AGAR: A REINFORCEMENT LEARNING SUBSTRATE FOR LLM PROGRAM EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Given a task and an evaluator, a language model can rewrite a candidate program while a search loop decides which rewrites survive — a practical route to algorithm discovery. But that loop is governed by five constants set by hand — which parent to select, how hard to mutate, how to keep diversity, what to remember, and a scalar score that never says which part of the program earned it — and reinforcement learning already has an estimator for each. The obstacle is that program evolution is not usually written down as a decision process. We formalize it as a Markov decision process whose action is the **modular prefix the model is conditioned on, not the program it emits**, so that credit assignment, value estimation, adaptive exploration and experience memory each attach to a distinct component. AGAR (*Algorithm Generation As RL*) provides the substrate this licenses: any estimator can be replaced or switched off without touching the controller, making the transfer auditable one mechanism at a time, with no gradient training of the backend model. Across 19 tasks, two backends and three seeds under one harness, AGAR improves on the stronger of two published baselines on most tasks, with the gains concentrated on the competitive-programming family. The formalization also yields a checkable reading of prior work: these systems are implicitly zero-discount, not by choice but because fitness is exogenous to an individual rather than a return over successors, leaving a discount factor nothing to act on.

1 INTRODUCTION

A long-horizon algorithm optimization task — find a program that scores well on a task-specific evaluator, given a budget of iterations and no gradient of the score with respect to the program text — is a sequential decision problem. It is usually not *written down* as one. It is written down as an evolutionary loop: maintain a population of programs, ask a language model to mutate or recombine them, evaluate the offspring, keep what scores well, repeat (Romera-Paredes et al., 2024; Novikov et al., 2025; Lange et al., 2025; Sharma, 2025). The results are real: improved bounds on open combinatorial problems, faster kernels, better heuristics.

The thesis. We take the position that such a task *can be modelled as an RL problem*, and that doing so is not a re-labelling exercise: the modelling choices determine which machinery is applicable, and the machinery then determines the system. Concretely, we formalize the loop as an MDP whose action is the *modular prefix* the model is conditioned on — not the program it emits — and we show that under that choice each load-bearing idea of modern RL (Sutton & Barto, 2018) attaches to a distinct component of the loop. M_1 (credit assignment), M_2 (value estimation), M_3 (adaptive exploration) and M_4 (experience memory) are the places the modelling says the estimators go. We are deliberate about the strength of that claim: the formalization forces which *kinds* of estimator the objective requires and whether each is an action factor or retrieved context, but it does not force that there are four of them, and the boundaries we draw between M_1, M_2 and M_3 are a factorization we chose rather than one the mathematics uniquely picks out. What we contribute is a *general evolutionary substrate* that carries this decomposition: an interface in which any one of the estimators can be replaced or switched off without touching the controller, so that the transfer is auditable one mechanism at a time.

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.

The gap and the obstacle. What the LLM contributes today is the mutation operator. Everything that decides *where to mutate, how hard, and what to remember* is a hand-set constant: five decisions, each with an estimator RL already has. The obstacle is not reward density — RL routinely operates under terminal-only reward, and LLM-driven program evolution does receive intermediate signal which the mutation operator consumes. The real difference is structural. In RL the value of a state is defined *recursively* over its successors, so terminal credit is propagable in principle. In evolution, fitness is an *exogenous property of an individual*; no Bellman equation relates a parent's fitness to its children's, so there is no recursion to propagate along at all. We make this precise in §2.1. The formalization below makes the recursion *definable* over the search without claiming to make the backup *empirical*: the value it recurses on is estimated in text space rather than learned from observed returns, and that boundary stays exactly where it is.

A checkable reading of prior work. Because the action is a prefix conditioning a frozen black-box model, the objective is $J(\pi) = \mathbb{E}[\sum_t \gamma^t r_t] + \lambda H(\pi)$. Setting $\gamma{=}0$ recovers the selection rule every system in this literature uses (Observation 1): immediate fitness plus a population entropy term. This is not a metaphor but a degeneracy — and not a criticism, since without a recursion a discount factor has nothing to act on. Making the recursion definable is what turns γ into a tunable knob.

A substrate, not a system. The substrate (§5) carries the transfer: interchangeable modules, interchangeable execution backends, native asynchrony. Its engineering problems are those of asynchronous RL post-training stacks (Luo et al., 2025; Zhu et al., 2025; miles Contributors, 2025; Hou et al., 2026); the optimand differs (text, not weights) and there is no gradient.

Contributions.

1. **A modelling of algorithm optimization as RL and the substrate it licenses** (§3). Action-as-prefix makes five RL tools expressible in text space, each replacing a named fixed rule in a specific module. The $\gamma{=}0$ degeneracy falls out of the same formalization. The contribution is an *interface* auditable one mechanism at a time, not a uniqueness result.
2. **Modular TextGrad: policy improvement in text space** (§4). Each module's persistent *guidelines* — the text-space analogue of policy weights — are optimized without gradient training. We report a measurement of the operator: a local-edit and validation-revert design that replaces the original rewrite operator (§4.2).
3. **Asynchronous execution and the credit-assignment problem it creates** (§5.2, §5.3). The streaming scheduler creates staleness: credit is assigned against a population that no longer exists. We characterize τ empirically (naive $\tau \leq W{-}1$ is refuted in practice) and relate it to asynchronous RL post-training (Hou et al., 2026).

2 Background: Two Independent Lineages

Evolutionary computation searches by maintaining a population, perturbing it, and keeping what scores well. Genetic programming applied this to program spaces decades ago (Koza, 1992); the LLM era replaced the syntax-tree mutation operator with a language model, which is what made the approach competitive on real algorithm-design problems (Romera-Paredes et al., 2024; Novikov et al., 2025). Later systems differ in how they manage the population — ShinkaEvolve adds a bandit over LLM backends and novelty sampling (Lange et al., 2025); SMC-based variants replace generational selection with sequential Monte Carlo resampling (Jiang et al., 2026); EvoX makes the search strategy itself the object of evolution (Liu et al., 2026) — but they share the *structure* of the signal. Fitness attaches to a completed individual, not to a step of a trajectory, and intermediate information informs the *generation* of the next candidate without ever accumulating into a return. The search state — which lineages are productive, which regions are exhausted — is therefore carried implicitly in the population rather than estimated.

