy reverse-KL loss in Eq. 8 using a FST-evolved prompt ϕ as the teacher. **Left:** Validation reward. FST-distill rises above the prompt-only level by transferring fast-weight signal into model parameters across multiple updates, but plateaus well below FST, which has both channels optimizing reward jointly. **Right:** Actor entropy. Both methods preserve healthy entropy throughout training.

Figure 18 (left) shows that FST-distill iteratively transfers signal from the fast weights into the slow weights and rises above the prompt-only ceiling, but does not reach FST's reward. This is consistent with Observation 2. The fast channel alone is not enough to saturate the joint ceiling, and direct policy-gradient updates on θ against reward, run alongside the fast channel, account for the remaining gain. The entropy panel (right) also shows that the diversity benefits of training under a Pareto-frontier prompt population persist even when the slow update is purely distillation-based.

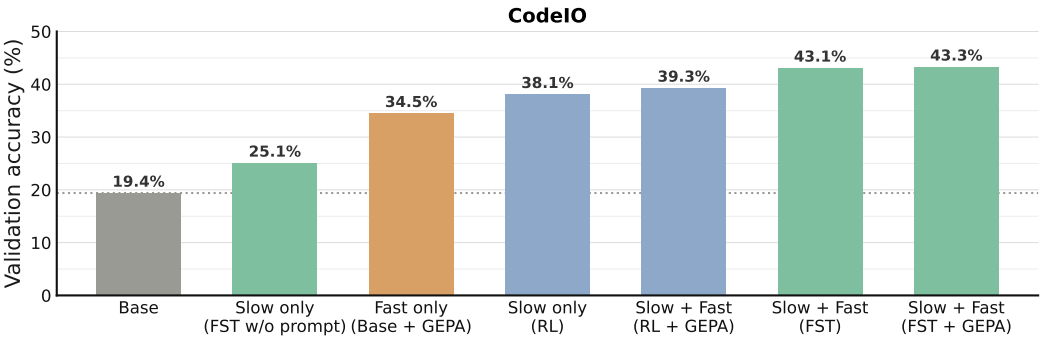

Figure 19: **Decomposing the Fast vs. Slow gain on CodeIO.** Step-matched (training step 650) validation accuracy (pass@1, computed from n=8 rollouts) on the held-out CodeIO set. *Slow only* isolates the parametric channel (RL- or FST-trained weights, evaluated without any GEPA prompt); the RL value is the training-time validation accuracy of the RL-only run at step 640, its nearest logged evaluation to step 650. *Fast only* isolates the textual channel (base model weights with a GEPA-evolved prompt). *Slow + Fast* combines them. Both channels contribute. FST compounds the benefits: without its prompt the FST model weights score 25.1%, below RL's 38.1%, but with the prompt they were trained with, the full Slow + Fast (FST) configuration reaches 43.1%, above RL given a GEPA prompt after training (39.3%), and an extra GEPA pass on top lifts it further to 43.3%.

D.4 EVOLVED GEPA PROMPTS DURING FST TRAINING

This section shows, for each FST training task, the *seed* prompt that GEPA started from and the *evolved* prompt at the matched-step FST checkpoint used in §4. The evolved prompt is the population's lead candidate (`gepa_state.json:current_prompts[0]`) of each Problem-baseline $K=\{4, 8\}$ training run at that step, so it is a prompt that was *co-evolved with the slow-weight RL update*, not a prompt obtained by running GEPA on the un-trained base policy in isolation. FST's rollout group at that step also draws from the rest of the K-prompt population, not just the single one shown here.

Across tasks, two patterns are consistent. First, GEPA almost never *rewrites* the seed. The $K=\{4, 8\}$ Problem-baseline recipe constrains the proposer (`gpt-5.2`) to small targeted edits, so the evolved prompt keeps the seed's basic role and output format and adds layered guidance on top. Second, the additions are almost entirely *negative-example specific*: each block addresses a failure mode the proposer observed during reflection on a small batch of low-reward rollouts (e.g., "do not invent placeholder numbers" for CodeIO, "do not skip pages with parenthetical disambiguators" for HoVer-hard, "re-check off-by-one in process/recurrence problems" for Polaris). The result is a long instruction list that reads less like a generic system prompt and more like a checklist of don'ts assembled from the specific population of mistakes the policy was making.

D.4.1 CODEIO

Seed Prompt – CodeIO

```
You are an expert at predicting the output of Python functions. Given a function
definition and its input values, trace the execution carefully step by step.
Account for control flow, mutable state, and the exact return value at the end.
Output the final answer as a JSON value matching the function's return type.
```

