

Reflective Context Learning: Studying the Optimization Primitives of Context Space

Nikita Vassilyev, William Berrios, Ruowang Zhang, Bo Han, Douwe Kiela, Shikib Mehri
Contextual AI

Abstract

Generally capable agents must learn from experience in ways that generalize across tasks and environments. The fundamental problems of learning, including credit assignment, overfitting, forgetting, local optima, and high-variance learning signals, persist whether the learned object lies in parameter space or context space. While these challenges are well understood in classical machine learning optimization, they remain underexplored in context space, leading current methods to be fragmented and ad hoc. We present Reflective Context Learning (RCL), a unified framework for agents that learn through repeated interaction, reflection on behavior and failure modes, and iterative updates to context. In RCL, reflection converts trajectories and current context into a directional update signal analogous to gradients, while mutation applies that signal to improve future behavior in context space. We recast recent context-optimization approaches as instances of this shared learning problem and systematically extend them with six classical optimization primitives: batching, grouped rollouts, improved credit-assignment signal, auxiliary losses, failure replay, and optimizer state. On AppWorld, BrowseComp+, and RewardBench2, these primitives improve over strong baselines, with their relative importance shifting across task regimes. We further analyze robustness to initialization, batching dynamics, and the impact of allocating stronger or weaker models to different optimization components. Our results suggest that learning through context updates should be treated not as a set of isolated algorithms, but as an optimization problem whose mechanisms can be studied systematically and improved through transferable principles.

1 Introduction

Generally capable agents must learn from experience in ways that generalize across tasks and environments. No fixed policy, however capable at training time, can anticipate every environment, user preference, or edge case it will meet in deployment, so adaptation from online experience is a prerequisite for robust, general-purpose agents (Sutton & Barto, 2018; Fang et al., 2025). The standard approach is to update model weights, but this requires expensive retraining, risks catastrophic forgetting, and is difficult to apply continuously during operation. An increasingly viable alternative is to optimize the agent’s *context* instead: the interpretable artifacts that shape behavior at inference time, such as prompts, structured playbooks, persistent memory, and behavioral guidelines (Mei et al., 2025). Context can be optimized iteratively — through reflection on execution experience rather than backpropagation over a loss surface — requiring no gradient computation, applying continuously as experience accumulates, and producing human-readable artifacts that can be inspected and debugged. As models become stronger reasoners and more faithful instruction followers, an optimizer that reasons about agent behavior can produce increasingly targeted revisions to the agent’s operational policy (Wang et al., 2023; Li et al., 2025).

The fundamental difficulties of learning do not disappear when the learned object moves from weights to context. Local optima and overfitting under greedy improvement (Mitchell, 1980), high-variance updates from limited or noisy samples (Williams, 1992; Greensmith et al., 2004), catastrophic forgetting from sequential updates (McCloskey & Cohen, 1989;

Kirkpatrick et al., 2017), and inefficient use of sparse, delayed feedback (Sutton & Barto, 2018) are properties of iterative optimization under partial information, not of any particular optimization medium. As we demonstrate, these failure modes arise equally when the optimization target is discrete context rather than continuous weights. The classical remedies — batching for variance reduction, replay for forgetting, momentum for stability, structural regularization for generalization (Kingma & Ba, 2015; Schaul et al., 2016; Bengio et al., 2009) — are motivated by properties of learning itself, not by properties of any particular optimization substrate, and they transfer to context space when appropriately adapted.

A growing number of methods have begun developing variants of a shared loop — (i) *attempt tasks*, (ii) *reflect on outcomes*, (iii) *update context* — introducing mechanisms such as batched textual gradients (Pryzant et al., 2023), modular credit assignment (Yuksekgonul et al., 2024), persistent structured memory (Zhang et al., 2026; Suzgun et al., 2026), verbal self-critique (Shinn et al., 2023), grouped rollouts with contrastive advantage (Cai et al., 2025), evolutionary search (Agrawal et al., 2026), optimizer-state retention (Yan et al., 2025), and sampling-based momentum (Ding et al., 2025). Yet these methods were introduced across a period of rapid change in models, prompting practices, and benchmarks, making it difficult to isolate the contribution of a learning primitive from the effects of model capability, prompt engineering, and task difficulty. The problem is compounded by a regime shift: early prompt optimization was largely concerned with conjuring the right tokens to elicit latent capabilities from imperfect instruction followers (Pryzant et al., 2023; Zhou et al., 2023), whereas current models are sophisticated enough that context updates can encode genuine strategies learned from ground-truth rewards and execution experience (Trivedi et al., 2024; Chen et al., 2025). In this regime, the bottleneck moves from search breadth to reflection quality, and which primitives matter, and how they compose, becomes urgent.

These observations motivate treating context-space adaptation not as a collection of isolated algorithms but as a single optimization problem that can be studied systematically. We formalize this view as **Reflective Context Learning (RCL)**, a framework centered on reflection as the core learning mechanism. RCL constrains the design space to the setting we believe is most practically viable and future-proof: strong reasoning models serving as the optimizer, structured playbook representations as the learned artifact, and iterative, SGD-like updates driven by reflective diagnosis of execution trajectories. Within this setting, we study which classical optimization primitives transfer to context space, how they compose, and how their relative importance shifts across task regimes — varying how the update signal is computed, how it is applied, and how optimizer state is managed across iterations. We do not claim to introduce the forward-reflect-update loop itself. ProTeGi (Pryzant et al., 2023) and TextGrad (Yuksekgonul et al., 2024) formalized gradient-like textual feedback, and ACE (Zhang et al., 2026) developed structured playbook artifacts for iterative agent improvement. Our contribution is to identify the shared structure these methods converge on, systematically study how classical optimization primitives compose in context space under controlled conditions, and show that the relative importance of these primitives shifts across task regimes and that standalone gains do not predict a primitive’s contribution under composition. Across AppWorld (Trivedi et al., 2024), BrowseComp+ (Chen et al., 2025), and RewardBench2 (Malik et al., 2025), reflection-quality primitives yield the most consistent gains, and training dynamics exhibit qualitative patterns — variance-induced oscillation, momentum-stabilized convergence — analogous to their parameter-space counterparts.

2 Reflective Context Learning

2.1 Problem Setting and Formulation

We consider a setting in which an agent $\mathcal{A}$, implemented as a ReAct loop (Yao et al., 2023) over a base language model, executes tasks by conditioning on both an input x (a query, environment state, or task specification) and a context artifact $\mathcal{C}$. The context artifact is the collection of all interpretable, externalized components that influence the agent’s behavior at inference time: structured playbooks of behavioral rules, persistent memory entries, tool definitions, retrieval indices, and operational guidelines (Mei et al., 2025). $\mathcal{C}$ is a superset of what is typically called a “prompt”: it includes any artifact the agent can access during

Table 1: Functional correspondence between classical gradient-based learning and reflective context learning, with representative prior work that explored each context-space analogue.

Classical Concept	RCL Analogue	Prior Work
Parameters θ	Context artifact $\mathcal{C}$ (playbook, memory, tools, guidelines)	All methods below
Forward pass $\hat{y} = f_\theta(x)$	Trajectory $\tau = \mathcal{A}(x, \mathcal{C})$	ReAct (Yao et al., 2023); Voyager (Wang et al., 2023)
Loss $\mathcal{L}(\hat{y}, y^*)$	Outcome signal $r = \mathcal{R}(\tau, y^*)$	—
Gradient $\nabla_\theta \mathcal{L}$	Reflective diagnostic $\Delta = g(\tau, r, \mathcal{C})$	ProTeGi (Pryzant et al., 2023); TextGrad (Yuksekonul et al., 2024)
Optimizer step	Context update $\mathcal{C} \leftarrow f(\mathcal{C}, \Delta)$	ACE (Zhang et al., 2026)
Minibatch $\mathcal{B} \subset \mathcal{D}$	Trajectory batch per reflection step	ProTeGi (Pryzant et al., 2023); TF-GRPO (Cai et al., 2025)
Momentum / Adam	Optimizer history + mutation rationales	ERM (Yan et al., 2025); Ding et al. (2025)
Replay buffer	Failure replay over historical hard cases	Dyn. Cheatsheet (Suzgun et al., 2026); ExpeL (Zhao et al., 2024)
Architecture choice	Context param. (flat vs. structured)	ACE (Zhang et al., 2026); DSPy (Khatab et al., 2024)
Regularization	Structural constraints (diffs, error $\rightarrow$ rule maps)	ACE (Zhang et al., 2026); MIPRO (Opsahl-Ong et al., 2024)

execution, whether injected directly into the context window or accessed dynamically through retrieval or tool use. Modern language models are specifically designed to be effective, faithful instruction followers (Ouyang et al., 2022; Anthropic, 2025), and this sensitivity to $\mathcal{C}$ is what makes context-space optimization practical: updates to $\mathcal{C}$ reliably produce corresponding changes in agent behavior.

Given a dataset of tasks $\mathcal{D} = \{(x_i, y_i^*)\}$ with ground-truth labels or a reward function $\mathcal{R}$ that evaluates trajectory quality, the learning problem is to find the context $\mathcal{C}$ that maximizes expected performance:

$$\mathcal{C}^* = \arg \max_{\mathcal{C}} \mathbb{E}_{(x, y^*) \sim \mathcal{D}} [\mathcal{R}(\mathcal{A}(x, \mathcal{C}), y^*)] \quad (1)$$

We focus on this supervised, experience-driven optimization setting throughout the paper: given a clear training signal and a collection of execution trajectories, how should the context be updated to improve future behavior?

Several prior methods have instantiated variants of this loop (Shinn et al., 2023; Pryzant et al., 2023; Yuksekonul et al., 2024; Zhang et al., 2026), reviewed in Section 2.2. We abstract over their specific implementations to define a general loop with three stages, each analogous to a stage in gradient-based training:

Forward pass (execution). Agent $\mathcal{A}$ executes a task, producing a trajectory and outcome:

$$\tau = \mathcal{A}(x, \mathcal{C}_t), \quad r = \mathcal{R}(\tau, y^*) \quad (2)$$

The trajectory τ is a sequence of actions, observations, and intermediate reasoning steps. The outcome r may be binary, scalar, or a structured execution trace. In classical learning, this corresponds to the forward pass $\hat{y} = f_\theta(x)$ and loss computation $\mathcal{L}(\hat{y}, y^*)$.