Reinforcement learning developed a different toolkit because it assumed a different structure, a trajectory over which value is recursively defined: credit assignment propagates outcome information backwards along it (Sutton, 1988); value estimation predicts return before it is observed (Mnih et al., 2015); adaptive exploration makes the explore/exploit trade-off a decision rather than a con-

stant (Auer et al., 2002; Garivier & Moulines, 2011); entropy regularization turns diversity into an objective term (Haarnoja et al., 2018); and experience replay makes the run's history a learnable resource (Lin, 1992; Schaul et al., 2016). Our five transferred tools are these five, and §3.3 states what each had to become.

2.1 What actually separates them

It is tempting to say that RL has dense reward, evolution has sparse reward, and that this is the obstacle. That is inaccurate in both directions: RL routinely operates under terminal-only reward, and LLM-driven program evolution does receive intermediate signal which the mutation operator consumes. The real difference is structural. In RL, the value of a state is defined recursively in terms of its successors, and terminal credit is in principle propagable along that recursion via returns or TD(λ) — sparse reward makes this *statistically* hard, but the problem stays well-posed. In evolution, fitness is an *exogenous property of an individual*; a child is not a successor state reached by acting in the parent's state but a new object emitted by a stochastic operator, and no Bellman equation relates a parent's fitness to its children's. There is therefore nothing to back up *along*, which is why evolutionary search assigns credit by *survival* rather than by attribution.

The formalization of §3 supplies the missing *structure*: by defining the action as a prefix and the transition as one generation step, it makes a Bellman recursion well-posed over the search. It does not supply the missing *signal*: our V_{M_2} is an estimate produced by a text-space heuristic, not a value learned by backing up observed returns. What *is* live is the forward half — the estimate computed at dispatch — through two channels active in every reported run: the M_3 selection rule $f(p) + \gamma V(p)$ and the M_2 lineage ranking. So:

We make the recursion definable. We do not make the backup empirical.

This also explains Observation 1: prior systems are $\gamma{=}0$ not because their designers chose greediness, but because without a recursion a discount factor has no successor value to act on.

2.2 Positioning

The recurring distinction between AGAR and the closest prior systems is that a rule which is fixed elsewhere becomes a learnable, replaceable component here (Appendix D.6 gives a row-by-row comparison). A parallel line of work builds substrates for applying RL to agents at scale (Luo et al., 2025; Zhu et al., 2025; miles Contributors, 2025; OpenClaw-RL Contributors, 2026), together with asynchronous post-training systems such as SAO (Hou et al., 2026). These are our closest methodological relatives: they share our engineering problems — decompose the loop into independently swappable components, make rollouts asynchronous, bound the staleness that asynchrony introduces — but in every one of them the learnable object is a set of model weights updated by a policy-gradient estimator, whereas here the model is frozen and the learnable object is the *text* that conditions it.

3 Formalization and the Transfer

3.1 LLM-driven program evolution as an MDP

We model one evolutionary step as a transition in an MDP $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$. Figure 1 shows the whole construction at once; the subsections below take its parts in turn.

State s_t is mixed-granularity by necessity: an individual-level part (the parent program, its fitness, its lineage, its age, its mutation history) and a population-level part (fitness statistics, a diversity measure, budget progress and remainder). Both are required — credit assignment and value estimation are questions about an individual, exploration and entropy about a population — and neither subsumes the other: a purely individual state cannot express "the population has converged", and a purely population-level state cannot express "this lineage has stalled".

Action a_t **is a modular prefix, not a program.** This is the load-bearing choice of the formalization. It is tempting to call the generated program the action, but the program is sampled from a frozen

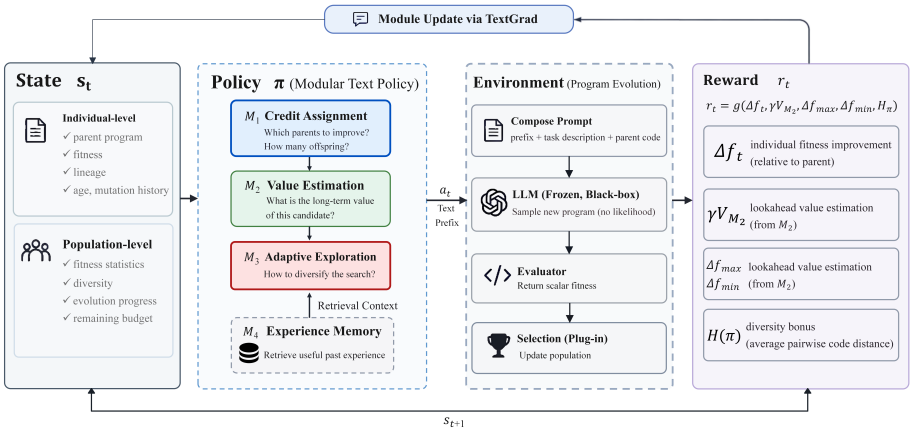


Figure 1: The transfer as a whole. Left to right: the mixed-granularity state s_t; the policy π, whose action a_t is the *modular text prefix* M_1–M_3 produce, with M_4 supplying retrieved context rather than an action factor; the environment, in which the frozen black-box LLM sees only that prefix plus the task description and parent code, and a pluggable selection operator folds the result back into the population; and the reward terms of Eq. equation 2. The outer arrows are the two loops that make this RL rather than a re-labelling: the TextGrad update over the module guidelines θ_i (top), and the $s_t \to s_{t+1}$ transition the search itself performs (bottom).

black-box LLM: it is a *product of the transition*, not something the search policy emits. What the policy emits is the text that conditions that sampling, so we define

$$a_t \;=\; \mathrm{Concat}\big(m_1(s_t),\, m_2(s_t),\, m_3(s_t)\big), \tag{1}$$

the concatenation of the credit-assignment, value-estimation and exploration modules' natural-language outputs. M_4 (experience memory) does not enter the prefix; it is retrieved context consumed by M_1–M_3. Two consequences follow, and both are checks that the formalization does work rather than relabel: π is a text-emitting object improvable without touching model weights (§4), and the module boundary is fixed by which factor of a_t a component writes rather than by taste.