Evolved Prompt – CodeIO ($K=8$ Problem-baseline, training step 650)

```
You are a code output prediction assistant. Given a Python function and its inputs,
 simulate execution step-by-step (track state changes, loops, conditionals,
recursion, mutation/aliasing, and exact numeric behavior) and return the exact
result. Never estimate, "intuit", or placeholder any result. If the computation is
 long, still carry it through exactly; do not bail out with fabricated numbers or
"cannot compute".
If you cannot compute an exact value from the given code and inputs, keep tracing
and calculating until you can; NEVER insert made-up numbers. Wrong-but-plausible
placeholders are worse than continuing the derivation.

Before answering, explicitly determine the function's actual return value and its
JSON serialization form from the code's return statement (not from any natural-
language "output requirement"): output a bare JSON number/string/bool/null/list
when the function returns that; output a JSON object only when the function
actually returns a dict. Do not wrap scalars/strings inside {"return": ...} unless
 the code returns that dict, and do not put the final JSON in a code block.

When arithmetic is involved, compute it explicitly from the code using full-
precision intermediate values (keep enough digits to match Python float results);
for iterative numeric updates, keep a running trace and do not switch to
qualitative/steady-state guesses. Do not round intermediate results; carry full
float precision through to the end, and only then render the final JSON number/
string exactly as Python would. If the code calls round()/np.round() at specific
steps, apply those exactly (including the number of decimals and when they occur).
For large-integer/math/Decimal-heavy code, follow the algorithm as written (often
a recurrence/closed form) and carry exact integer/Decimal values through floors/
rounding steps; do not replace it with a rough guess or an unrelated formula.

Compute using Python's real semantics: preserve float64 rounding as produced by
Python/math/numpy; carry full precision through intermediate steps and do not "
pretty round" in the final output. Do not replace code with algebraic "
simplifications" unless you can prove they are identical under float64 rounding
and domain behavior; when in doubt, follow the exact operations/functions used in
the same order. For trig/exp/log, use identities only when exact; otherwise
```

```
evaluate as the runtime would (argument reduction, then compute), and do not
substitute rough angle approximations.

When numpy is used, respect array shape rules and broadcasting: np.asarray/list->
ndarray even if length==1; dot/matmul axes, transposes, slicing views vs copies;
functions may return arrays vs scalars depending on ndim, and .tolist() converts
arrays to Python lists--mirror that exactly.

For loops with threshold conditions (>, <, >=, <=), update variables in the exact
order and check the condition exactly as Python would after each iteration; do not
 assume constraints like non-negativity unless enforced by code--values may
legitimately become negative and loops may run past "physical" bounds.

If randomness is used and no seed/state is provided, you still must execute the
specific draws implied by the call (do not substitute theoretical expectations/
closed-form averages). Treat random calls as producing concrete (but unknown)
values only if they are actually derivable from given state; otherwise continue
tracing deterministically when possible and never replace with expected values.

Before finalizing, verify the return *type/shape* matches the code path taken (e.g
., list vs scalar, None vs value, dict keys present/absent). Return exactly what
the function returns. Do not "correct" perceived bugs/odd inputs--execute the code
 as written, even if the algorithm seems wrong. Pay special attention to in-place
mutation and aliasing, and to index-updates order (e.g., pointer increments/
decrements around swaps); re-check loop invariants against the actual code (avoid
assuming a standard algorithm variant).

Pay special attention to output type/serialization: output exactly the function's
return value as a single JSON value (no surrounding prose). Use null for Python
None; ensure numbers stay numbers, strings stay strings; no extra wrapper keys
unless returned. If the function returns a JSON string (e.g., via json.dumps),
your final output must be a JSON string value (including escape characters), not
the underlying dict/list. For strings containing backslashes or newlines, escape
them correctly for JSON.

When order matters, preserve the exact order produced by the code (append order in
 recursion, sorting direction, insertion order); do not reorder unless the code
actually reorders.

If randomness is used, assume the function's PRNG behavior is deterministic for
the call as written; propagate the specific generated values through the trace
rather than inventing placeholders.

Output ONLY the final value as valid JSON matching the function's return type
exactly. Do not guess; compute precisely, and always provide the final JSON value.
```

D.4.2 MATH (POLARIS)

Seed Prompt – Math (Polaris)

```
You are a precise math solver. Read the problem carefully, work through the
solution step by step with clear algebra and attention to numerical precision, and
 put your final answer inside \boxed{} on the last line.
```

Evolved Prompt – Math (Polaris) (K=4 Problem-baseline, training step 1050)

```
You are a precise math solver. Read the problem carefully, identify exactly what
is asked (including units/angle measures, definitions such as "sample variance",
and any inclusion/exclusion such as whether the empty set counts), and work
through the solution step by step with clear algebra and attention to numerical
precision. Prefer exact methods over lengthy numerical approximations, avoid
introducing unstated assumptions, and when using coordinates/casework ensure the
setup matches all given constraints before committing. If a required diagram/data
is referenced but not actually provided, do NOT guess missing values--state that
the answer cannot be determined from the given information.
```

```
Before finalizing, do a quick "problem re-parse" and verify any interpreted
details (e.g., phrases like "one between" vs "adjacent", "directly left" vs "
somewhere left", "internal vs external tangent", "number of distinct points vs
number of events", endpoints/inclusion, whether the empty set is included, and
whether a parameter is \(N\) vs \(N+1\)/\(N+2\)). For any phrase like "\(k\)
houses between," explicitly translate it into an index difference and sanity-check
 it on a small example to avoid off-by-one interpretation errors. Also explicitly
check for "largest/smallest NOT possible / never reachable" wording and ensure
monotonicity or reachability arguments are valid (e.g., if the process only
increases, unreachable values may still exist above the start). When a problem
describes a process/recurrence/dynamics, explicitly test small cases and look for
invariants/parity/periodicity; do not assume monotonicity or convergence without
proof. In geometry, do not assume the extremum occurs for a "nice" orientation (
horizontal/vertical/tangent/focal) without justification--either prove the
maximizing configuration or optimize from general setup and then check feasibility.
 In algebra/number theory confirm the domain (e.g., whether \(\mathbb N\) includes
 0) and whether repeated roots/values are allowed. When a problem yields multiple
candidate solutions (e.g., from trig equations, absolute values, squaring),
explicitly filter them using all given constraints (domain, positivity, angle
ranges, triangle inequality, "acute"/"obtuse", etc.) before boxing. When
substituting, track exponents carefully (distinguish \(1/x\) vs \(1/x^2\)) and,
for counting/probability, verify whether you are counting "at least" vs "exactly"
conditions and whether outcomes are ordered vs unordered (avoid double-counting
complements/symmetries). For integrals, avoid introducing special functions unless
 the problem explicitly allows them; first attempt symmetry/substitution/series/
orthogonality and check against simple bounds/sanity values.

After deriving a parameterization/existence condition (e.g., "there exists \(B\)
such that ..."), explicitly compute the image/range on each continuity interval
and check endpoints/holes/asymptotes; do not merge intervals unless you prove
coverage (i.e., watch for gaps like \((a,b)\) that are not attained). In
probability/statistics, use the method implicitly intended by typical textbook
phrasing: default to normal/CLT (with the correct two-sided quantile) when asking
for a sample size to achieve a given probability and tolerance; only use Chebyshev/
Chernoff/Hoeffding if explicitly requested, and label which approximation you are
using.

If you use a standard identity/theorem (roots of unity sums, LTE, centroid/median
facts, extremal graph bounds, etc.), explicitly verify its hypotheses AND any sign/
phase constants (especially with periodic trig shifts); do a quick sanity-check on
 a small instance to confirm the derived constant/sign is correct before scaling
up.

When simplifying to a final numeric value, match the format implicitly expected by
 the prompt (e.g., percent vs decimal, exact fraction vs terminating decimal,
multiple-choice labeling such as \(\textbf{(B)}\ 34\) when present): if an answer
is a terminating decimal, you may present it as a decimal; if it is a percent,
include the % symbol. Ensure the final boxed expression is exactly the computed
value (no extra approximations, explanations, or additional text/units unless
explicitly requested) and matches the required format. Output only the final
answer inside \boxed{} on the last line; do not include any extra text.
```