Backward pass (reflection). The reflector g takes the trajectory, its outcome, and the current context as input and produces a structured diagnostic signal:

$$\Delta = g(\tau, r, \mathcal{C}_t) \quad (3)$$

The diagnostic Δ is a natural-language analysis of what failed, why, and what components of C_t should be revised. This is the context-space analogue of gradient computation $\nabla_{\theta}\mathcal{L}$: converting execution experience into a directed update signal. The key difference is that Δ is produced by LLM reasoning over trajectories rather than by differentiating through a computation graph; the reflector may be a separate, potentially stronger model.

Optimizer step (mutation). The mutator f takes the current context and the diagnostic signal and produces an updated context:

$$C_{t+1} = f(C_t, \Delta) \quad (4)$$

This is the context-space analogue of the optimizer step $\theta_{t+1} = \theta_t - \alpha \nabla \mathcal{L}$. The mutator applies the reflector’s recommendations, subject to structural constraints such as only modifying specific playbook rules or maintaining version history.

The full update, iterating over samples drawn from $\mathcal{D}$, is:

$$C_{t+1} = f(C_t, g(\tau_t, r_t, C_t)) \quad (5)$$

The analogy is functional, not formal: there is no differentiable loss surface, and Δ can be wrong or vague in ways real gradients cannot. We claim not mathematical equivalence but that the functional roles are preserved (Table 1) and that the classical mechanisms that improve gradient-based learning improve reflective context learning for the same underlying reasons. When we refer to *optimization primitives* throughout this paper, we mean mechanisms that improve the quality, stability, or efficiency of this reflective update process, which we study in Section 3.

2.2 Prior Work and the Emergence of Context-Space Learning

Context-space optimization has its roots in prompt engineering and discrete prompt tuning. Soft prompt tuning (Lester et al., 2021; Li & Liang, 2021) offered a gradient-based alternative over continuous embeddings, but operates in a space where gradient steps produce micro-level perturbations rather than structured, interpretable revisions. In-context learning (Brown et al., 2020) demonstrated that context is a powerful conditioning mechanism but involves no iterative refinement. Early discrete methods like APE (Zhou et al., 2023) introduced iteration through generate-and-score search, establishing context as an optimization target, but the update signal remained purely scalar with no diagnosis of why one candidate outperformed another.

The transition to genuine learning began with the introduction of *reflection* as the update mechanism. Reflexion (Shinn et al., 2023) demonstrated that verbal self-critique, appended to context across episodes, improves agent performance without weight updates. ProTeGi (Pryzant et al., 2023) formalized this as “textual gradients”: batched natural-language critiques derived from minibatches of failures and applied via beam search. TextGrad (Yuksekgonul et al., 2024) generalized the gradient metaphor to compound AI systems, treating multi-component pipelines as computation graphs with textual feedback propagation. Together, these methods established that context-space learning shares the structure of gradient-based optimization — reinforced by concurrent work framing the same loop as reinforcement learning without weight updates (Cai et al., 2025; Agrawal et al., 2026).

From this shared foundation, prior work has explored several directions of improvement. **Structured parameterization** enables localized credit assignment: Dynamic Cheatsheet (Suzgun et al., 2026) curates persistent memory, ACE (Zhang et al., 2026) uses structured playbooks with delta edits, and DSPy (Khattab et al., 2024) and MIPRO (Opsahl-Ong et al., 2024) optimize modular LM programs. **Variance reduction:** ProTeGi minibatches critiques across failures, and Training-Free GRPO (Cai et al., 2025) adds grouped rollouts with contrastive semantic advantages. **Optimizer state and momentum:** ERM (Yan et al., 2025) retains historical feedback, Ding et al. (2025) introduces sampling-based momentum, and OPRO (Yang et al., 2023) passes optimization history into context. **Search and frontier maintenance:** EvoPrompt (Guo et al., 2024) and PromptBreeder (Fernando et al., 2024) evolve candidate populations, and GEPA (Agrawal et al., 2026) combines population search with reflective

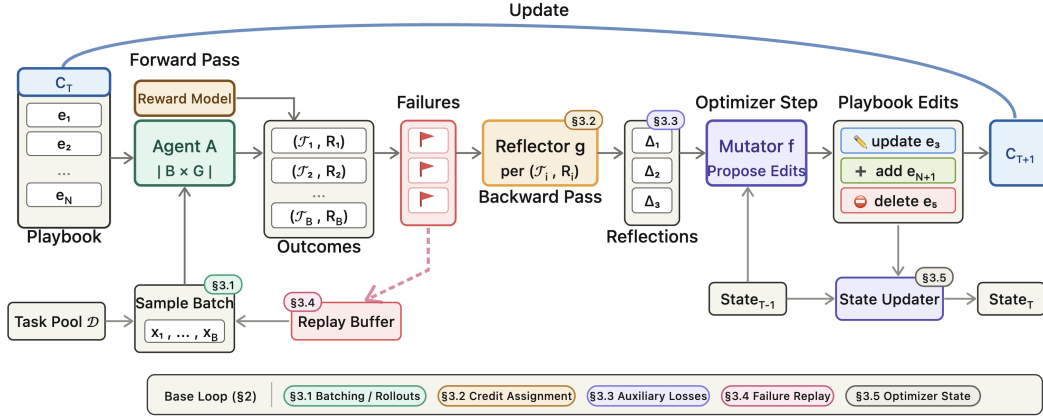


Figure 1: The RCL optimization loop with primitives mapped to stages. A batch of B tasks is sampled from $\mathcal{D}$ (with replay, §3.4), each executed G times (§3.1). Failed traces, with dual-trace annotations (§3.2), go to a multi-head reflector (§3.3) producing per-trace diagnostics Δ_i ; the mutator f turns these into structured edits to entries e_i , conditioned on a rolling optimization state document (§3.5).

diagnosis. **Policy-level learning:** ExpeL (Zhao et al., 2024) extracts reusable cross-task insights, and Agent-Pro (Zhang et al., 2024) revises behavioral guidelines.

Each direction has been explored in isolation, under different models, benchmarks, and prompting conventions, making it difficult to attribute improvements to the primitive itself; Appendix C maps each method onto the RCL loop.

3 Optimization Primitives

The core update $C_{t+1} = f(C_t, g(\tau_t, r_t, C_t))$ is a single-sample, stateless, greedy step: one trajectory informs one reflection, which produces one edit, with no memory of prior iterations and no mechanism to escape a poor basin. Applied repeatedly, this minimal loop exhibits the same pathologies as its parameter-space counterpart — high-variance updates, sparse credit assignment, catastrophic forgetting, and local optima. We introduce six primitives that address these pathologies at specific stages of the loop (Figure 1); Appendix B states each formally as an operator on Eq. 5. Section 4.2 evaluates each primitive against the ACE baseline and studies their composition.

3.1 Batching and Grouped Rollouts

A single trajectory τ_t produces a single diagnostic Δ_t whose content is dominated by the idiosyncrasies of that example — the context-space analogue of the high-variance updates that minibatching addresses in parameter-space optimization (Robbins & Monro, 1951; Pryzant et al., 2023; Cai et al., 2025). We implement two complementary axes of variance reduction. **Task batching** samples B tasks per iteration and reflects on each failed trace independently, producing per-trace diagnostics $\Delta_1, \dots, \Delta_k$ for the $k \leq B$ failures. These are passed jointly to the mutator, which identifies recurring patterns and filters one-off anomalies; when $k = 0$ no edit is made. **Grouped rollouts** execute each task G times under the same playbook C_t ; groups containing both passes and failures provide contrastive signal that lets the reflector isolate the decision points responsible for the outcome difference while controlling for task difficulty, and groups with uniform outcomes fall back to single-trace reflection (Eq. 3). This is the only setting in which the reflector receives positive traces, grounding its diagnoses in demonstrated successful behavior. Batching reduces variance across the task distribution; grouped rollouts reduce variance within each task.

3.2 Improved Credit Assignment

The outcome signal $r = \mathcal{R}(\tau, y^*)$ is terminal, leaving the reflector to attribute it across an entire trajectory and playbook with no intermediate supervision — the same sparse-reward problem that motivates step-level reward modeling (Lightman et al., 2023) and value decomposition (Sunehag et al., 2017). We address this with **dual-trace** credit assignment, in which the agent tags its own reasoning; no human or auxiliary annotators are involved. Each task is executed twice concurrently: once with the standard playbook $\mathcal{C}_t$, the only trace used for evaluation, and once with a variant that appends a behavioral suffix — marked as not part of the playbook — asking the agent to cite which entry guided each action (or none), flag uncertainty, and note where guidance was missing. These self-reports are evidence, not ground truth: the reflector cross-checks each tag against the actions actually taken, so a citation inconsistent with observed behavior is discounted rather than trusted, and cross-referenced tags then support entry-level attribution (Appendix B). The annotated trace’s outcome is discarded, because instrumentation alters behavior and makes it an unreliable measure of playbook quality; this is also why the annotated variant is not run everywhere. Composed with grouped rollouts, executions per task become $G + 1$, with the annotated trace excluded from the contrastive group.

3.3 Auxiliary Losses and Structural Inductive Biases

Without explicit structure, unconstrained reflections collapse toward surface-level trajectory retelling — analogous to the representation collapse that auxiliary objectives prevent in multi-task learning (Jaderberg et al., 2016). We impose structure at two levels. **Playbook parameterization:** $\mathcal{C}$ is organized into named sections with individually addressable entries $e_1, \dots, e_N$, forcing the mutator to express updates as localized edits (UPDATE, ADD, DELETE on individual entries) rather than holistic rewrites. **Reflection schema:** the reflector g is decomposed into three parallel diagnostic heads (Figure 1) — failure attribution (actionable gap, execution variance, or intractable), root cause analysis, and coverage gap specification — each forcing a structured classification that gates the mutator (e.g., execution-variance labels bias toward no-ops, preventing unnecessary edits from noisy signal). The correspondence to auxiliary losses is functional: keeping the reflection predictive of three sub-targets prevents collapse to narrative, while the edit-operator constraint plays the role of structural regularization.