Transition P has three stages: the execution LLM samples a program conditioned on a_t and the parent; an evaluator returns a scalar fitness; a selection operator folds the offspring into the population. The first stage is a frozen black box with no accessible likelihood, which forces a model-free treatment. The third is *pluggable*, which is where value-guided selection enters (Table 5).

Reward R has individual and population terms:

$$\hat{r}_t \;=\; \underbrace{\Delta f_t \;+\; \gamma\, V_{M_2}(\text{child})}_{R_{\text{individual}}} \;+\; \underbrace{w_{\max}\,\Delta f_{\max} \;+\; w_{\text{mean}}\,\Delta f_{\text{mean}}}_{R_{\text{population}}} \;+\; \underbrace{\lambda\, H(\pi)}_{\text{entropy}}. \tag{2}$$

Δf_t is the fitness improvement over the parent. The γV_{M_2} term is where foresight enters: an estimate of what this child is worth as an *ancestor*, filled in retrospectively once its subtree exists (§5). γ is a fixed scalar applied only inside Eq. equation 2 — never passed to the LLM as text, and independent of the `horizon` parameter that controls how much lineage the modules see (see Appendix D.2 for defaults).

3.2 The objective, and the γ=0 degeneracy

$$\max_{\pi} \;\; J(\pi) \;=\; \mathbb{E}_{\tau\sim\pi}\Big[\textstyle\sum_t \gamma^t r_t\Big] \;+\; \lambda H(\pi). \tag{3}$$

We instantiate $H(\pi)$ as mean pairwise code distance within the population, with a behaviour-class Shannon entropy as a secondary indicator. The choice is deliberate: it costs zero additional LLM calls and is already monitored by M_3, so the entropy term is measurable at every step rather than estimated. We add no KL penalty; unlike RLHF there is no natural reference policy to stay close to.

Observation 1 (Existing systems are the γ=0 special case) *Setting γ=0 in Eq. equation 2 deletes the V_{M_2} term and collapses Eq. equation 3 to $\mathbb{E}[r_0] + \lambda H(\pi)$, i.e. a one-step objective in which an individual is worth exactly its immediate fitness improvement plus its contribution to population statistics. This is precisely the selection rule used by every LLM program-evolution system we are aware of.*

Observation 1 is an algebraic identity, not a metaphor, and it is the cleanest reading the formalization gives us of the prior art: γ is a knob whose zero setting reproduces a one-step objective, because a discount factor with no successor value to act on has nothing else to be. We state it as an identity over Eq. equation 2 and leave the empirical consequences of setting $\gamma > 0$ to future work.

We read this as a consequence of §2.1 rather than a shortcoming of prior work. A discount factor needs a successor value; absent a recursion over the search there is none, so γ=0 is the only setting available. What §3.1 contributes is not a better choice of γ but the structure that gives γ a referent at all — with the honest counterpart that the V_{M_2} it multiplies is estimated in text space rather than bootstrapped from observed returns.

3.3 The transfer

Table 5 in Appendix D.7 is the centre of the paper. Each row names an RL tool, the evolutionary fixed rule it displaces, the module it becomes, and — most importantly — *what had to change* given the absence of an empirical backup (§2.1).

We stress the direction of derivation. We did not choose four modules and then find RL names for them; the four factors of a_t in Eq. equation 1 plus the retrieved context M_4 are what the transfer produces, and the fact that M_4 is *not* a factor of the action — it is context — is a consequence of the formalization rather than an implementation detail.

Instantiations. A choice of implementation for each of M_1–M_4 defines a point in a configuration space (§5.1). Throughout the paper we report two such points, Base and Combo, differing in the M_2 estimator and the M_3 decision rule (statistical vs. LLM-based; rule-based vs. discounted-UCB). They are sampled points, not competing systems; we do not argue for one over the other.

3.4 Critic independence, and what it forces

Each module is improved against its *own* critic, which judges only whether that module's output was correct on its own terms — was the attribution consistent with the observed lineage? did the predicted ranking match the realized ordering? — and never attempts to apportion final fitness across modules. The reason is structural: all four outputs are concatenated into one prefix (Eq. 1) and consumed jointly by a black-box LLM, so their contributions to the resulting program are not separable from the inside. This has a design consequence: which transferred tool carries the gain can only be answered by external leave-one-out ablation, not from inside the system. We do not report that decomposition: at n=3 the level-to-level differences of Table 2 already sit inside the noise. See Appendix D.1 for the full module data contracts.

4 Modular TextGrad: Policy Improvement in Text Space

Section 3 defines what we optimize. This section defines how: the method is TextGrad (Yuksekgonul et al., 2024) applied to a target its original formulation already fits — an objective with no analytic gradient, computed by an external system, over a variable that is itself text. The frozen backend model writes code; each module's *persistent guidelines* θ_i are the variable being optimized; the loss is $\hat{r}$ of §3; and the "gradient" is the rollout context shown to the LLM critic. Appendix D.5 gives the full operator definitions and the RL-to-text correspondence table.

Two features of our instance are unlike the tasks TextGrad was demonstrated on. First, the variable is *modular*: the policy is a concatenation of per-module guidelines, so the gradient must be taken per module and a critique that conflates them is unusable. Second, the loss is measured over a *period* of noisy evolutionary steps rather than a single forward pass, so the update rests on a small-sample signal and must decide what to do when that signal is inconclusive. The second feature is what §4.2 is about.

4.1 Operators

For each module i we define a per-module textual gradient operator G_i and a shared update operator U (full definitions in Appendix D.5). The object θ_i is the module's *persistent guidelines*: a block of text that survives across evolutionary steps and conditions every subsequent invocation of that module — not the module's per-step output, which is an action, not a policy parameter. Optimizing the output instead of the guidelines would fail to persist and therefore cannot constitute policy improvement.

4.2 What we measured, and why the operator was replaced

The rewrite operator — critique in, new guidelines out, unconditionally — is the direct transposition of TextGrad to this setting. Two measurements over 206 runs (9,844 update pairs) forced its replacement. **(i) The variable was resampled, not improved**: the guideline block hit its 2000-character budget at the *first* update of every run and stayed there; textual similarity between adjacent versions was 0.18, between first and last 2–6%, without exception. **(ii) Nothing closed the loop**: a rewrite that made things worse persisted until the next rewrite replaced it, so there is no mechanism by which θ_i at step 100 should beat θ_i at step 1.