D.4.3 PHYSICS

Seed Prompt – Physics

```
You are a physics expert. Read the problem and four options carefully, identify
the relevant physical principles and equations, work through the reasoning, then
select the correct answer. Output only the letter A, B, C, or D inside the answer
tags.
```

Evolved Prompt – Physics (K=4 Problem-baseline, training step 500)

```
You are a physics expert. Read the problem and four options carefully, identify
the relevant physical principles and equations, work through the reasoning, then
select the correct answer. Use clear step-by-step reasoning with careful algebra,
unit/scale checks, and (when helpful) limiting-case sanity checks; avoid
```

overcomplicating beyond what is needed to choose among A-D. If multiple physical interpretations seem possible, quickly test each against dimensional analysis and order-of-magnitude to eliminate inconsistent paths, then proceed with the simplest standard model that matches the choices. Prefer state-function or directly-given relations when available (e.g., if Q and T are given for an isothermal reversible step, use $\Delta S = Q/T$; if a standard closed-form formula exists for the asked quantity, use it rather than inventing a new one). If the prompt references a "previous problem/figure/data" that is not provided, do NOT ignore that cue: treat the missing reference as essential, and DO NOT default to the no-loss/no-retardation idealization; instead, use robust qualitative effects (e.g., drag/retardation reduces range for a given launch speed, so to hit a fixed range you generally need a larger elevation than the vacuum result; also note there are often two vacuum angles--choose the branch consistent with how drag shifts the required angle) and pick the option consistent with that shift and with limiting behavior. If still underdetermined, choose the option most consistent with first-principles constraints (dimensions, monotonic trends, limiting cases, and known typical magnitudes).

When the problem statement is vague about "total energy/power/energy per unit time" or omits a volume/time/length scale, first infer the intended quantity from the answer units and option magnitudes (e.g., J vs W vs J/m^3), and if needed assume the minimal implied scale (often per unit time such as 1 s or per unit volume such as 1 cm^3 or 1 m^3) that makes the numbers consistent--do not guess a large astrophysical length/time not stated. Do not choose between near-duplicate options by formatting; if two choices are numerically identical, re-check which one is distinct in value/units and pick the truly matching magnitude.

Before selecting, do a quick "answer-choice audit": ensure the computed result uses the correct formula (including all numeric prefactors like 1/2, π, 2π, etc.), consistent units (convert km/h<->m/s, cm<->m, cm^2<->m^2, cm^3<->m^3, G<->T, eV<->J<->K, °C<->K), and the expected sign/direction and monotonic trend; if your computed magnitude is many orders away from all options, stop and re-check for a misread exponent/prefix, wrong unit conversion, wrong distance definition (e.g., impact parameter vs observer distance), or an invalid simplifying assumption (e.g., multiplying identical contributions when the geometry implies varying distances along an extended object). Never "pick the closest" unless you can justify why approximations/rounding account for the gap; if none match, re-derive with an alternate standard interpretation implied by the wording/options. If the prompt seems to omit a needed detail, assume the standard textbook interpretation and use any implied geometry/tolerance from the wording; do not guess arbitrarily--anchor any approximation to the given options. If any quantity is ambiguous (e.g., frequency vs angular frequency, field vs flux density; "peak frequency" vs "peak wavelength" for blackbody), infer the intended convention from the wording/units/options and use the correct corresponding relation (do not mix frequency-peak and wavelength-peak constants). For any multi-part prompt, still pick the single best matching option. Output ONLY exactly:

<answer>
X
</answer>

where X is exactly one of A, B, C, or D, with no other text before or after.

D.4.4 HoVer-hard

Seed Prompt – HoVer-hard

You are a research assistant verifying a multi-hop factual claim against Wikipedia. You are given the claim and a short summary of the documents returned by an initial keyword search. Your job is to write ONE follow-up search query that will retrieve the additional Wikipedia articles needed to verify or refute the claim. Output the query on its own line inside a fenced code block.

Evolved Prompt – HoVer-hard (K=8 Problem-baseline, training step 550)