3.4 Failure Replay and Curriculum Strategy

A single reflection-mutation cycle may not resolve a failure: the edit may be partial, may interact negatively with existing entries, or may require refinement only apparent on re-encounter in a different batch context. Experience replay buffers address analogous issues in parameter-space learning (Lin, 1992; Schaul et al., 2016); in context space, where edits can directly contradict or subsume one another, the need is at least as acute. We maintain a **failure replay buffer**: each iteration, a ratio ρ of the B sampled tasks draws from replayed failures and the remainder samples fresh from $\mathcal{D}$, ensuring edits must generalize across examples rather than overfit to one. Tasks graduate out of the buffer after consecutive passes and are evicted after prolonged failure, implementing a curriculum that concentrates optimization effort where the marginal return is highest.

3.5 Optimizer State and Momentum

A stateless optimizer may revert a change from two iterations ago because the evidence that motivated it has scrolled out of context — the analogue of the oscillation that momentum (Polyak, 1964; Kingma & Ba, 2015) was designed to prevent. We maintain a structured, rolling **optimization state document** S_t , updated by a dedicated model call after each iteration, tracking a change ledger, playbook assessment, open hypotheses, and optimization phase. S_t is injected into the mutator f but excluded from the reflector g — mirroring how momentum in Adam operates on the optimizer step rather than on gradient computation — so that diagnostics remain unbiased by the consensus of past iterations.

4 Experiments

4.1 Setup

We evaluate on three benchmarks that place different demands on the optimization loop. **AppWorld** (Trivedi et al., 2024) is a multi-step interactive coding benchmark scored by Task Goal Completion (TGC) on held-out Normal (168 tasks) and Challenge (417 tasks) splits; seed scores of 78–82% on Normal make for a narrow *seed-to-ceiling gap* — the distance between the seed score and the achievable ceiling — where gains come from correcting procedural failure modes. **BrowseComp+** (Chen et al., 2025) is a web research benchmark scored by LLM-judged accuracy on 150 held-out queries; seed scores of 29–41% make the gap wide, requiring the model to discover general search heuristics it does not yet possess. **RewardBench2** (Malik et al., 2025) is a response-ranking task scored by accuracy on a 281-example test split; seed scores of 67–76% in a near-deterministic, non-agentic environment make this a calibration problem. Training pools, validation splits, and rollout budgets are given in Appendix F.

We use two agent models: Gemini 3.1 Flash Lite (Lite) and GPT-5.4 Nano (Nano). All runs use Claude Opus 4.6 as reflector and mutator, train for 30 iterations, and are evaluated on held-out splits never seen during training. Throughout, **Seed** denotes the unoptimized starting configuration — the benchmark’s system prompt plus a small seed playbook derived from its official setup (empty for BrowseComp+; Appendix I) — which is the artifact RCL optimizes. We compare against two baselines. **ACE** (Zhang et al., 2026) is our primary baseline: the base loop of Eq. 5 with none of the primitives of Section 3 active, and the substrate every configuration here builds on. **GEPA** (Agrawal et al., 2026) is a sample-efficient prompt optimizer that pairs reflective diagnosis with genetic Pareto search over candidate prompts; we use the official DSPy implementation (Khatab et al., 2024) with `auto="light"` and mini-batch size 3, matching our batch size. Our reproduction scope for both baselines is detailed in Appendix F.

All methods share the same agent model, optimizer model, and evaluation protocol; only the primitives vary, and each benchmark uses its own seed playbook and system prompt (Appendices H–I). ACE reflects on a single failed trace per iteration ($B=1$); the Batching primitive raises this to $B=3$, and the composed RCL configuration uses $B=3$, $G=3$, and replay ratio $\rho=0.5$ across all benchmarks. These are fixed rather than swept and every run is single-seed, since one configuration is 30 iterations of long-horizon rollouts: $B=3$ and $G=3$ sit near a practical context ceiling because we pass full trajectories to the reflector and mutator, and $\rho=0.5$ balances fresh sampling against consolidation. Section 4.5 studies sensitivity to seed quality; code and run configurations are available at <https://github.com/nvassilyev/RCL> (Appendix G).

4.2 Main Results

Table 2 reports each primitive added individually to ACE, the composed RCL, and both baselines. GEPA is competitive on BrowseComp+ and RewardBench2 but falls below Seed in three of eight conditions, all on AppWorld; RCL never falls below Seed in any condition.

Reflection quality gives the best returns per compute. Optimizer state and auxiliary losses — which improve the reflector and mutator without additional executions — beat ACE on the majority of conditions across all three benchmarks: optimizer state adds +4.8 TGC on AppWorld Normal/Lite and +6.4 on BrowseComp+/Nano; auxiliary losses add +5.5 on AppWorld Challenge/Lite and +8.5 on RewardBench2/Nano. Grouped rollouts improves reliably too, but at additional execution cost. The two cheapest primitives being among the most effective makes diagnostic precision, not execution volume, the binding constraint.

Execution-side primitives must be tuned to task dynamics. Batching gains when the failure distribution is broad (+5.4 over ACE on AppWorld Normal/Lite) but can hurt when failures are diverse: on BrowseComp+/Lite it *degrades* by −6.0 relative to ACE, overloading

Table 2: **Main results.** Each primitive is added individually to the ACE baseline. AppWorld: TGC; BrowseComp+/RewardBench2: accuracy. Lite = Gemini 3.1 Flash Lite, Nano = GPT-5.4 Nano. All runs use Claude Opus; 30 training iterations. Deltas vs. Seed. Best result per column is in **bold**.

	AppWorld				BrowseComp+		RewardBench2	
	Normal		Challenge		Test Set		Test Subset	
	Lite	Nano	Lite	Nano	Lite	Nano	Lite	Nano
<i>Baselines</i>								
Seed	78.0	81.5	64.3	75.8	28.9	40.7	75.9	67.2
GEPA	82.7 ^{+4.7}	76.2 ^{-5.3}	62.6 ^{-1.7}	66.7 ^{-9.1}	35.3 ^{+6.4}	48.7 ^{+8.0}	79.2 ^{+3.3}	76.5 ^{+9.3}
ACE	83.3 ^{+5.3}	86.9 ^{+5.4}	69.1 ^{+4.8}	80.7 ^{+4.9}	37.3 ^{+8.4}	50.0 ^{+9.3}	79.2 ^{+3.3}	62.7 ^{-4.5}
<i>ACE + individual primitive</i>								
+ Failure Replay	84.5 ^{+6.5}	82.7 ^{+1.2}	73.1 ^{+8.8}	81.8 ^{+6.0}	40.0 ^{+11.1}	47.3 ^{+6.6}	80.4 ^{+4.5}	65.4 ^{-1.8}
+ Optimizer State	88.1 ^{+10.1}	84.5 ^{+3.0}	73.9 ^{+9.6}	79.4 ^{+3.6}	36.0 ^{+7.1}	56.4 ^{+15.7}	82.8 ^{+6.9}	69.9 ^{+2.7}
+ Credit Assignment	83.9 ^{+5.9}	84.5 ^{+3.0}	71.5 ^{+7.2}	79.4 ^{+3.6}	37.3 ^{+8.4}	46.3 ^{+5.6}	77.2 ^{+1.3}	63.5 ^{-3.7}
+ Grouped Rollouts	86.3 ^{+8.3}	86.8 ^{+5.3}	72.4 ^{+8.1}	81.4 ^{+5.6}	38.7 ^{+9.8}	50.7 ^{+10.0}	81.5 ^{+5.6}	77.8 ^{+10.6}
+ Batching	88.7 ^{+10.7}	87.2 ^{+5.7}	71.2 ^{+6.9}	83.4 ^{+7.6}	31.3 ^{+2.4}	48.3 ^{+7.6}	82.9 ^{+7.0}	63.6 ^{-3.6}
+ Auxiliary Losses	86.3 ^{+8.3}	89.2 ^{+7.7}	74.6 ^{+10.3}	80.6 ^{+4.8}	38.0 ^{+9.1}	54.7 ^{+14.0}	82.5 ^{+6.6}	71.2 ^{+4.0}
<i>Composed</i>								
RCL (all primitives)	89.3 ^{+11.3}	89.1 ^{+7.6}	71.9 ^{+7.6}	83.7 ^{+7.9}	37.3 ^{+8.4}	51.3 ^{+10.6}	82.4 ^{+6.5}	69.8 ^{+2.6}

the mutator with competing signal. Grouped rollouts help most where the reflector needs contrastive signal: +3.0 on AppWorld Normal/Lite and +15.1 on RewardBench2/Nano, the largest improvement over ACE in the table. Credit assignment adds modest value on AppWorld for the Lite agent, whose multi-step procedural traces benefit from entry-level attribution, but degrades Nano and adds none on BrowseComp+, where terminal feedback already localizes the problem to the search strategy. Execution-side primitives should therefore be configured to the variance structure of the target environment rather than applied uniformly.

The seed-to-ceiling gap shapes what kind of learning occurs. On AppWorld, where the gap is narrow, multiple primitives contribute: batching and optimizer state lead on Normal/Lite (+5.4 and +4.8 over ACE), auxiliary losses and optimizer state on Challenge/Lite (+5.5 and +4.8); playbooks accumulate targeted procedural rules via ADD mutations. On BrowseComp+, where the gap is wide, results are agent-dependent: optimizer state gives the largest gain for Nano (+6.4) but slightly degrades Lite (−1.3), where failure replay leads instead (+2.7), and playbooks show a high ratio of UPDATE mutations as strategies are refined over repeated encounters. On RewardBench2, over-instruction poses the greatest risk: it is a single-turn judgment task, and entries learned from procedural failure modes interfere with the naturalistic reasoning it requires. This is most visible on Nano, where ACE falls from 67.2 at seed to 62.7 (−4.5) and failure replay only partially recovers it (65.4). Contrastive signal is the effective remedy: grouped rollouts reach 77.8 (+15.1 over ACE, +10.6 over Seed), because the reflector sees correct and incorrect rankings under the same playbook and can identify which criteria are predictive.