The repaired operator replaces U with a local edit plus deferred validation. The policy is an explicit numbered rule list; G_i returns at most one ADD and one EDIT/REMOVE per period (NONE is an expected answer); validation is deferred `verdict_lag` periods because a new rule's children are mostly leaves at install time, making immediate scoring equivalent to $\gamma{=}0$. Cost falls from 8 to 2 LLM calls per update (only M_1 and M_4 are edited; the other two were writing guidelines no code path read). The budget is bounded by construction, and the operator became legible: discrete rules with provenance can be reported at run end, which the rewrite operator made impossible. Full measurement detail is in Appendix D.5.

5 Substrate: Extensibility and Asynchronous Execution

A modelling choice only becomes a system if the interfaces it implies are real. This section describes the two properties of the substrate the transfer actually depends on — replaceable components and native asynchrony — and one problem the second creates for the first. We develop the asynchronous part against the closest engineering precedent, the asynchronous RL post-training systems surveyed in §2.

5.1 Extensibility, stated so that it can be checked

"Extensible" is easy to assert, and every substrate paper asserts it. We give three measurable forms. (a) *Configuration space*: the choice of implementation for $M_1 \times M_2 \times M_3 \times M_4$ spans 180 configurations, of which we have evaluated 14; Base and Combo are two of those points. (b) *Integration cost*: replacing M_3's LLM explore/exploit call with the discounted-UCB bandit of Garivier & Moulines (2011) added 236 lines in one new module file and changed 0 lines in the controller or the module protocol — and *deleted* 26 lines of hand-written budget and stagnation heuristic that the discounting subsumes. That deletion is the part we would point to: the claim is not that new mechanisms are cheap to add, but that adding a principled one can remove a hand-tuned constant.

(c) *Interchangeable backends*, and a non-obvious cost. Backends register through a single entry point and can be swapped between a thread pool and a distributed runtime. During development this produced a failure worth reporting: under the distributed backend, per-task serialization gives each rollout an isolated module instance for free, while under the thread-pool backend a shared evaluator instance guarded by an environment lock caused rollouts to be **silently serialized** — no

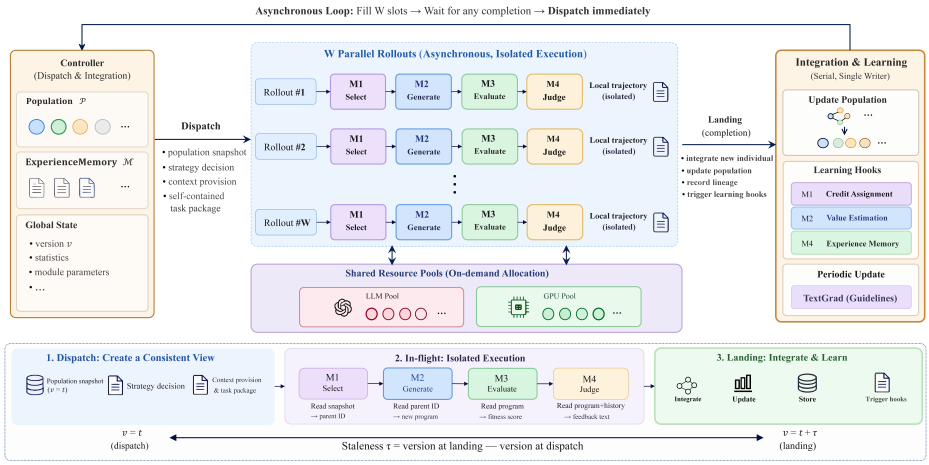


Figure 2: The substrate. Top: the streaming loop — fill W slots, wait for *any* completion, dispatch a replacement immediately — with the controller as sole writer of the population and the W in-flight rollouts reading a dispatch-time snapshot and writing only their own trajectory. The four pipeline stages (SELECT, GENERATE, EVALUATE, JUDGE) are the substrate's; the replaceable modules sit in the SELECT slot. Shared resource pools are acquired per use and returned. Bottom: the same loop in the three moments used throughout this paper, with τ defined as the population-version difference between dispatch and landing.

error, no warning, and worse the longer each evaluation takes, i.e. exactly on the workloads where concurrency matters most. The fix was explicit per-task isolation of module state. The general lesson is that *extensibility is not freedom; it is an obligation on the interface to state its isolation semantics*. An interface permitting a backend swap without specifying whether state is shared has not made the system extensible, only replaceable. Appendix D.9 gives the integration diff in full.

5.2 Asynchronous Execution

The scheduler is streaming rather than generational: it fills W in-flight rollouts, waits for the first to complete, folds the result into the population, and immediately dispatches a replacement. There is no batch barrier, so a single slow evaluation delays one slot rather than the whole generation — which matters here because per-rollout times across the tasks we study span $3.7\times$ (median 269 s on the fastest task against 1007 s on the slowest), so a straggler holds a slot for several times the median. As an engineering decision this needs no justification: it is what every production RL stack does, and the reasons are the same.

The interesting part is what the asynchrony does to the *learning rule*, and it is here that the evolutionary setting diverges from the RL one.

Comparison with asynchronous RL post-training. The asynchrony forces single-rollout consumption, which is the native form for evolutionary rollouts (each offspring is scored against the population, not against siblings) rather than a concession. SAO (Hou et al., 2026) reaches the same design from the post-training side; Appendix D.9 details the two further points of contact that look like correspondences but are not.

5.3 Staleness: What Asynchrony Does to Credit Assignment

The dispatch path samples a parent, computes the M_2 estimate, and runs the M_3 decision *at dispatch time*; the resulting fitness lands some number of population updates later. Call that gap τ, so $\tau \equiv 0$ is the synchronous case. τ has *no* a priori upper bound: because the scheduler is streaming, a slow rollout can outlive an arbitrary number of faster ones, and τ is bounded only by the tail of the evaluation-time distribution. Bounding it at $W-1$ would require the generational synchronisation we rejected for throughput. We therefore characterise τ empirically rather than bounding it; the

Table 1: Best-at-budget, normalized mean ± std across all 19 tasks (3 seeds per system). Each task score is divided by the per-task maximum over all four systems before averaging, so the best system on each task scores 1.0 and the number is comparable across tasks. Per-task raw scores are in Appendix D.10. ◇ two of the 19 tasks have fewer than 3 seeds completed for AGAR (ahc015, ahc027).