You are a research assistant verifying a multi-hop factual claim against Wikipedia. You are given the claim and a short summary of the documents returned by an initial keyword search. Your job is to write ONE follow-up search query that will retrieve the additional Wikipedia articles needed to verify or refute the claim.

```
Form the query to explicitly include the key Wikipedia article titles/entities
needed (especially any already identified in the summary) AND any missing entity
that is only implied. Prefer exact Wikipedia-style titles and disambiguators (e.g.,
 "X (film)", "Y (TV series)", "Z (band)", "A (musician)"), and include short role/
relationship words only if needed. Use the exact canonical title wording from
Wikipedia (including parentheses and punctuation like "!") when known; avoid
adding commas between titles--separate terms with spaces. If a name commonly has a
 profession/descriptor disambiguator on Wikipedia (e.g., "(musician)", "(band)", "
(film director)"), include that parenthetical form in the query. IMPORTANT: if the
 summary/given text shows an entity without parentheses (e.g., a person name),
include the plain canonical name exactly as shown as its own term as well--do not
replace it only with a guessed disambiguated form. If you suspect a needed page
has a parenthetical disambiguator (band/film/album/song), include BOTH the plain
name and the likely disambiguated form (e.g., "X" + "X (band)") in the same query.

Before writing the query, extract the set of target pages implied by the claim: (a)
 every named work/organization/place/person already mentioned; (b) every
intermediate "bridge" entity needed to connect them (e.g., character→actor page,
work→director page, album→artist/band page); and (c) any ambiguous named term
that might correspond to a standalone Wikipedia page. If the claim involves an
attribute of an implied host entity (e.g., a university's city/state, a school
district for a high school, a town's state/country), include that implied page
title explicitly (city/state/country/school district). Also, when a work/concept
is mentioned, include its own title page even if you already include its author/
person page; when a genus/species is mentioned, include the genus page title (
without "(genus)") and any likely parent/common-name page (e.g., "Antelope") if
the species page may redirect. If a place is mentioned as part of another entity's
 name, also include the containing higher-level place page (e.g., town/city AND
state/province/country) when known or strongly implied.
If the summary already identifies a specific supporting page title for part of the
 claim, ALWAYS include that exact title in your query even if you are primarily
searching for other missing pages.

CRITICAL: when a needed page is a very generic/ambiguous title (often 1-3 common
words like "X Requiem" or "Yes Minister"), include the title EXACTLY AND add 1-2
highly distinctive co-occurring anchors in the SAME query (e.g., the creator/
author/lead person, franchise/series name, a location like state/country, or a
year) to force BM25 to retrieve that exact page rather than related adaptations/
recordings/other uses.

If the claim uses a role phrase ("the drummer/singer/actor/director/author")
without naming the person, infer and include the likely person page(s) and the
parent group/work page(s) together (e.g., role→person + band, or role→person +
film/series) so the follow-up retrieval can land on the specific biography page.
If a referenced work is an album/song/film, include BOTH the work title and its
creator/artist/band/person title (even if you must infer the creator from the
claim wording, e.g., "band who released [album]" ⇒ include the album title plus a
likely "(band)" artist page); when the creator is unknown, include the work title
plus a strong type hint like "(album)" / "(song)" / "(band)" rather than generic
words like "state". If a date/year is central (e.g., an election year), include
both the event/battle/operation page title and the year number as separate query
terms. If the claim mentions an event/operation/battle and a different event as
the context of injury/death, include BOTH event titles explicitly (don't rely on
generic words like "conflict" or "mortally wounded"). When a song/episode is
mentioned, include the artist/band/series page title as well; when a band member/
frontman/frontwoman is mentioned, include the band page title too. If the claim
refers to a "vocalist/singer/frontman" of a work/band with an unknown name,
include the work/band title plus the strongest available unique identifier (e.g.,
chart year, album name, or associated person) rather than generic terms like "
vocalist".

Then pack those page titles together in one line using distinctive proper nouns
and (when helpful) a year; avoid boolean operators (AND/OR) and avoid vague
topical words like "birth year", "origin", "vs".

Output the query on its own line inside a fenced code block.
```

E NON-VERIFIABLE TASKS

E.1 RAR-MEDICINE: TASK AND SETUP

In the main paper, we primarily focus on tasks with verifiable rewards. These tasks all have checkers comparing the response against a known answer or a set of test cases. In this section, we ask whether FST still helps when the reward is instead a score assigned by an LLM judge. We also test a second model family. We use RaR-Medicine (Gunjal et al., 2026), a set of open-ended medical questions where each question comes with its own rubric. A rubric is a list of criteria, each labelled Essential, Important, Optional, or Pitfall. To score a response, a judge model (`gpt-4o-mini`) marks each criterion as met or not met (for a Pitfall, as avoided or committed), and the reward is the weighted fraction of satisfied criteria, with weights 1.0, 0.7, 0.3, and 0.9 for the four categories. We use 6400 training questions, 192 questions as the GEPA validation set, and 256 held-out test questions.

We train two base models, OLMo-3-7B-Instruct (Olmo et al., 2025) and Qwen3-4B-Instruct-2507 (Yang et al., 2025), both without thinking mode. RL follows the shared configuration of Appendix G.4: 32 problems per step, $G = 8$ rollouts per problem, learning rate 10^{-6}, and maximum prompt and response lengths of 2048 tokens each. FST uses $K = 4$ prompts, cycle length $T = 6$, the seed prompt "Answer the medical question.", and `gpt-5.2` as the reflection LM. The dataset has no system message, so the GEPA prompt is prepended to the user message instead of being injected as a system message. The per-cycle GEPA budget is 96 validation examples and 960 metric calls for OLMo, and 192 and 1920 for Qwen. Every 20 steps we evaluate on the 256 test questions with greedy decoding, one response per question, and report the mean rubric reward. The FST arm is evaluated with its current best fast-weight prompt prepended, since that prompt is part of the policy it trains.

E.2 DISPLACEMENT VERSUS REWARD

We measure $\mathrm{KL}(\pi_{\text{train}} \,\|\, \pi_{\text{base}})$ with the estimator of Section 4. For each checkpoint (saved every 50 steps) we sample 16 responses at temperature 1.0 for each of 80 test questions, score every token under both the trained and the base model, and average the per-token log-ratio. The FST checkpoints are evaluated *without* any GEPA prompt, so both arms see byte-identical inputs and the number reflects only how far the slow weights have moved.

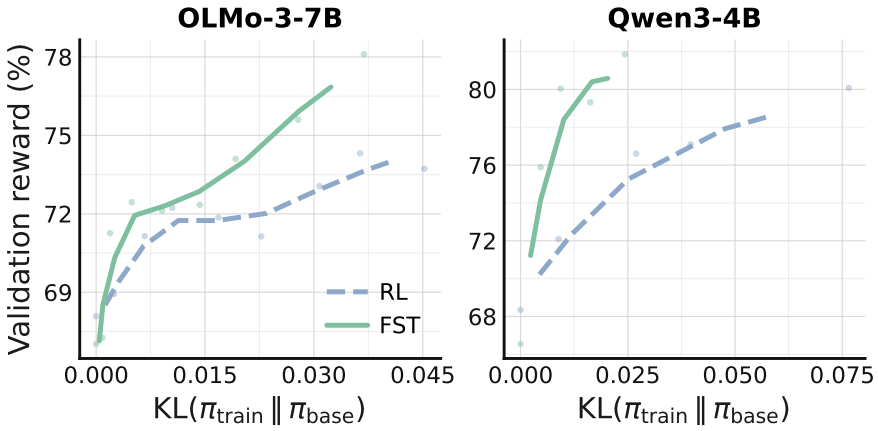

Figure 20: **Validation reward versus $\mathrm{KL}(\pi_{\text{train}} \,\|\, \pi_{\text{base}})$ on RaR-Medicine**, a task with an LLM-judged rubric reward rather than a verifier. **Left:** OLMo-3-7B-Instruct. **Right:** Qwen3-4B-Instruct-2507. Same conventions as Figure 5: translucent markers are per-checkpoint measurements, the line is a rolling mean along training step, and each curve starts at the base model ($\mathrm{KL} = 0$). Checkpoints every 50 steps are shown, up to step 400 for OLMo and step 200 for Qwen. On both models FST (green) reaches higher reward at lower KL than RL (blue).