4.3 Primitive Interactions under Composition

Composition is not additive, and standalone value does not predict compositional role. To separate the two, we perform leave-one-out ablations of the full RCL optimizer (Table 5; full analysis in Appendix D), which measure a primitive’s role once the optimizer is assembled rather than its marginal value when added to ACE. The two disagree: auxiliary losses improve 7 of 8 settings standalone, yet removing them from full RCL *improves* RewardBench2/Nano by +12.6, while credit assignment helps only 3 of 8 standalone but its removal produces the largest drop in three settings. Two primitives are consistently load-bearing: removing grouped rollouts hurts 7 of 8 settings and ties the eighth (down to −9.5 on AppWorld Normal/Lite and −11.3 on RewardBench2/Nano), and removing

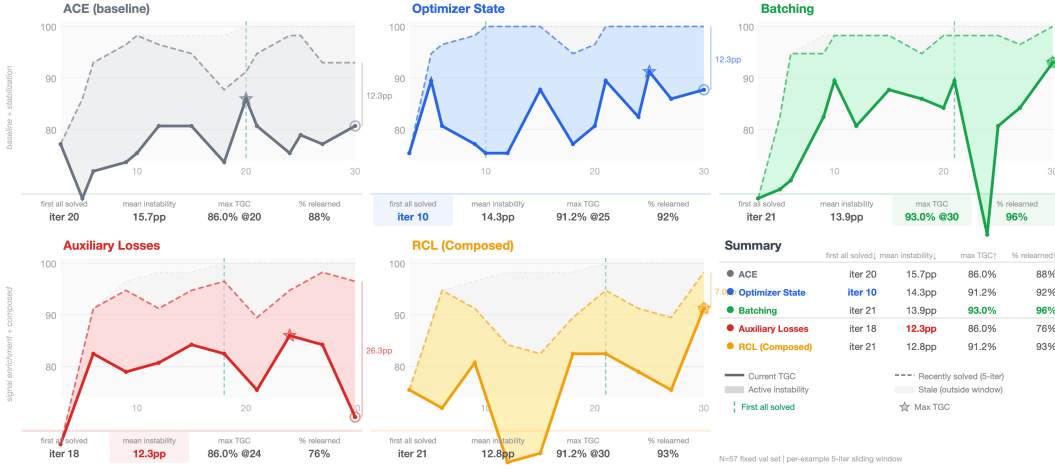


Figure 2: **Learning dynamics on AppWorld dev (Lite agent, 57 tasks).** Solid: current TGC. Dashed: recently solved rate (solved $\geq 1\times$ in the trailing 5 iterations). Colored shading: active instability; gray: stale regressions (Section 4.4). Stars: peak TGC. Green verticals: first full coverage.

failure replay causes the largest regression in the table (-18.0 on BrowseComp+/Nano). Primitive contributions are real but non-additive, depending on the task regime, the agent, and the mechanisms they combine with.

4.4 Training Dynamics

To understand how primitives affect the optimization trajectory, not just the endpoint, we track per-example solve status on a fixed 57-task AppWorld dev set at every checkpoint (Figure 2). For each run we report the current TGC alongside a *recently solved rate*: the fraction of tasks solved at least once within a trailing 5-iteration window. The gap between these curves is *active instability*; the further gap to the all-time envelope is *stale regression* (Appendix E).

All primitives eventually solve every dev task at least once, but they differ sharply in *when* they achieve complete coverage and *how much* they retain. Optimizer State reaches full coverage earliest (iteration 10) and pairs high peak TGC (91.2%) with strong relearning (92%): the rolling state document prevents the mutator from reverting useful edits, a stabilizing effect analogous to momentum. Batching gets there later (iteration 21) and oscillates more in mid-training, but reaches the highest peak TGC (93.0%) and relearning rate (96%) of any configuration. Auxiliary Losses runs the other way — the *lowest* instability (12.3pp) but the lowest relearning (76%) and a peak (86.0%) no higher than ACE — a conservative rather than exploratory dynamic. The composed RCL configuration inherits both: low instability (12.8pp) with high peak TGC (91.2%), consistent with the complementary-pathology view from Section 4.2.

4.5 Sensitivity of the Optimization Loop

We examine three design axes — initialization, optimizer model allocation, and reflection aggregation (Figure 3) — to understand how sensitive the loop is to choices outside the primitives themselves.

Seed robustness. RCL converges to 72–76 TGC from all three seed playbooks on AppWorld Challenge — empty, decent, and high-quality (Appendix J) — while ACE oscillates severely from the empty seed (44.2 at iteration 30 vs. RCL’s 72.4). The primitives’ marginal contribution scales inversely with seed quality: $+28.2$ from the empty seed, $+3.8$ from the

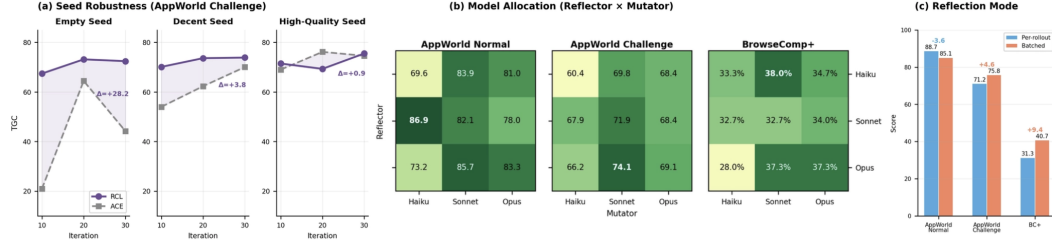


Figure 3: **Design choice analysis (Lite agent).** (a) Seed robustness on AppWorld Challenge: RCL converges to 72–76 TGC from all three seeds; ACE diverges from weaker initializations. (b) Reflector $\times$ mutator allocation; no configuration dominates uniformly. (c) Per-trace vs. batched reflection at $B=3$.

decent seed, +0.9 from the high-quality seed — consistent with the primitives addressing pathologies that are most acute far from a good basin.

Model allocation. Figure 3b varies the reflector and mutator across all nine Haiku/Sonnet/Opus combinations. Stronger reflectors tend to help on harder tasks (Opus/Sonnet reaches 74.1 on Challenge, the best of the nine allocations on that split), but the pattern is not monotonic: Haiku/Sonnet matches or exceeds several Opus-reflector configurations on AppWorld Normal and BrowseComp+, and Sonnet, not Opus, is the most consistently strong mutator. Reflector capability binds with task difficulty: a Haiku reflector already suffices on the easier and near-deterministic regimes, whereas the hardest split is where the strongest reflector pays off. Mutation rewards faithful, constrained editing; an overly capable mutator may over-interpret diagnostics rather than execute them. Matching capacity to each role’s demands matters more than uniformly maximizing capability.

Reflection aggregation. Figure 3c compares per-trace vs. batched reflection at $B=3$. Batched reflection helps on harder tasks (Challenge +4.6, BrowseComp+ +9.4) where failures share structural root causes, but degrades on AppWorld Normal (−3.6) where failures are diverse. The same distinction explains the batching result in Table 2: aggregation at the *reflector* helps BrowseComp+, while aggregation at the *mutator* hurts it (−6.0 vs. ACE) — in diverse-failure regimes the binding constraint is the mutator’s capacity to reconcile competing recommendations, not the volume of signal. We use per-trace reflection throughout, as it composes cleanly with grouped rollouts.

5 Conclusion

RCL treats context-space adaptation as an optimization problem with diagnosable pathologies, studied under controlled conditions where only the primitives vary. (1) Diagnostic precision matters more than execution volume: primitives that improve the reflection signal give the largest gains per unit of compute. (2) Which primitives help depends on the task regime, and composition is not additive: no single primitive dominates, and the full optimizer does not uniformly beat the best individual one. (3) Matching model capacity to each role matters more than maximizing it: a faithful mutator paired with a strong reflector outperforms the reverse. (4) Context-space training dynamics mirror parameter-space phenomena: oscillation, momentum-stabilized convergence, and a tradeoff between stability and relearning.

More broadly, these results suggest that context-space optimization will benefit from the same systematic discipline that classical ML brings to parameter-space learning: diagnosing pathologies, composing remedies, and studying interactions under controlled conditions. Control has a cost: our runs are single-seed and the optimizer’s hyperparameters are held fixed rather than swept (Section 4.1), so the ordering we report is established at equal compute and we do not claim it is invariant to those choices. Adaptive primitive selection, second-order state tracking, and extension to continual deployment remain open. As models grow more capable, the ceiling on what can be learned through context updates rises accordingly — making principled optimization of that learning process increasingly consequential.

References

- Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziemis, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2026. URL <https://arxiv.org/abs/2507.19457>.
- Anthropic. The claude model spec, 2025. URL <https://docs.anthropic.com/en/docs/about-claude/claude-model-spec>.
- Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *ICML*, 2009.
- 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.
- Yuzheng Cai, Siqi Cai, Yuchen Shi, Zihan Xu, Lichao Chen, Yulei Qin, Xiaoyu Tan, Gang Li, Zongyi Li, Haojia Lin, Yong Mao, Ke Li, and Xing Sun. Training-free group relative policy optimization. *arXiv preprint arXiv:2510.08191*, 2025.
- Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Hao-ran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhui Chen, and Jimmy Lin. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent, 2025. URL <https://arxiv.org/abs/2508.06600>.
- Zixin Ding, Junyuan Hong, Zhan Shi, Jiachen T. Wang, Zinan Lin, Li Yin, Meng Liu, Zhangyang Wang, and Yuxin Chen. Scaling textual gradients via sampling-based momentum. *arXiv preprint arXiv:2506.00400*, 2025.
- Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, Zhaochun Ren, Nikos Aletras, Xi Wang, Han Zhou, and Zaiqiao Meng. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems, 2025. URL <https://arxiv.org/abs/2508.07407>.
- Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *ICML*, 2024.
- Evan Greensmith, Peter L. Bartlett, and Jonathan Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *JMLR*, 5:1471–1530, 2004.
- 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 *ICLR*, 2024.
- Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.
- Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. In *ICLR*, 2024.
- Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