Backend	OE	SE	AGAR Base	AGAR Combo
DeepSeek-V4-Flash	0.855±0.193	0.919±0.138	0.949±0.107	**0.990±0.024**◇
GLM-5.2	0.897±0.131	0.929±0.093	**0.979±0.045**	0.972±0.052

measured distribution is in Appendix D.4, where at $W{=}8$ the maximum is 22 against a naive bound of 7.

The consequence connects to Table 5: credit is assigned against a population that no longer exists. M_2's baseline statistics are τ updates old; M_3's selection rule consumes a value estimate computed against a superseded population; M_4's novelty distances are computed against a stale population; and the Meta layer consumes a rollout buffer produced under superseded guidelines — which is precisely the sense in which §4's operator is off-policy. The standard correction does not transfer: SAO's clipped importance ratio needs the two policies to be comparable *per token*, and our "policy" is a block of natural language that an update may replace wholesale. A clipping-style correction would require first defining a distance between two texts and arguing that it bounds a likelihood ratio, and we have no such argument. We report the delay instead of correcting it.

6 Experiments

We ask two questions, both of which the formalization makes answerable: (i) does the transfer help at all, under one harness and against published systems? (ii) how much of that is the accumulated substrate? The third component of the paper's evidence is Observation 1 itself, which is an algebraic identity over Eq. equation 2 and therefore requires no experiment; we state it here as a fact about the construction and leave the empirical consequences of the discount to future work.

6.1 Setup

Tasks. 19 tasks in three families: 10 competitive-programming heuristic-optimization problems (C++), 3 mathematical construction problems, and 6 systems/ML algorithm-discovery problems (Python). Median per-rollout time differs across tasks by 3.7× (269–1007 s) and score scales differ entirely, so all aggregation is on per-task normalized scores.

Models, seeds and baselines. 2 backend models (DeepSeek-V4-Flash, GLM-5.2) × 3 seeds; we treat $n{=}3$ as supporting qualitative conclusions only. Baselines are OpenEvolve and ShinkaEvolve, run under their own harnesses with task specifications ported by us, plus vanilla — our substrate with every transferred module disabled. vanilla is the scientifically important baseline; the external systems are the comparability check.

6.2 Quality at matched budget

Both instantiations of AGAR rank above both baselines on DeepSeek; Base leads on GLM with Combo close behind. The standard deviation of AGAR is noticeably lower than that of the baselines, partly because five tasks are near-ceiling for all systems (circle, cloudcast, erdos, first_ac, prism): on those tasks every normalized score is close to 1.0, which suppresses variance. The meaningful signal is on the remaining 14 tasks, and the ALE (competitive programming) family carries most of the gain.

6.3 Does the transfer work, and how much is the substrate?

The mechanisms this substrate carries fall into two groups that have to be ablated separately, because each group is held fixed inside the other group's comparison. The substrate layers (L_0–L_3)

are active in both arms of any module ablation and so cancel in its difference; conversely the transferred modules are active at every substrate level. This section reports the substrate ladder, which is the comparison we can make with complete data. The module-level decomposition is a separate measurement, and §7 explains why we do not report it.

The substrate ladder, accumulated. Each level adds exactly one thing to the one below it. L_0 is repeated sampling — rewrite the current best and nothing else; L_1 adds inspiration samples and a natural-language critic, i.e. naive evolution; L_2 adds a prioritized replay buffer (island pruning, purge, diversity slots); and L_3 adds sample-level credit assignment (a per-test-case weakness report). L_2 and L_3 are exactly the Base and Combo of Table 1, so this ladder is also what separates those two instantiations.

Table 2: Substrate ladder (raw task score, mean over 3 seeds, every cell run to the full 100 fitness evaluations). Backend: DeepSeek-V4-Flash.

Task	L_0	L_1	L_2	L_3	$L_0 \to L_3$
ahc008	0.5441	0.7380	0.5452	0.9041	$+66.2\%$
ahc015	0.9687	0.9835	0.9336	0.8284	-14.5%
ahc016	0.6670	0.7316	0.9160	1.0188	$+52.7\%$
ahc024	0.8659	0.8819	0.8774	0.8636	-0.3%
ahc039	0.8797	0.8978	0.9404	0.8843	$+0.5\%$
Mean	**0.7851**	**0.8465**	**0.8425**	**0.8998**	$\mathbf{+14.6\%}$

This ladder must be read carefully, and we read it narrowly. End-to-end ($L_0 \to L_3$) the mean gains $+14.6\%$, and L_0 is a genuine floor: it is the substrate running a bare rewrite loop with every accumulated mechanism switched off. That is the comparison which shows the mechanisms are doing work rather than the harness.

But **per task the ladder is not monotone**, and we do *not* claim that each added level helps. $L_1 \to L_2$ falls on three of five tasks and the overall mean dips slightly ($0.8465 \to 0.8425$); ahc015 declines from L_1 through L_3 ($0.9835 \to 0.8284$) and ahc024 is flat. Two of the five tasks carry essentially the whole mean gain (ahc008 $+66.2\%$, ahc016 $+52.7\%$). At n=3 the level-to-level differences sit inside the noise, and the per-task non-monotonicity is the direct evidence of that. What the table supports is that the accumulated stack is worth $+14.6\%$ over the bare loop — not an attribution of that figure to any one level.

7 Discussion

What the transfer does and does not establish. The formalization of §3 buys three things: a precise sense in which the prior art is a special case (Observation 1), a factorization of the loop into modules that the objective constrains but does not uniquely determine, and a vocabulary in which mechanisms from the RL literature can be stated, implemented and compared inside an evolutionary loop. What it buys, in one sentence, is that the Bellman recursion becomes *definable* over evolutionary search. What it does not buy is an *empirical* backup: the value the recursion consumes is estimated in text space rather than learned from observed returns. The two data-backed limitations and the full unverified list appear in Appendix D.8.