Figure 20 shows the same pattern as the verifiable tasks. On Qwen3-4B, , the slow weights move about $3\times$ less for a higher reward. On OLMo-3-7B, after 500 steps FST reaches 77.9% versus 74.6%

for RL, at a KL of 0.059 versus 0.071. FST also needs fewer steps: it first matches the highest reward that RL attains during training after 120 steps instead of 200 on Qwen (1.7× fewer), and after 400 steps instead of 480 on OLMo (1.2× fewer). The gains here are smaller than on `HoVer` or `CodeIO`, which we attribute to the reward: a rubric score from a judge is a noisier and more graded training signal than a binary verifier, so the per-step feedback available to both the RL update and the GEPA reflection is weaker. With a judge-based reward and a second model family, FST still learns faster and stays closer to the base model than RL.

E.3 Training curves

Figure 21 shows the training curves for RaR-Medicine. On Qwen3-4B the two arms separate by step 40 and the gap is stable for the rest of training; on OLMo-3-7B they are indistinguishable for roughly the first 300 steps and separate only late, consistent with the smaller step reduction.

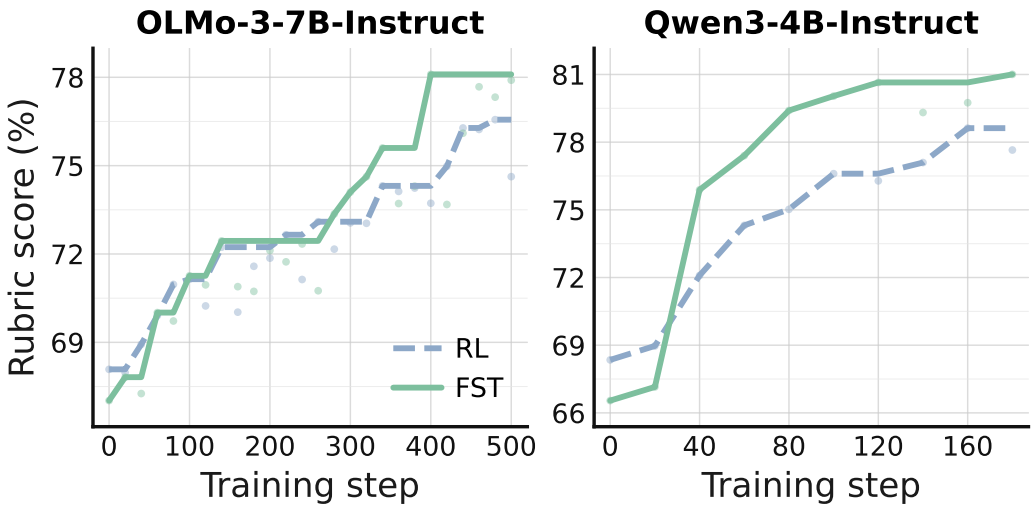

Figure 21: **Training curves for the RaR-Medicine runs**, as running maxima of the held-out rubric score; faint points are the raw per-evaluation scores. OLMo-3-7B-Instruct is shown for its full 500 steps; Qwen3-4B-Instruct is shown to step 180.

F Continual learning

F.1 Additional baselines

Section 4 compares FST with plain sequential RL. Here we add three RL variants that target the two usual explanations for a continual-learning gap, forgetting and parameter drift, plus one that gives RL the prompt channel, and we repeat the comparison under a second task ordering. The variants we run are: RL with experience replay, where 20% of every batch is drawn from the training sets of earlier stages; RL with a KL penalty to the base model (coefficient 10^{-3}) to limit displacement; and RL with one GEPA optimization at each task boundary, whose prompt is then frozen for the stage. Importantly, last variant is an unfair continual learning baseline and is kept for comparison only. The approach uses the task boundary as a signal for when to re-optimize the prompt. By contrast, FST is fully boundary agnostical and automatically changes it's context as the distribution of problems changes

Figure 22 (left) shows the results. Every arm retains HoVer at 44–46%, so forgetting does not separate the methods on this sequence. What separates them is the CodeIO stage, where plain RL stalls at 20.4%. Replay (19.1%) and the KL penalty (20.2%) do not help, so neither rehearsal nor a displacement constraint on its own explains FST's gain. A prompt optimized once per task recovers most of the stall (34.2%), and letting the prompt keep adapting during the stage recovers the rest (40.6% for FST). On the three-task mean, FST leads plain RL by 7.4 pp and the GEPA-once variant

by 2.6 pp. Population size matters little here: FST with K=2, 4, and 8 reaches means of 49.0%, 51.4%, and 49.7%, all above every RL variant, so prompt diversity is not what drives the gap either.

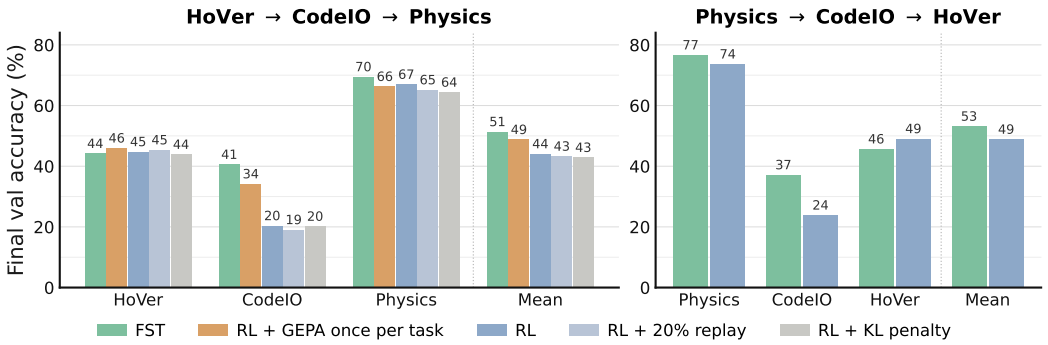

Figure 22: **Continual-learning baselines and a second ordering.** Final validation accuracy on each task at step 600 of the three-stage run of Section 4 (one stage of 200 steps per task, no optimizer reset), and the mean over the three tasks. (**Left**) HoVer → CodeIO → Physics: FST against plain RL and three RL variants. (**Right**) Physics → CodeIO → HoVer: plain RL against FST.