- James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *PNAS*, 114(13):3521–3526, 2017.
- Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *EMNLP*, 2021.
- Wenwu Li, Xiangfeng Wang, Wenhao Li, and Bo Jin. A survey of automatic prompt engineering: An optimization perspective. *arXiv preprint arXiv:2502.11560*, 2025.
- Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *ACL*, 2021.
- Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let’s verify step by step. In *The twelfth international conference on learning representations*, 2023.
- Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3):293–321, 1992.
- Saumya Malik, Valentina Pyatkin, Sander Land, Jacob Morrison, Noah A. Smith, Hannaneh Hajishirzi, and Nathan Lambert. Rewardbench 2: Advancing reward model evaluation. *arXiv preprint arXiv:2506.01937*, 2025.
- Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of learning and motivation*, 24:109–165, 1989.
- Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, Chenlin Zhou, Jiayi Mao, Tianze Xia, Jiafeng Guo, and Shenghua Liu. A survey of context engineering for large language models. *arXiv preprint arXiv:2507.13334*, 2025.
- Tom M. Mitchell. The need for biases in learning generalizations. Technical report, Rutgers University, 1980.
- 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 *EMNLP*, 2024.
- Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *NeurIPS*, 2022.
- Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.
- Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with “gradient descent” and beam search. In *EMNLP*, 2023.
- Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pp. 400–407, 1951.
- Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *ICLR*, 2016.
- Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik R. Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

- Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning. *arXiv preprint arXiv:1706.05296*, 2017.
- Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.
- Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *EACL*, 2026.
- Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjana Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *ACL*, 2024.
- Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.
- Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.
- Cilin Yan, Jingyun Wang, Lin Zhang, Ruihui Zhao, Xiaopu Wu, Kai Xiong, Qingsong Liu, Guoliang Kang, and Yangyang Kang. Efficient and accurate prompt optimization: The benefit of memory in exemplar-guided reflection. In *ACL*, 2025.
- Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *ICLR*, 2023.
- 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.
- Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *ICLR*, 2026.
- Wenqi Zhang, Ke Tang, Hai Wu, Mengna Wang, Yongliang Shen, Guiyang Hou, Zeqi Tan, Peng Li, Yueting Zhuang, and Weiming Lu. Agent-pro: Learning to evolve via policy-level reflection and optimization. In *ACL*, 2024.
- Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *AAAI*, 2024.
- Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *ICLR*, 2023.

A Design Rationale and Future-Proofing

Several choices in the RCL formulation of Section 2.1 are deliberate and designed to accommodate increasingly capable models.

Independent components. The reflector g , mutator f , and agent $\mathcal{A}$ are defined as independent components that may be instantiated by the same model or by different models of varying capability. This separation reflects a practical reality: the cognitive demands of diagnosing why a strategy failed (reflection) are qualitatively different from the demands of executing a constrained text edit (mutation) or executing tasks in an environment (forward pass). Allocating model capacity independently to each role allows the system to improve as any individual component improves.

Structured context artifact. We formulate $\mathcal{C}$ as a structured, modular artifact rather than a flat text string. A structured playbook with individually addressable rules enables *localized* updates: the reflector can identify that a specific rule caused a failure and propose a targeted revision, while a flat prompt can only be rewritten holistically. The granularity of the learned artifact determines the granularity of credit assignment that is possible, just as the choice of network architecture in classical learning determines the granularity of gradient updates.

Expanding scope with model capability. These choices are designed to be future-proof. As base models improve, the reflector can produce more precise diagnoses, the mutator can execute more sophisticated edits, and the scope of what $\mathcal{C}$ can encode expands from phrasing adjustments to genuine multi-step strategies, tool definitions, and structured behavioral policies. The formulation imposes no ceiling on what can be learned through context updates; it only requires that the learning signal be available to evaluate trajectory quality.

B Formal Primitive Definitions

This appendix states each primitive of Section 3 as an operator on the base update of Eq. 5, and gives the fully composed RCL update.

Table 3: Optimization primitives, the classical pathologies they address, and the RCL loop stage each targets.

Primitive	Stage	Pathology	Prior Work
Batching	Execution	High variance from single samples	ProTeGi; TF-GRPO
Grouped Rollouts	Execution	Confounded attribution	TF-GRPO
Credit Assignment	Reflection	Sparse terminal reward	TextGrad
Auxiliary Losses	Reflection	Surface-level diagnostics	—
Failure Replay	Sampling	Forgetting learned tactics	Dyn. Cheatsheet; ExpeL
Optimizer State	Mutation	Oscillation from stateless updates	ERM; OPRO

Task batching. Instead of reflecting on a single trajectory, we sample B tasks per iteration $\{x_1, \dots, x_B\} \sim \mathcal{D}$, execute each, and reflect on each failed trace independently, producing per-trace diagnostics $\Delta_1, \dots, \Delta_k$ for $k \leq B$ failures. These are passed jointly to the mutator:

$$\mathcal{C}_{t+1} = f(\mathcal{C}_t, \Delta_1, \dots, \Delta_k) \quad (6)$$

This parallels minibatching in SGD, where averaging gradients over B samples reduces update variance by $\mathcal{O}(1/B)$; here the “averaging” is performed by the mutator’s reasoning rather than by arithmetic mean.

Grouped rollouts. Each task x_i is executed G times under the same playbook $\mathcal{C}_t$, yielding a group $\{\tau_i^{(1)}, \dots, \tau_i^{(G)}\}$ with outcomes $\{r_i^{(1)}, \dots, r_i^{(G)}\}$. Groups containing both passes and failures provide contrastive signal — the reflector receives both a successful trace τ^+ and a failed trace τ^- for the same task:

$$\Delta_i = g(\tau_i^+, \tau_i^-, r_i^+, r_i^-, \mathcal{C}_t) \quad (7)$$

When a group contains no contrastive signal (all G traces pass or all fail), the reflector falls back to the single-trace signature of Eq. 3; the contrastive form of Eq. 7 applies only when both outcomes are present. Batching reduces variance across the task distribution; grouped rollouts reduce variance within each task, analogous to the distinction between inter-sample and intra-sample variance reduction in Monte Carlo methods.

Credit assignment. Let $\mathcal{C}_t^{\text{ann}}$ denote the annotated variant of $\mathcal{C}_t$ described in Section 3.2. Each task is executed twice concurrently:

$$\begin{aligned} \tau^{\text{std}} &= \mathcal{A}(x, \mathcal{C}_t), & r^{\text{std}} &= \mathcal{R}(\tau^{\text{std}}, y^*) \\ \tau^{\text{ann}} &= \mathcal{A}(x, \mathcal{C}_t^{\text{ann}}), & r^{\text{ann}} &= \mathcal{R}(\tau^{\text{ann}}, y^*) \end{aligned} \quad (8)$$

The standard trace τ^{std} remains uncontaminated for evaluation; the annotated trace τ^{ann} makes the agent’s decision process observable, enabling entry-level attribution. The reflector receives both traces but only the standard outcome:

$$\Delta = g(\tau^{\text{std}}, \tau^{\text{ann}}, r^{\text{std}}, \mathcal{C}_t) \quad (9)$$

The annotated outcome r^{ann} is excluded because instrumentation alters the agent’s behavior, making r^{ann} an unreliable measure of the playbook’s quality; the annotated trace is used only for its decision-process observability, not its outcome. Attribution is scored per entry: an entry cited on an incorrect action counts as evidence of harm, cited on a correct action as evidence of help, and uncited entries as neutral. When composed with grouped rollouts, total executions per task become $G + 1$: G baseline traces plus one annotated trace, excluded from the contrastive group because instrumentation alters behavior.

Auxiliary losses. $\mathcal{C}$ is organized into named sections with individually addressable entries $e_1, \dots, e_N$, and the mutator is constrained to express updates as localized edit operations — $\text{UPDATE}(e_j, e'_j)$, $\text{ADD}(e_{N+1})$, $\text{DELETE}(e_k)$ — rather than holistic rewrites. This structural constraint is analogous to sparsity regularization in parameter space: it limits the degrees of freedom per update, preventing the mutator from making sweeping changes that overfit to the current batch. The reflector g is decomposed into three parallel diagnostic heads:

$$\Delta = (\Delta^{\text{attr}}, \Delta^{\text{root}}, \Delta^{\text{gap}}) \quad (10)$$

where $\Delta^{\text{attr}} \in \{\text{actionable gap}, \text{execution variance}, \text{intractable}\}$ is a failure attribution classification, Δ^{root} is a root cause analysis, and Δ^{gap} specifies a coverage gap in the current playbook. These heads interact with the mutator: execution-variance attributions bias toward no-ops (preventing unnecessary edits from noisy signal), while actionable-gap attributions with specific root causes drive targeted additions or modifications. The decomposition forces the reflector to produce structured diagnoses rather than unstructured narrative, analogous to how auxiliary loss heads in multi-task learning force intermediate representations to capture specific aspects of the input.

Failure replay. We maintain a failure replay buffer $\mathcal{B}_t$ that modifies the sampling distribution at each iteration. Let $\rho \in [0, 1]$ be the replay ratio. At each iteration, $\lfloor \rho \times B \rfloor$ tasks are drawn from $\mathcal{B}_t$ and the remaining $B - \lfloor \rho \times B \rfloor$ are sampled fresh from $\mathcal{D}$:

$$\{x_1, \dots, x_B\} \sim (1 - \rho) \cdot \text{Uniform}(\mathcal{D}) + \rho \cdot \text{Uniform}(\mathcal{B}_t) \quad (11)$$

Tasks enter $\mathcal{B}_t$ upon failure and are managed by two thresholds: a task *graduates* (is removed) after n_{grad} consecutive passes across iterations, confirming the playbook has durably learned to handle it; a task is *evicted* after n_{evict} consecutive failures, indicating it may be intractable under the current playbook and should not dominate the training signal. This implements a curriculum that concentrates optimization effort where the marginal return is highest, analogous to prioritized experience replay (Schaul et al., 2016) where samples are weighted by their learning utility rather than drawn uniformly.

Optimizer state. In gradient-based optimization, momentum maintains an exponential moving average $m_t = \beta m_{t-1} + (1 - \beta) \nabla \mathcal{L}_t$, smoothing the update trajectory. We implement an analogous mechanism in context space: a structured, rolling optimization state document S_t , updated by a dedicated model call h after each iteration:

$$S_{t+1} = h(S_t, \Delta_1, \dots, \Delta_k, C_t, C_{t+1}) \quad (12)$$

S_t tracks a change ledger (what was modified and why), playbook assessment (which entries are working well vs. poorly), open hypotheses (conjectured failure modes not yet confirmed), and optimization phase (exploratory vs. convergent). The state document is injected into the mutator but excluded from the reflector:

$$C_{t+1} = f(C_t, \Delta_1, \dots, \Delta_k, S_t) \quad (13)$$

This asymmetry mirrors how momentum in Adam operates on the optimizer step rather than on gradient computation: the reflector’s diagnostics Δ_i remain unbiased by the consensus of past iterations, while the mutator can use S_t to contextualize current diagnostics, avoid reverting previously validated changes, and maintain consistency across iterations.