Future work. The most direct next step is wiring the TD correction into the loop, turning M_2 from a one-shot estimator into a learned one. Beyond that: importance correction for staleness (Hou et al., 2026); cross-model critics; leave-one-out module ablation at larger n; standardizing the V scale so $\gamma{>}0$ can be tested cleanly; and using the 180-configuration space of §5.1 to ask which regions of mechanism space are worth exploring — a question that only becomes askable once mechanisms are interchangeable.

REFERENCES

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, 2002.

Aurélien Garivier and Eric Moulines. On upper-confidence bound policies for switching bandit problems. In *Algorithmic Learning Theory*, 2011.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, 2018.

Zhenyu Hou, Yujiang Li, Jie Tang, and Yuxiao Dong. Single-rollout asynchronous optimization for agentic reinforcement learning. *arXiv preprint arXiv:2607.07508*, 2026.

Jiachen Jiang, Huminhao Zhu, and Zhihui Zhu. SMCEvolve: Principled scientific discovery via sequential Monte Carlo evolution. *arXiv preprint arXiv:2605.15308*, 2026.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z. Pan, Alexander Du, Kurt Keutzer, Alvin Cheung, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. EvoX: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026.

Xufang Luo, Yuge Zhang, Zhiyuan He, Zilong Wang, Siyun Zhao, Dongsheng Li, Luna K. Qiu, and Yuqing Yang. Agent Lightning: Train ANY AI agents with reinforcement learning. *arXiv preprint arXiv:2508.03680*, 2025.

miles Contributors. miles: A production-grade framework for reinforcement learning on large language models. `https://github.com/radixark/miles`, 2025. GitHub repository. SGLang-native RL post-training backend for large-scale MoE and dense models; supports asynchronous and off-policy training, rollout routing and partial-rollout resumption.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenClaw-RL Contributors. Openclaw-RL: An open reinforcement learning stack for long-horizon agents. `https://github.com/openclaw/openclaw-rl`, 2026. GitHub repository.

Long Ouyang, Jeffrey Wu, Xu Jiang, et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, 2022.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *International Conference on Learning Representations*, 2016.

Zhihong Shao, Peiyi Wang, Qihao Zhu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Asankhaya Sharma. OpenEvolve: An open-source evolutionary coding agent. `https://github.com/algorithmicsuperintelligence/openevolve`, 2025. GitHub repository.

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

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.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025.

Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An LLM post-training framework for RL scaling. `https://github.com/THUDM/slime`, 2025. GitHub repository.

A Statement on the Use of AI Assistants

LLMs were used in this work in two distinct roles, which we separate deliberately.

As the object of study. The system described in this paper invokes large language models as its mutation operator, as its module policies, and as the critics of §4. These uses are the subject matter of the paper and are documented in §3–§5 and Appendix B, including model identifiers, decoding parameters and call accounting.

As a writing and coding aid. The authors used an LLM-based assistant for implementation and analysis support: refactoring module code, writing the experiment submission and result-aggregation scripts, computing the summary statistics reported in §6.3 from the raw run artifacts, and revising LaTeX prose. It was also used to audit consistency between the manuscript and the implementation, which is how several discrepancies between the two were located. All scientific claims, experimental designs, analyses and conclusions are the authors'. The authors verified every reported number against the raw experimental artifacts and take full responsibility for the content of the paper, including any errors.

No AI system is listed as an author.

B Reproducibility Statement

§6.1 gives the task suite, the two backend models and the budget. Table 1 reports the normalised mean across all 19 tasks per condition (each task score divided by the per-task maximum, then averaged); per-task raw scores and the full two-backend detail are in Appendix D.10. Table 2 reports the same for each substrate level. Seeds are fixed at 42, 123 and 456 throughout and control population initialization and sampling order; every condition is three seeds $\times$ 100 fitness evaluations, with no exceptions.

The metric is best-at-budget on each task's own objective, normalized per task for aggregation only. The saturation criterion — all four conditions within 1% — was applied before ranks were inspected, and the tasks it excludes are marked in Table 1. Decoding uses temperature 0.7 and a $393{,}216$-token cap; top-p is left at the serving default rather than pinned, which we note because it is part of the sampling distribution we do not control. Both backends are hosted models rather than frozen artifacts, so exact reproduction of absolute scores may not be possible; we regard the relative comparisons as the reproducible content.

§6.3 states which conclusions are limited by sample size: at n=3 the level-to-level differences of the substrate ladder sit inside the noise, which is why we report the end-to-end figure and decline to attribute it to any single level, and why no per-module leave-one-out decomposition is reported. Baseline numbers are obtained by us running OpenEvolve and ShinkaEvolve under our own evaluation protocol, with task specifications ported by us, rather than copied from the cited papers. Those ports are part of the release, since the baseline conditions are other people's systems and should be independently auditable. Runs execute in a container image released with the code, each allocated four CPU cores and 16 GiB of memory with no GPU; per-run LLM call counts and token counts are logged. The framework, all module implementations, every task specification, the exact configuration file for each reported run, and the raw logs will be released publicly upon publication. Each run directory in the release contains the configuration actually used, so any reported cell traces to the settings that produced it rather than to a described protocol.

C Ethics Statement

This work studies automated algorithm discovery on benchmark optimization problems: no human subjects, no personal data, no demographically sensitive attributes. Every generated program is executed in a sandboxed evaluation environment and none is deployed. Compute expenditure is the one cost we consider material, and we quantify it rather than elide it.

D Additional Material

D.1 Module interfaces

Each module is a class implementing the substrate's `Module` protocol, invoked with two arguments: a read-only `context` (the experiment-level world — population snapshot, configuration, accessors) and a writable `result` (the per-rollout trajectory). That split is what makes the concurrency safe (§5.2): a worker never writes the world, and the controller never reads a worker's intermediate state.

The data contract below is taken from the implementations rather than paraphrased, because it is what makes the module boundary checkable: a component's identity is which field of a_t it writes.