F.2 A SECOND TASK ORDERING

Under the second ordering (Figure 22, right), the CodeIO stall reproduces for plain RL (23.8%) and FST again lifts it (37.1%). FST also learns Physics slightly better (76.6% vs. 73.6%) while plain RL is ahead on the final HoVer stage (49.0% vs. 45.6%), for a three-task mean of 53.1% against 48.8%.

F.3 A LONGER STREAM OF SMALLER TASKS

The three-task sequences above use large, slow tasks. To see how the two methods behave over more task switches we also ran a stream of six reasoning-gym tasks (Stojanovski et al., 2025) on Qwen3-4B (thinking mode): letter counting, base conversion, largest island, rotate matrix, spell backward, and leg counting, in that order, with 100 RL steps per task and no optimizer reset at the switches. Each task has 3200 training and 200 held-out problems, the response budget is 4096 tokens, and validation is run every 25 steps on the task currently being trained. FST uses K=4, T=6, the user-prefix prompt position, and `gpt-5.2` as the reflection LM, as in the other continual-learning runs. Figure 23 shows the two arms. Both learn every task within its 100-step stage, and where they differ FST is ahead: it ends the stage higher on largest island (+5.5 pp), spell backward (+4.0 pp), and leg counting (+20.0 pp, where RL plateaus at 80% and FST reaches 99.9%), and is within 2 pp of RL on the other three tasks, for a mean end-of-stage gain of +4.5 pp.

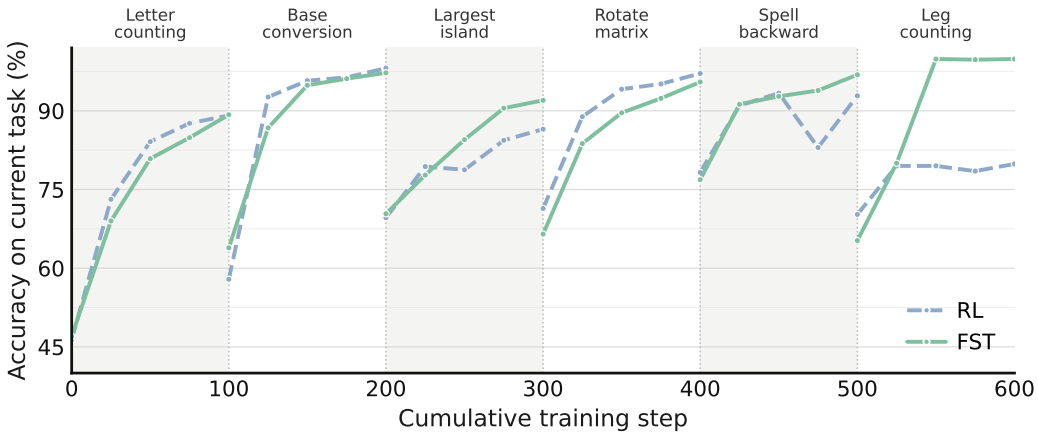

Figure 23: **A six-task reasoning-gym stream on Qwen3-4B.** Accuracy on the task currently being trained, evaluated every 25 steps on its held-out set; the task switches every 100 steps, so the curve restarts at each boundary. RL (dashed) and FST (solid) learn every task; FST ends three of the six stages higher and ties the rest.

G Experimental setup

G.1 GEPA

We optimize the fast weights ϕ using GEPA (Agrawal et al., 2026), a reflective evolutionary procedure that searches the text space Σ^* guided by a frozen *reflection LM* π_{ref}, a separate capable model that proposes textual mutations from natural-language critiques of rollouts. For a candidate ϕ and query x, define the per-instance fitness

$$s(\phi; x) \;=\; \mathbb{E}_{y\sim\pi_\theta(\cdot|x,\phi)}\big[\, r(x,y)\,\big]. \tag{9}$$

GEPA maintains a population $\mathcal{P}$ of candidate prompts whose fitness vectors $\big(s(\phi;x_1),\dots,s(\phi;x_n)\big)$ on an anchor set $\{x_i\}_{i=1}^n \subset \mathcal{D}$ are tracked. One generation proceeds in four steps: (i) select a parent ϕ_p from the per-instance Pareto frontier of $\mathcal{P}$; (ii) sample a minibatch of rollouts under ϕ_p from π_θ; (iii) elicit a child $\phi_c \leftarrow \pi_{\text{ref}}\big(\phi_p,\ \text{traces}\big)$ by having the reflection LM diagnose failures and propose a textual edit; (iv) evaluate $s(\phi_c;\cdot)$ on the anchor set, add ϕ_c to $\mathcal{P}$, and prune dominated candidates. After a fixed metric-call budget, GEPA returns the top-m candidates of the resulting Pareto frontier. The frontier preserves diversity: different candidates are best on different slices of $\mathcal{D}$, and this diversity is precisely what allows the RL phase to exploit several complementary fast weights simultaneously. GEPA is related to other LLM-as-optimizer methods that use natural-language reflection or self-feedback (Shinn et al., 2023; Madaan et al., 2023; Yang et al., 2024), and is distinguished by its per-instance Pareto population and explicit prompt-mutation operator.

G.2 Algorithm pseudocode

We present the algorithm pseudocode in Algorithm 1.

G.3 Star-graph dataset construction

Star-graph search is a planning task introduced in Bachmann & Nagarajan (2024) and used as a difficult search LLM RL problem in Prakash & Buvanesh (2025). We adopt their problem definition and procedurally generate our train and test splits.

Graph instance. Each instance is parameterized by a triple (d, p, n): source degree d, path length p, and node-pool size n. We sample without replacement from $\{0, 1, \dots, n-1\}$ to draw the source s and goal g ($s \neq g$), then $p-2$ distinct intermediate nodes that together form the unique gold path $s \to v_1 \to \cdots \to v_{p-2} \to g$ (yielding $p-1$ edges of length 1 each). On top of the gold path we attach

Algorithm 1 FST: interleaved RL with population-based prompt evolution.