Full composed update. Incorporating all six primitives, the RCL update becomes:

$$C_{t+1} = f\left(C_t, \{g(\tau_i^+, \tau_i^-, \tau_i^{\text{ann}}, r_i, C_t)\}_{i \in \mathcal{B}_t^o}, S_t\right) \quad (14)$$

where $\mathcal{B}_t^o$ denotes the replay-mixed batch of B tasks (Eq. 11), each executed $G + 1$ times (grouped rollouts plus one annotated trace), g is the multi-head reflector of Eq. 10, and S_t is the optimizer state. Each primitive addresses a distinct pathology at a specific stage of this update; Section 4.2 evaluates their individual and composed contributions.

C Prior Methods as Instances of the RCL Loop

Table 4 situates each prior context-optimization method within the RCL loop by its reflector, mutator, optimizer state, and task regime. Each method instantiates one or two of the primitives of Section 3 in isolation — e.g., ProTeGi batches textual gradients, TF-GRPO adds grouped contrastive rollouts, ERM and OPRO retain optimizer history — under its own base model, benchmark suite, and prompting conventions. None varies the primitives under a fixed substrate or compares their contribution across task regimes, which is the gap our controlled study addresses.

D Leave-One-Out Analysis of the Composed Optimizer

Standalone value does not predict compositional role. To distinguish marginal gains from compositional role, we perform leave-one-out ablations of the full RCL optimizer. Table 2 measures the marginal value of adding a single primitive to ACE, whereas Table 5 measures that primitive’s role once the full optimizer is assembled. These are not the same quantity. Auxiliary losses illustrate this clearly: as a standalone addition to ACE, they improve 7 of 8 settings, yet removing them from full RCL causes only modest degradations in most conditions and even improves RewardBench2/Nano by 12.6 points. Credit assignment shows the converse pattern. As a standalone addition, it helps only 3 of 8 settings and ties one, but removing it from full RCL produces the largest drop in three settings—AppWorld Normal/Nano, BrowseComp+/Lite, and RewardBench2/Lite—and causes a further 10.0 point drop on BrowseComp+/Nano. Batching shows a similar reversal on BrowseComp+/Nano: it hurts when added to ACE, but removing it from full RCL causes a 14.6-point drop. Standalone gains therefore do not reliably predict a primitive’s compositional role.

Two primitives are consistently load-bearing. At the same time, the leave-one-out results do not imply that the composed optimizer is arbitrary. Two primitives are especially load-bearing in composition. Removing grouped rollouts hurts 7 of 8 settings and ties the remaining one; it never exceeds full RCL and produces the largest drop on AppWorld

Table 4: Prior context-optimization methods as instances of the RCL loop, grouped by developmental thread.

Method	Reflector (g)	Mutator (f)	State / Memory	Regime
<i>Loop development</i>				
Reflexion (Shinn et al., 2023)	Verbal self-critique	Append to memory	Episodic history	Single-ep.
ExpeL (Zhao et al., 2024)	Experience extraction	Insight reuse	Extracted knowledge	Cross-task
Agent-Pro (Zhang et al., 2024)	Belief + policy critique	DFS-style search	World model beliefs	Policy
Dyn. Cheatsheet (Suzgun et al., 2026)	Failure summarization	Curated append	Persistent cheatsheet	Task learn.
ACE (Zhang et al., 2026)	Trajectory critique	Structured delta edits	Playbook history	Policy
<i>Reflection mechanism</i>				
ProTeGi (Pryzant et al., 2023)	Minibatch text gradient	Beam search + bandit	None	Micro-opt.
TextGrad (Yuksekgonul et al., 2024)	Textual differentiation	LLM-proposed edit	None	Modular opt.
ERM (Yan et al., 2025)	Exemplar reflection	Beam search	Historical feedback	Micro-opt.
TF-GRPO (Cai et al., 2025)	Semantic group advantage	Context update	Experiential library	Task learn.
Ding et al. (2025)	Sampling momentum	Textual grad. descent	Past distributions	Micro-opt.
<i>Search and selection</i>				
APE (Zhou et al., 2023)	None (score only)	Monte Carlo selection	None	Instr. search
EvoPrompt (Guo et al., 2024)	None (score only)	Evolutionary mutation	Candidate pop.	Instr. search
PromptBreeder (Fernando et al., 2024)	None (score only)	Self-referential	Meta-prompts	Instr. search
GEPA (Agrawal et al., 2026)	Pareto-aware reflection	Evolutionary + Pareto	Candidate pop.	Policy
<i>Structured program optimization</i>				
DSPy (Khatab et al., 2024)	N/A (compiler)	Module-level compilation	Program structure	Program opt.
MIPRO (Opsahl-Ong et al., 2024)	Proposal scoring	Bayesian surrogate	Module-level state	Program opt.

Table 5: **Leave-one-out ablation of the composed RCL optimizer.** Each row removes one primitive from the full RCL composition. Deltas are relative to full RCL. Positive deltas indicate settings where removing a primitive improves over the full composition, while negative deltas indicate settings where that primitive contributes positively in composition. Largest degradation per column is underlined.

	AppWorld				BrowseComp+		RewardBench2	
	Normal		Challenge		Test Set		Test Subset	
	Lite	Nano	Lite	Nano	Lite	Nano	Lite	Nano
RCL (all primitives)	89.3	89.1	71.9	83.7	37.3	51.3	82.4	69.8
– Failure Replay	85.7 ^{-3.6}	88.5 ^{-0.6}	71.5 ^{-0.4}	<u>74.9^{-8.8}</u>	38.7 ^{+1.4}	<u>33.3^{-18.0}</u>	83.1 ^{+0.7}	59.4 ^{-10.4}
– Optimizer State	89.9 ^{+0.6}	85.5 ^{-3.6}	71.2 ^{-0.7}	82.5 ^{-1.2}	34.0 ^{-3.3}	46.7 ^{-4.6}	74.6 ^{-7.8}	64.1 ^{-5.7}
– Credit Assignment	89.9 ^{+0.6}	<u>85.0^{-4.1}</u>	73.4 ^{+1.5}	82.1 ^{-1.6}	<u>32.7^{-4.6}</u>	41.3 ^{-10.0}	<u>74.3^{-8.1}</u>	75.1 ^{+5.3}
– Grouped Rollouts	<u>79.8^{-9.5}</u>	86.8 ^{-2.3}	71.9	79.2 ^{-4.5}	36.7 ^{-0.6}	45.3 ^{-6.0}	78.5 ^{-3.9}	<u>58.5^{-11.3}</u>
– Batching	83.9 ^{-5.4}	86.3 ^{-2.8}	70.4 ^{-1.5}	79.4 ^{-4.3}	40.0 ^{+2.7}	36.7 ^{-14.6}	81.2 ^{-1.2}	<u>64.6^{-5.2}</u>
– Auxiliary Losses	87.5 ^{-1.8}	89.2 ^{+0.1}	<u>70.0^{-1.9}</u>	82.2 ^{-1.5}	33.3 ^{-4.0}	49.3 ^{-2.0}	80.9 ^{-1.5}	82.4 ^{+12.6}

Normal/Lite (−9.5) and RewardBench2/Nano (−11.3). Removing failure replay shows a similarly strong pattern, including the largest drop on AppWorld Challenge/Nano (−8.8)

and the single largest regression in the table on BrowseComp+/Nano (−18.0). Batching and optimizer state also help in many settings—each removal hurts 7 of 8 conditions—but their effects are more distributed than concentrated in a single dominant regime. This is consistent with the training-dynamics analysis, where optimizer state mainly stabilizes convergence and batching mainly affects peak performance and relearning. The full optimizer therefore depends on several complementary mechanisms, even if their effects overlap enough that standalone ablations are a poor proxy for compositional contribution.

The remaining interactions are regime-dependent. The remaining reversals point to regime-dependent interaction effects. BrowseComp+, which the main results characterize as a wide-gap, strategy-learning benchmark, is especially sensitive on Nano to removing failure replay (−18.0), batching (−14.6), and credit assignment (−10.0). The replay and batching drops fit the broader picture of a setting that requires repeated refinement of difficult failures, while the credit-assignment drop reinforces the gap between standalone and compositional value identified above. RewardBench2/Nano shows a different pattern: removing grouped rollouts sharply hurts (−11.3), reinforcing the importance of contrastive signal in a near-deterministic calibration task, while removing auxiliary losses (+12.6) and credit assignment (+5.3) improves performance, consistent with the broader observation that over-structuring can hurt Nano on this benchmark. AppWorld, where the seed-to-ceiling gap is narrower, shows distributed sensitivity rather than a single dominant dependency: no single primitive dominates, with the most critical removal shifting from grouped rollouts on Normal/Lite to failure replay on Challenge/Nano. Primitive contributions in context-space optimization are therefore real but non-additive. Their effect depends on the task regime, the agent, and the other mechanisms with which they are combined.

E Extended Training Dynamics

This appendix gives the precise definitions of the training-dynamics metrics summarized in Section 4.4, and repeats the analysis under three window sizes.

Metrics. We track two quantities at each checkpoint, both computed per-example and then aggregated. The *current TGC* (solid lines) is the fraction of dev tasks solved at that checkpoint. The *recently solved rate* (dashed lines) is the fraction of dev tasks solved at least once in a trailing window of $w=5$ iterations: for each task, we check whether any checkpoint in the window solves it, then take the fraction across tasks. The recently solved rate is always $\geq$ the current TGC, because any task solved now was also solved within the window. This rate can decrease over time as tasks fall out of the trailing window.

The gap between the recently solved rate and the current TGC decomposes into two components, visualized as shading in Figure 2. *Active instability* (colored shading) is this gap directly: the fraction of tasks the optimizer has solved within the last w iterations but that the current playbook fails on. These are recent regressions — capabilities demonstrated within the window but not currently retained. *Stale regressions* (gray shading) capture a second layer: the gap between the all-time per-example best-so-far envelope and the recently solved rate. These are tasks solved at some point historically but not within the trailing window — capabilities lost earlier in training and not yet recovered. Together, the two shadings decompose the full gap between demonstrated capability (all-time envelope) and current performance into recent forgetting (active) and older forgetting (stale).