Module	**Reads**	**Writes**
M_1 credit	population accessor `rl_evolve` metadata	`m1_prefix` (str)
M_2 value	population accessor `child_stats` lineage tree	`m2_ranking` (list) `m2_v_scores` (dict) `m2_prefix` (str)

M_3 exploration	population accessor
	`m2_ranking`
	`m2_v_scores`
		`result.selection`
		`m3_prefix` (str)
		`m3_decision` (str)

M_4 memory	rollout history	retrieved context for M_1–M_3

Two structural facts are visible in this table and are worth stating because they are the content of §3's claim rather than implementation detail. First, M_3 is the only module that writes `result.selection`: it is the sole Actor, while M_2 writes a ranking and never a decision. Second, M_4 writes no prefix field — it is retrieved context consumed by the other three, which is why it carries no θ_i and is not a factor of the action.

The modules execute strictly in sequence, $M_1 \rightarrow M_2 \rightarrow M_3$, then generation, evaluation and judging. M_3 depends on M_2's ranking; M_1 and M_2 are mutually independent and could in principle run concurrently, but the implementation does not do this and we do not claim it.

D.2 Reward and entropy details

Defaults for Eq. equation 2: $\gamma = 0.95$ (with $\gamma = 0$ as the degenerate arm of Observation 1), $w_{\max} = 0.7$, $w_{\text{mean}} = 0.3$, $\lambda = 0.1$.

$H(\pi)$ is instantiated as the mean pairwise cosine distance between the feature vectors of population members, $1 - \langle v_i, v_j \rangle / (\|v_i\| \|v_j\|)$, averaged over all pairs with both vectors present. Programs without a feature vector are skipped rather than imputed. The quantity costs no additional LLM calls and M_3 already monitors it for its diversity floor, which is why the entropy term is measured at every step rather than estimated.

We report no sensitivity sweep over λ. The weights above were fixed before the reported runs and not tuned against them; given that $n{=}3$ leaves the level-to-level differences of Table 2 inside the noise, a sweep at this sample size would not be able to separate λ settings either, and we prefer to state that than to report a sweep that cannot resolve what it appears to measure.

D.3 Task suite

Nineteen tasks in three families. The competitive-programming family is drawn from AtCoder Heuristic Contest problems (ahc008–ahc046), scored by each contest's own objective. The mathematical-construction family (erdos, first_ac, second_ac) scores a constructed object against a target bound. The systems/ML family (circle, cloudcast, eplb, llm_sql, prism, txn) scores a discovered algorithm on a held-out workload.

Score scales are not comparable across tasks — prism reports values near 26, cloudcast near 0.0016, txn in the thousands — so every aggregate in this paper is computed on per-task normalized scores and every table reports raw scores per task. Median per-rollout wall-clock time ranges from 269 s (eplb) to 1007 s (ahc016), a $3.7\times$ spread; this is the straggler profile that motivates streaming dispatch (§5.2).

Six tasks are classified *saturated* — circle, cloudcast, erdos, first_ac, prism on both backends, plus ahc026, which reaches the same ceiling by 200 evaluations. The criterion is that all four conditions fall within 1%, and it was applied before ranks were inspected. The independent evidence that this classification is not post-hoc is the behaviour of the substrate ladder, where the saturated tasks cluster at a ceiling across every level.

D.4 Staleness measurements

Table 3 gives the empirical distribution of τ under the W-sweep. τ is computed at landing time as the population-version at landing minus the version stamped at dispatch. At $W{=}1$ the scheduler is synchronous, so $\tau \equiv 0$ strictly — this is a calibration check on the instrumentation itself; every row with $W{>}1$ is therefore a measurement, not an accounting artefact.

Table 3: Distribution of τ (number of population updates between dispatch and landing), pooled over 3 seeds per configuration. The last column is the fraction of rollouts that exceed the naive bound $W{-}1$.

Task	W	n	**Mean**	**Median**	**p95**	**Max**	$> W{-}1$
ahc011 (C++)	1	60	0.00	0	0	0	0.0%
ahc011 (C++)	2	330	0.90	1	2	5	13.0%
ahc011 (C++)	4	322	2.74	3	5	10	20.8%
ahc011 (C++)	8	333	6.05	6	12	22	18.6%
eplb (Python)	1	250	0.00	0	0	0	0.0%
eplb (Python)	2	306	0.97	1	2	3	15.4%
eplb (Python)	4	90	2.46	2	5	6	21.1%

Two patterns are worth noting. First, the mean tracks $0.88\,(W{-}1)$ on both tasks across the full sweep — well below the naive $W{-}1$ scaling but clearly non-zero, so staleness is real and cannot be treated as negligible. Second, the tail is heavy and does not converge: at $W{=}8$ roughly one in five rollouts exceeds $W{-}1$, and the maximum of 22 is more than three times the naive bound, a value that would require generational synchronisation to suppress.

Completion detection is by insertion-order scan with a 5 s polling timeout, so rollouts finishing inside the same poll window fold into the population in dispatch order rather than true completion

order. Against a median rollout time of several hundred seconds this biases the measured distribution negligibly, but it is a measurement convention rather than an exact observable and we state it as such.

The originally planned second column — the relationship between τ and M_2 estimation error — **is not reported**. τ is written to the run log but is not threaded into M_2's statistics, so the per-rollout correspondence between the two quantities is not available. Establishing it would require plumbing the dispatch version number through to the value estimator, which we have not done.

D.5 TEXTGRAD OPERATOR DEFINITIONS

For each module i the textual gradient operator is $G_i : \mathcal{T} \times \mathcal{C} \times \mathbb{R} \to \mathcal{T}$ and the update operator is $U : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$, where $\mathcal{T}$ is the space of natural-language strings and $\mathcal{C}$ is the rollout context accumulated since the last update. One update cycle is

$$g_i^{(t)} = G_i\big(\theta_i^{(t)},\, c^{(t)},\, \hat{r}^{(t)}\big), \qquad \theta_i^{(t+1)} = U\big(\theta_i^{(t)},\, g_i^{(t)}\big). \tag{4}$$

In TextGrad's terms, (G_i, U) is the i-th module's step and the backward pass over the pseudo-graph is the period's worth of rollouts: the loss node is the evaluator's score, the intermediate nodes are the module outputs that conditioned the candidate, and the leaf being optimized is θ_i.