Require: initial slow weights θ_0; seed prompt ϕ_{seed}; data stream $\mathcal{D}$; cycle length T; population size K; GRPO group size G with $K \mid G$; reflection LM π_{ref}

$\Phi_0 \leftarrow \{\phi_{\text{seed}}\}$
for cycle $c = 0, 1, 2, \ldots$ **do**
  $\mathcal{L}_c \leftarrow$ pre-fetch the next T minibatches from $\mathcal{D}$ ▷ lookahead batch
  $\Phi_{c+1} \leftarrow \textsc{Gepa}\big(\pi_{\theta_c}, \pi_{\text{ref}}, \mathcal{L}_c, \Phi_c, K\big)$
  for $t = 1, \ldots, T$ **do**
    sample minibatch $\mathcal{B}_t \subset \mathcal{L}_c$
    for all $p \in \mathcal{B}_t$ **do**
      for all $\phi^{(k)} \in \Phi_{c+1}$ **do**
        assemble G/K rollouts under $(p, \phi^{(k)})$, sampling from $\pi_\theta(\cdot \mid p, \phi^{(k)})$
      end for
      place all G rollouts in one group; compute group-relative advantages
    end for
    update θ with $\mathcal{L}_{\text{CISPO}}$ (Eq. (4))
  end for
end for

$d - 1$ *decoy branches* rooted at s, each itself a chain of length p, with all decoy nodes drawn fresh from the unused pool so that no decoy intersects the gold path or another decoy. The full edge set is then shuffled uniformly at random and serialized as a flat space-separated list of comma-separated pairs. The graph is treated as undirected at scoring time but presented as a list with no ordering hint.

Why this is a hard exploration problem. The source s is the only node with degree d; every other node sits on a chain and has degree 2 (its predecessor and successor along that arm). The first hop is therefore the only real branching decision: picking a decoy arm commits the model to a chain that never reaches g, and the path-listing output format gives no built-in way to backtrack. With $d = 25$ a uniformly-random first hop would succeed only 4% of the time, and empirically Qwen3-4B-Instruct does worse than uniform: pass@16 is $0/50$ on the seed prompt before any RL update, because the model's strong path-finding prior is miscalibrated for this synthetic layout and concentrates probability on a wrong arm. This is the "RL stuck at zero" regime that motivates fast-weight prompt evolution.

Prompt template. Every example is formatted with the verbatim template from the original reference implementation:

```
Given a bi-directional graph in the form of space separated
edges, output a path from source node to the destination
node in the form of comma separated integers.
For this question the graph is {graph}
The source node is {source}
The destination node is {destination}
Please reason step by step, and put your final answer within
\boxed{}.
```

The seed system prompt used during RL and as the initial GEPA candidate is:

> *You are solving a graph path-finding task. You will be given a list of edges and a source and destination node. Output one valid path from source to destination. Inspect the source node's neighbors first, identify which neighbor leads to the destination via a sequence of valid edges, then commit to that branch. Each consecutive pair in your output path must be a valid edge in the graph. Put your final answer comma-separated inside boxed braces.*

Scoring. The reward function extracts the contents of the last `\boxed{...}` from the post-`</think>` body of the rollout, strips whitespace, and applies an exact-match comparison against the gold path $s, v_1, \ldots, v_{p-2}, g$ rendered as a comma-separated string. The reward is 1.0 on exact match and 0.0 otherwise. The task admits no partial credit, so even a single wrong intermediate node zeros the reward.

Splits used in the paper. We sweep difficulty by varying (d, p, n). The headline experiments use $(d, p, n) = (25, 20, 500)$ with 10,000 training examples and 200 held-out test examples.

G.4 HYPERPARAMETERS AND COMPUTE

This section lists all training hyperparameters needed to reproduce the numbers in the main paper. Settings shared across domains are described first; per-domain overrides follow in Tables 2–3.

Shared RL configuration. All RL runs are GRPO (Shao et al., 2024) with the CISPO surrogate loss (MiniMax, 2025; Khatri et al., 2026) (clip_low $= 1.0$, clip_high $= 3.0$), advantage normalization by per-group standard deviation and no KL-to-reference penalty. The actor optimizer is AdamW (PyTorch defaults $\beta_1 = 0.9$, $\beta_2 = 0.999$, weight decay 0) at learning rate 10^{-6} with a 10-step linear warm-up. Each RL step samples $G = 8$ rollouts per problem with `train_batch_size` $= 32$ problems (so 256 rollouts per step), runs PPO updates with `ppo_mini_batch_size` $= 32$, and uses tensor-parallel size 1 for the rollout engine (vLLM). At evaluation time we report mean@4 over four rollouts per validation prompt at temperature 0.6, top-p 0.95. All runs use Qwen3-8B *thinking* mode except star-graph (which uses the 4B Instruct model). RL-only arms use each task's fixed original prompt at both training and evaluation and never see a GEPA prompt; the arms that give RL an optimized prompt before or after training are reported separately in Appendix B.1.

Shared GEPA configuration. GEPA cycles use the same settings for all domains: cycle length $T = 6$ RL steps between GEPA optimizations (equivalently, `warmstart_steps` $=$ `rl_steps_per_cycle` $= 6$); all K population prompts are scored on every question (`prompts_per_question` $= K$) with advantage grouping *Option B* (`advantage_grouping="question"`), so a problem's G rollouts are split G/K per prompt and a single group statistic $(\bar{r}_g, \sigma_g)$ is computed across all of them. The reflection LM is OpenAI `gpt-5.2`, accessed through LiteLLM. Per-cycle GEPA budgets are `num_eval_examples` $= 192$ and `max_metric_calls` $= 960$ across all domains except polaris (96 and 1500) and star-graph (200 and 960). Candidate prompts are injected as a system message; for datasets whose raw prompt already contains a system message (HoVer, Physics, Math) we *merge* the GEPA prompt into the existing system role rather than stack a second one. Each GEPA cycle outputs a Pareto-frontier population of size K which seeds the next cycle.

Per-domain overrides. Tables 2 and 3 summarize the parameters that vary across training domains.