We report four summary statistics per run. *First all solved*: the first iteration at which every dev task has been solved at least once (cumulative, all-time). This does not mean all tasks are solved simultaneously — it means the optimizer has, at some point, produced a playbook capable of solving each individual task. *Mean instability*: the mean active instability gap across all checkpoints. *Max TGC*: the highest current TGC at any single checkpoint. *% relearned*: when a task flips from solved to unsolved between consecutive checkpoints, that is an unlearn event; if it later flips back to solved, that is a recovery. % relearned is the fraction of unlearn events that are eventually recovered, measuring how often forgetting is reversible.

Section 4.4 summarizes two forms of instability — active (recent regressions within the trailing window) and stale (older regressions outside the window) — using a window of $w=5$ iterations. The window size controls how strictly we define “recent”: a shorter window requires a task to have been solved very recently to avoid being counted as stale, while a longer window is more forgiving. Varying the window reveals whether a primitive maintains capabilities through sustained re-solving (narrow active gap even at short windows) versus relying on one-off discoveries that are not reproduced (active gap widens as the window shrinks, with more instability shifting from active to stale). We show this analysis for all eight configurations (the six primitives, ACE, and the composed RCL) under three window sizes.

5-iteration window (Figure 4). This is the strictest recency condition and produces the widest active instability gaps: a task must have been solved within the last 5 checkpoints to avoid being classified as stale. Under this view, Optimizer State and Grouped Rollouts show the narrowest active gaps, indicating they retain capabilities through consistent re-solving. Failure Replay shows the widest active gaps but also the highest % relearned — tasks that regress are quickly re-encountered via the replay buffer and re-solved, producing high volatility but reliable recovery.

10-iteration window (Figure 5). The wider window is more forgiving: tasks solved anywhere in the last 10 iterations count as recently solvable, shifting instability from the active to the stale category. Active gaps narrow across all primitives relative to the 5-iteration view, but the relative ordering is preserved. This confirms that the differences between primitives reflect genuine differences in retention behavior, not artifacts of the window size.

All-time envelope (Figure 6). With no window, the recently solved rate becomes the cumulative best-so-far: a task counts as recently solvable if it was ever solved. The dashed line is now monotonically non-decreasing and converges to 100% for all runs, confirming that every primitive eventually discovers a playbook capable of solving each dev task. All instability is now active by definition (the stale category is empty), and the remaining gap — between the all-time envelope and current TGC — is the classical forgetting measure: capabilities ever demonstrated but not currently retained. Even under this most permissive view, meaningful gaps persist for all primitives except Optimizer State and Grouped Rollouts, underscoring that catastrophic forgetting is a genuine pathology in context-space optimization.

F Experimental Setup Details

Training pools and splits. AppWorld: we sample training trajectories from a pool of 90 tasks and evaluate on the held-out Normal (168 tasks) and Challenge (417 tasks) test splits. BrowseComp+: a training pool of 100 queries, 30 queries for validation, and 150 held-out queries for evaluation. RewardBench2: a training pool of 1,307 examples, 277 for validation, and a 281-example test split. In all cases the optimizer sees at most $B \times T$ training examples over T iterations, where B is the batch size; coverage of the full training pool is neither required nor guaranteed, and sampling is governed by the replay buffer when active (§3.4).

Rollout protocol. At each iteration the optimizer samples and executes a set of tasks, then selects up to B failed traces for reflection and mutation. The rollout budget is calibrated to the seed failure rate of the current playbook so that each iteration reliably surfaces enough negatives to fill a batch. This calibration matters because long-horizon agentic tasks are expensive to execute: each rollout may involve multi-step tool use, API calls, and environment interaction, making it impractical to sample until failures appear by chance. Rollout budgets therefore reflect the differing seed failure rates — AppWorld’s high seed pass rate requires fewer executions per iteration to surface failures, while BrowseComp+’s low pass rate demands larger rollout batches. When grouped rollouts are active, groups are prioritized for selection based on contrastive signal: groups containing at least one pass and one fail are selected first, as these provide the sharpest attribution; if all groups



Figure 4: Training dynamics for all primitives with a 5-iteration sliding window. Format follows Figure 2: solid = current TGC; dashed = recently solved rate (per-example union over 5 iterations); colored shading = active instability (solved within window but not now); gray shading = stale regressions (solved historically but not within window). This is the strictest recency condition and produces the widest active instability gaps.

fail uniformly and the evaluation signal is non-binary, we select groups with the largest variance.

Computational cost. Optimization cost is dominated by task executions and optimizer-model calls; all learned artifacts are text, so memory overhead is negligible. Relative to the ACE loop, grouped rollouts multiply executions by G and credit assignment adds one execution per task, while auxiliary losses and failure replay add none — they restructure the reflection prompt and the sampling distribution respectively — and optimizer state adds a single state-update call per iteration, independent of B . Batching scales the rollout budget to surface B failed traces and grows the mutator’s input accordingly. Two of the cheapest primitives, optimizer state and auxiliary losses, are among the most effective (Section 4.2).

ACE reproduction scope. We adopt ACE’s structured delta edits, helpful/harmful bullet scoring, and the Generator–Reflector–Curator decomposition. We omit two optional mechanisms: (i) the embedding-based de-duplication step, replacing it with explicit update/delete operations that manage playbook size without an external embedding model, and (ii) multi-round Reflector refinement, using a single reflection pass instead. Neither omission changes the fundamental loop; both simplify the infrastructure while keeping the baseline competitive.

GEPA configuration. GEPA collects execution traces and applies natural-language reflection to diagnose errors and propose prompt revisions, with a genetic Pareto search over candidate prompts maintaining a diversity frontier. We use the official DSPy implementation (Khatab et al., 2024) with `auto="light"`, the setting that most closely matches our experimental budget (typically exceeding 30 optimizer iterations), and a mini-batch size of 3, equal to our batch size $B=3$. GEPA performs reflective updates on the training set but scores candidates on a held-out validation set to maintain its Pareto frontier; we use 56 tasks for AppWorld, 30 queries for BrowseComp+, and 277 examples for RewardBench2.

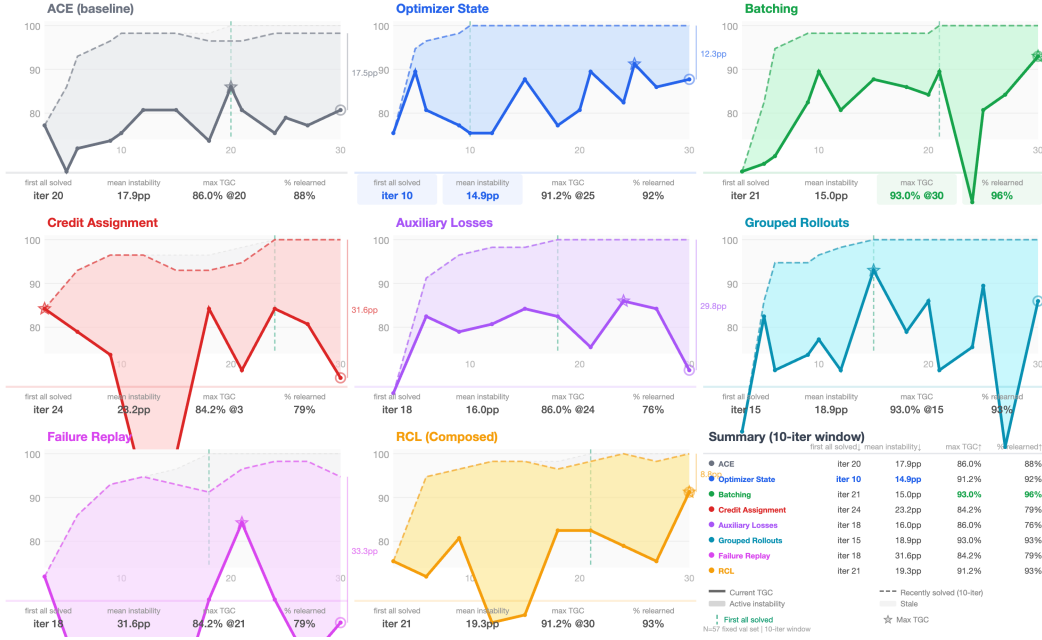


Figure 5: Training dynamics with a 10-iteration sliding window. Tasks solved anywhere in the last 10 iterations count as recently solvable, shifting instability from the active to the stale category. Active gaps narrow relative to the 5-iteration view (Figure 4), but the relative ordering of primitives is preserved.

G Reproducibility

Code is available at <https://github.com/nvassilyev/RCL>. The repository contains the full RCL implementation and the per-benchmark configurations used for every run reported here, together with supporting experimental artifacts. The system prompts and seed playbooks are also reproduced in Appendices H–J. We use GitHub issues and discussions as the correspondence channel for questions about the paper or code.

H Prompts

This section provides the prompts used for AppWorld (Figure 7), BrowseComp+ (Figure 8), and RewardBench2 (Figure 9).

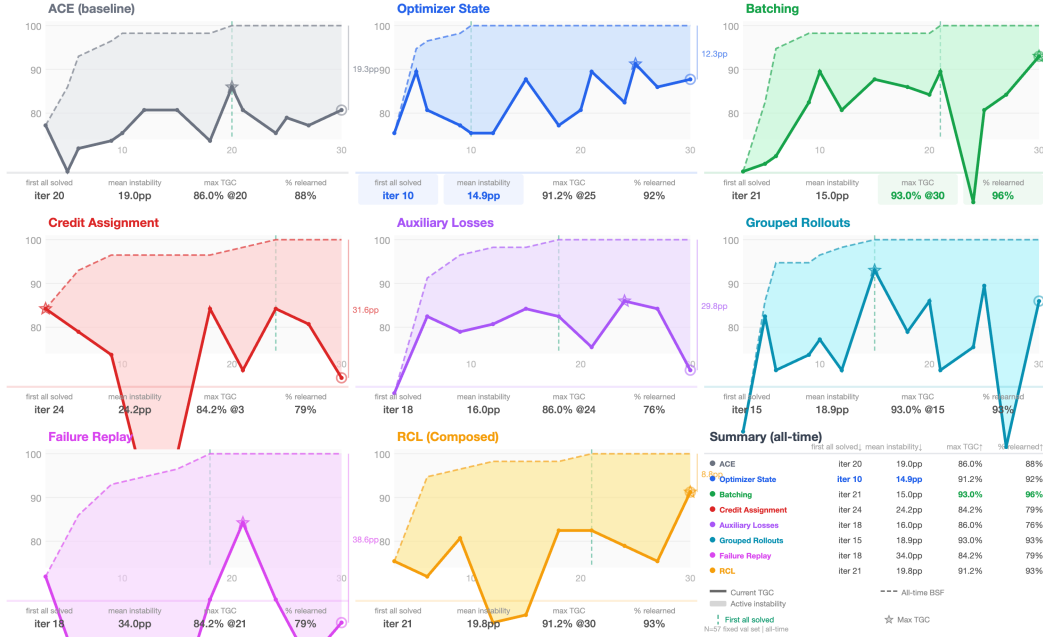


Figure 6: Training dynamics with the all-time best-so-far envelope (no window). The dashed line is now monotonically non-decreasing and converges to 100% for all runs. All instability is active by definition (the stale category is empty). The remaining gap measures capabilities ever demonstrated but not currently retained. This is the most permissive view and produces the narrowest gaps.

I Seed Playbooks

This section provides the seed playbooks for which we initialized the ACE and RCL experiments in AppWorld (Figure 10), and RewardBench2 (Figure 11). For Browsecomp+, we initialize the playbook with no entries.

J Quality Playbooks on AppWorld

For these experiments, we define the different qualities of playbook initialization for learning the dynamics of AppWorld task. We experiment with an empty playbook with 0 entries, decent playbook (Figure 12), and high quality (Figure 13).

K LLM Disclosure

In addition to the LLM components described as part of the research methodology (Section 2), large language models were used to assist with implementation, analysis, and writing portions of this paper.

AppWorld System Prompt
I am your supervisor, and you are an AI Assistant whose job is to complete my day-to-day tasks fully autonomously. To do this, you will need to interact with app(s) (e.g., spotify, venmo etc) using their associated APIs on my behalf. Use the <code>execute_python</code> tool to run code one step at a time, observe the output, then decide the next step. Key APIs: <pre>execute_python(code="print(apis.api_docs.show_app_descriptions())") execute_python(code="print(apis.api_docs.show_api_descriptions(app_name='spotify'))") execute_python(code="print(apis.api_docs.show_api_doc(app_name='spotify', api_name='login'))")</pre> Key instructions: A. General instructions: - Act fully on your own. Never ask for confirmation. - Never invent or guess values - always look them up via APIs. - Never leave placeholders - always fill in real values. B. App-specific instructions: - All credentials are in the Supervisor app: <code>apis.supervisor.show_account_passwords()</code> - References to friends/family refer to phone contacts. - Get current date/time from <code>datetime.now()</code> or phone API, never your internal clock. - Paginated APIs: Always loop through all pages. Don't stop at first page. C. Code-operation instructions: - Use the <code>execute_python</code> tool to run code. Execute one step at a time, observe its output, then decide the next step. - Always use <code>print()</code> to see API return values - otherwise output is hidden. - Always check API specs before calling an API. D. Task-completion: - Call <code>apis.supervisor.complete_task(answer=...)</code> when done. - Keep answers minimal (just the value, not full sentences). - Numbers must be numeric ("10" not "ten"). # Playbook You have been given a curated playbook of strategies, common mistakes, and proven solutions. Read it carefully and actively apply its guidance throughout your task execution. <i>[Dynamically generated playbook entries are appended here during evaluation.]</i>

Figure 7: Base system prompt for AppWorld benchmark. The playbook section header is followed by dynamically generated playbook entries during evaluation.

BrowseComp System Prompt
You are a deep research agent. You need to answer the given question by interacting with a search engine, using the search tool provided. Please perform reasoning and use the tool step by step, in an interleaved manner. You may use the search tool multiple times. Your response should be in the following format: Explanation: {{your explanation for your final answer. Cite evidence documents inline by enclosing their docids in square brackets [.]}} Exact Answer: {{your succinct, final answer}} Confidence: {{your confidence score between 0% and 100% for your answer}} # Playbook You have been given a curated playbook of strategies, common mistakes, and proven solutions. Read it carefully and actively apply its guidance throughout your search and reasoning. <i>[Dynamically generated playbook entries are appended here during evaluation.]</i>

Figure 8: Base system prompt for BrowseComp benchmark. The playbook section header is followed by dynamically generated playbook entries during evaluation.

RewardBench2 System Prompt
You are evaluating candidate assistant responses for the same user prompt. Your job is to act like a careful reward model judge: - score each candidate independently rather than rewarding position or style alone - prefer responses that are correct, instruction-following, relevant, complete, and appropriately safe - penalize hallucinations, mathematical mistakes, broken constraints, evasive padding, unsafe compliance, and bad refusals - only treat responses as tied if they are genuinely equally strong overall Return ONLY valid JSON with this schema: <pre>{ "reasoning": "2-5 concise sentences explaining the main quality differences", "ratings": [0.0, 0.0], "best_response_ids": ["1"] }</pre> Rules: - ratings must be numeric scores from 0.0 to 10.0 - ratings must be a JSON array containing exactly one score per candidate, in candidate-id order - "best_response_ids" must contain the id(s) with the highest score - do not include markdown fences or any text outside the JSON object # Playbook You have been given a curated playbook of judging strategies, failure patterns, and calibration rules. Read it carefully and apply it while rating candidates. <i>[Dynamically generated playbook entries are appended here during evaluation.]</i>

Figure 9: Base system prompt for RewardBench2 benchmark. The playbook section header is followed by dynamically generated playbook entries during evaluation.

Seed Playbook – AppWorld
strategies_and_hard_rules (3 entries) Always end code blocks with “`” on its own line. Never use ellipsis (...) in code. Write small, incremental code chunks – verify each step works before proceeding. Variables persist across code blocks – reuse them instead of re-fetching. apis_to_use_for_specific_information (3 entries) Check API docs with <code>apis.api_docs.show_api_doc(app_name, api_name)</code> before calling any API. Use supervisor app for account credentials: <code>apis.supervisor.show_account_passwords()</code> Use phone app for contact/person lookup: <code>apis.phone.show_contacts()</code> to find friends/family. useful_code_snippets_and_templates (1 entries) Paginated APIs require looping: <code>while True: page = api(page_index=i); if not page: break</code> common_mistakes_and_correct_strategies (1 entries) Example credentials shown in few-shot examples are FAKE – always fetch real credentials from supervisor. verification_checklist (1 entries) Always call <code>apis.supervisor.complete_task()</code> when done – this marks the task complete. Only pass an <code>answer=</code> argument if the task explicitly asks you to find, calculate, or return a specific value. For action-only tasks (e.g., send a message, play a song), call <code>complete_task()</code> with NO <code>answer</code> argument.

Figure 10: Seed Playbook – AppWorld. Contains 9 entries across 5 sections.

Seed Playbook – RewardBench2
core_ranking_criteria (1 entries) Score each candidate independently against the user prompt before comparing candidates to each other. instruction_following_checks (1 entries) When the prompt contains explicit constraints, verify each candidate against every constraint instead of rewarding vague relevance. factuality_and_math_checks (1 entries) Penalize factual or mathematical mistakes even if the answer sounds polished or confident. safety_and_refusal_calibration (1 entries) Reward appropriate refusals only when the request is actually unsafe or disallowed; otherwise penalize unnecessary refusal. tie_handling_and_score_calibration (1 entries) For multiple valid answers, keep the scores of correct answers close together and reserve larger gaps for separating correct from incorrect responses. common_mistakes (1 entries) Do not let verbosity, politeness, or formatting polish outweigh correctness, constraint satisfaction, and relevance.

Figure 11: Seed Playbook – RewardBench2. Contains 6 entries across 6 sections.

Seed Playbook (Decent) – AppWorld
apis_to_use_for_specific_information (3 entries) Check API docs with <code>apis.api_docs.show_api_doc(app_name, api_name)</code> before calling any API. Use supervisor app for account credentials: <code>apis.supervisor.show_account_passwords()</code> Use phone app for contact/person lookup: <code>apis.phone.show_contacts()</code> to find friends/family. strategies_and_hard_rules (2 entries) Write small, incremental code chunks - verify each step works before proceeding. Variables persist across code blocks - reuse them instead of re-fetching. useful_code_snippets_and_templates (1 entries) Paginated APIs require looping: <code>while True: page = api(page_index=i); if not page: break</code> verification_checklist (1 entries) Always call <code>apis.supervisor.complete_task()</code> when done - this marks the task complete. Only pass an <code>answer=</code> argument if the task explicitly asks you to find, calculate, or return a specific value. For action-only tasks (e.g., 'send a message', 'play a song'), call <code>complete_task()</code> with NO answer argument.

Figure 12: Seed Playbook (Decent) – AppWorld. Contains 7 entries across 4 sections.

Seed Playbook (High Quality) – AppWorld
strategies_and_hard_rules (3 entries) Always end code blocks with “<code>”</code> on its own line. Never use ellipsis (...) in code. Write small, incremental code chunks - verify each step works before proceeding. Variables persist across code blocks - reuse them instead of re-fetching. apis_to_use_for_specific_information (3 entries) Check API docs with <code>apis.api_docs.show_api_doc(app_name, api_name)</code> before calling any API. Use supervisor app for account credentials: <code>apis.supervisor.show_account_passwords()</code> Use phone app for contact/person lookup: <code>apis.phone.show_contacts()</code> to find friends/family. useful_code_snippets_and_templates (1 entries) Paginated APIs require looping: <code>while True: page = api(page_index=i); if not page: break</code> common_mistakes_and_correct_strategies (1 entries) Example credentials shown in few-shot examples are FAKE - always fetch real credentials from supervisor. verification_checklist (1 entries) Always call <code>apis.supervisor.complete_task()</code> when done - this marks the task complete. Only pass an <code>answer=</code> argument if the task explicitly asks you to find, calculate, or return a specific value. For action-only tasks (e.g., 'send a message', 'play a song'), call <code>complete_task()</code> with NO answer argument.

Figure 13: Seed Playbook (High Quality) – AppWorld. Contains 9 entries across 5 sections.