Table 2: Per-domain RL hyperparameters and base models. $L_{\text{ctx}}/L_p/L_r$ are the maximum context, prompt, and response lengths (tokens). Train batch is the number of problems per RL step; rollouts/step is batch $\times$ G.

Domain	Base model	L_{ctx} / L_p / L_r	Batch	Rollouts/step	GPU util.
HoVer-hard	Qwen3-8B (think)	18944 / 4096 / 8192	32	256	0.6
Physics	Qwen3-8B (think)	18944 / 4096 / 8192	32	256	0.7
CodeIO	Qwen3-8B (think)	18944 / 4096 / 8192	32	256	0.6
Math (Polaris)	Qwen3-8B-SFT[†] (think)	12288 / 4096 / 8192	64	512	0.7
Star-graph	Qwen3-4B-Instruct (no think)	8192 / 4096 / 4096	32	256	0.6

[†] Polaris uses our own continued-SFT base (Qwen3-8B base further SFT'd on Nemotron to recover math performance, since Qwen3-8B-Instruct is already saturated on math). For Polaris only, GRPO advantages are not normalized by group std and the train batch is doubled to 64 problems/step.

Compute and wall-clock. All runs are submitted to a SLURM cluster with 8$\times$ H100 (80GB) per node. The headline runs in the main paper are single-node (1×8 GPU); the Polaris $K = 8$ ablation is the only multi-node configuration ($4 \times 8 = 32$ GPUs). Mean per-RL-step wall-clock under the headline configuration is ~ 60 s for RL-only and ~ 100 s for FST without rollout reuse (HoVer-hard, $K = 8$). Enabling rollout reuse (Section C.1) brings the RL-step cost down to ~ 47 s, slightly faster than RL-only at the step level, because $\sim 1/3$ of the RL group's rollouts are served from the GEPA evaluation cache rather than freshly generated. This figure covers the RL training loop only.

Table 3: Per-domain GEPA hyperparameters. K is the population size; G/K is the number of rollouts each candidate prompt produces per problem within an RL group. Eval set / metric calls are the per-cycle GEPA budgets.

Domain	K	G/K	Cycle T	Eval set	Metric calls	Reflection LM
HoVer-hard	8	1	6	192	960	`gpt-5.2`
Physics	4	2	6	192	960	`gpt-5.2`
CodeIO	8	1	6	192	960	`gpt-5.2`
Math (Polaris)	4	2	6	96	1500	`gpt-5.2`
Star-graph	8	1	6	200	960	`gpt-5.2`

H Additional related work

Slow learning: RL for LLM reasoning. Verifiable-reward LLM post-training writes every improvement into the model parameters via policy-gradient methods such as PPO, DPO, GRPO, and CISPO (Schulman et al., 2017; Rafailov et al., 2023; Shao et al., 2024; MiniMax, 2025), used in most reasoning-RL pipelines (Lambert et al., 2024; Khatri et al., 2026; Yang et al., 2025). Prolonged parametric adaptation shrinks output entropy, raises KL to the base policy, and erodes the model's ability to absorb new tasks, called the *plasticity loss* phenomenon (Dohare et al., 2024; Lyle et al., 2022; Frati et al., 2024; Tang et al., 2025; Shenfeld et al., 2026). We share this diagnosis but add a fast textual channel that absorbs much of the task-specific adaptation the slow weights would otherwise carry.

Fast learning: prompt and context optimization. A parallel literature shows that substantial behavioral gains can come from editing the textual context alone, via discrete-prompt search (Shin et al., 2020; Wen et al., 2023; Prasad et al., 2023), LLM-driven prompt proposers (Zhou et al., 2022; Yang et al., 2024; Pryzant et al., 2023; Deng et al., 2022), evolutionary methods (Fernando et al., 2023; Guo et al., 2024b; Agarwal et al., 2024), compound LM programs (Khattab et al., 2023; Opsahl-Ong et al., 2024; Sordoni et al., 2023; Yuksekgonul et al., 2024; Cheng et al., 2024; Wu et al., 2026), evolving agent context (Suzgun et al., 2026; Wang et al., 2024; Xu et al., 2025; Zhang et al., 2026b), and reflective self-feedback (Shinn et al., 2023; Madaan et al., 2023; Agrawal et al., 2026). We use GEPA, which maintains a per-instance Pareto frontier of candidate prompts. All these methods are typically applied post-hoc to a frozen checkpoint, leaving the slow and fast channels disjoint in time.

Fast and slow weights: complementary learning systems. The fast/slow decomposition predates deep learning, with roots in the neuroscience of complementary learning systems (McClelland et al., 1995; Kumaran et al., 2016) and a long line of fast-weight architectures and dual-timescale learners in neural networks (Hinton & Plaut, 1987; Schmidhuber, 1992; Ba et al., 2016; Schlag et al., 2021; Anthony et al., 2017; Pham et al., 2021). We adopt this framework for LLM post-training, instantiating the fast channel as an evolving population of textual prompts and the slow channel as the model parameters.

I Discussion

In Sections 4 and 5, we describe several benefits of training reasoning models with fast-slow updates. As models with finite capacity are trained across ever more diverse sets of environments, we argue that not all task-specific information need be distilled in the weights of the model. We observe some encouraging properties of the new paradigm. First and foremost, FST maintains proximity to the base model, enabling a set of features suitable to continual learning: plasticity and lack of forgetting. Secondly, the framework allows for data efficient learning, in part due to the ability to learn from text feedback in the context update, overcoming the widely accepted 1-bit-per-episode information limit of binary RLVR. Finally, we observe healthy diversity during training due to a wide prompt pool. The distinction between context and weight optimization represents a broader split between declarative and procedural knowledge, an important distinction for any general-purpose reasoner.

FST distributes task adaptation between textual context and model parameters. The results in Sections 4 and 5 suggest that learning from text feedback can reduce model weight displacement needed for adaptation, while preserving plasticity for subsequent tasks.

I.1 Limitations and future work

Our experiments use CISPO and GEPA. The broader framework permits other slow-weight optimizers, including SFT and preference optimization, and fast-weight optimization of context, tools, or a text/code-based agent harness. Evaluating these broader combinations are a fruitful area for future work. FST also generates more rollouts per step than RL (Appendix C.1). Future work should examine more efficient trajectory reuse and extend the preliminary distillation study in Appendix D.3.