The RL-to-text correspondence:

	RL	**AGAR**
Policy parameters	network weights θ	module guidelines $\theta_i \in \mathcal{T}$
Action	$a_t \sim \pi_\theta(\cdot\|s_t)$	prefix fragment $m_i^{(t)}$
Policy improvement	gradient step on θ	$U(\theta_i, G_i(\cdot))$

with the module's per-step behaviour given by $m_i^{(t)} = \mathrm{LLM}\big(\theta_i^{(t)},\, s_t,\, M_4.\mathrm{memory}\big)$.

D.6 RELATED SYSTEMS

Table 4: AGAR relative to the closest prior systems. The pattern is the same in each row: prior work fixes a rule that our formalization turns into a component optimized against J.

System	**What it does**	**Difference in AGAR**
AlphaEvolve (Novikov et al., 2025)	Fixed prompt template + evolutionary search	Prefix is learnable
ShinkaEvolve (Lange et al., 2025)	Bandit over LLMs + novelty sampling	Full tool transfer, not only a bandit
EvoX (Liu et al., 2026)	Co-evolves solutions and search strategy	Strategy improved by a critic against explicit J, not by selection
SMC-Evolve (Jiang et al., 2026)	Principled sampling, fixed rules	Rules are learned, not fixed
TextGrad (Yuksekgonul et al., 2024)	General-purpose text optimization	Coupled to an RL objective in an evolutionary loop
RLHF / GRPO (Ouyang et al., 2022; Shao et al., 2024)	RL to train LLM parameters	No parameter training; text-space optimization

D.7 THE TRANSFER

D.8 LIMITATIONS (FULL LIST)

- **Per-module attribution is unresolved.** The substrate ladder (Table 2) reports end-to-end gain but no leave-one-out decomposition of the four transferred modules; which of M_1–M_4 carries the gain is open.

Table 5: The transfer. Each RL tool keeps its function but changes its form to work without an empirical backup (column 4 is the honest column).

RL tool	Rule replaced	Module	What changed
Credit assignment	Success never attributed	M_1	Textual attribution along genealogy, not numerical backup along chain
Value estimation	Must evaluate to rank	M_2	Pre-evaluation *ranking*; scalar V estimated, not learned
Adaptive exploration	Fixed temperature	M_3	Discrete explore/exploit decision + parent choice
Entropy regularization	Island diversity	M_3 (floor)	Explicit $\lambda H(\pi)$ in Eq. equation 2
Experience replay	Population *is* memory	M_4	Two tiers: durable abstract rules; per-direction path detail
Policy improvement	Hand-tuned hyperparams	Meta: TextGrad	Text-space improvement operator (§4)
TD correction	—	—	$\delta_t = r_t^{\text{true}} - \hat{r}_t$: formalized, not wired in

- M_2 **is an estimator, not a value function.** It is not trained against realized descendant fitness, so the γV_{M_2} term carries whatever bias the text-space heuristic has, with no mechanism to correct it over the run.
- V**'s "orthogonality" is partly self-inflicted.** V correlates negatively with fitness, which we read as carrying information fitness cannot see; but V is also what selects parents, and those descendants' gains are what define V. The correlation is in part manufactured by the selection rule itself.
- γ **only discriminates among individuals with descendant evidence.** `lineage_return` carries a horizon: programs born within the last horizon iterations score $V{=}0$ and the ranking tie-breaks back to fitness, so γ's discriminative power applies to *older* individuals.
- **Unequal LLM budget.** At matched evaluations we spend 2–4$\times$ the baselines' calls; we report no LLM-call-matched arm.
- **TextGrad has no convergence guarantee**, and its measured effect is confounded by the redundant signal path identified in §4.
- M_2 **bootstrapping.** The estimator calibrates itself with its own estimates; we observe no divergence but offer no argument that it cannot occur.
- **Critic and module share a backend model**, so self-evaluation blind spots are possible; a cross-model critic is left to future work.
- **Scope.** 3 seeds support qualitative conclusions only; 2 backend models; main runs are single-worker.
- **Baseline fairness.** Baseline systems are ported by us; our ports are released so the conditions can be checked independently.

D.9 Extensibility case study

The integration reported in §5.1(b) replaces M_3's LLM explore/exploit decision with the discounted-UCB bandit of Garivier & Moulines (2011). The arms are EXPLOIT and EXPLORE; on each update every arm's discounted pull count and reward sum decay by a factor g before the played arm is credited, and the decision is $\arg\max_a \; s_a/n_a + c\sqrt{\log(N{+}1)/n_a}$. Cold start plays the least-pulled arm until both have at least one discounted pull.

The diff is 236 lines in one new module file. The new class subclasses the existing M_3 and overrides parent selection; it preserves the output contract exactly (`result.selection` plus the `m3_prefix` and `m3_decision` metadata of Appendix D.1). Neither the controller nor the module protocol is modified: the substrate contains zero references to this mechanism, and it is selected by one configuration key.

Integrating it also *deleted* 26 lines — the hand-written budget and stagnation decay heuristic that the discounting subsumes. That deletion is the part we would point to: the claim of §5.1 is not that new

mechanisms are cheap to add, but that adding a principled one can remove a hand-tuned constant, which is the same claim the paper makes about the transfer as a whole.

We note the limit of this case study honestly. The bandit is off by default and is not used in any reported run (§7), so what this demonstrates is that the interface admits the mechanism at the stated cost — not that the mechanism helps.

D.10 PER-TASK RESULTS, FULL DETAIL

Raw scores (mean over 3 seeds unless noted) for both backends. † = saturated (all systems within 1%). ◇ = fewer than 3 seeds completed.

Task	OE	SE	AGAR Base	AGAR Combo	**win?**
Backend: DeepSeek-V4-Flash					
ahc008	0.6659	0.8511	0.5452	0.9041	Combo
ahc011	0.4083	0.4635	0.4886	0.6515	both
ahc015	0.8593	0.8087	0.9999◇	0.8978◇	both
ahc016	0.6972	0.8602	0.9160	1.0188	both
ahc024	0.6464	0.8327	0.8774	0.8636	both
ahc025	0.6944	1.3799	2.616	3.0896	both
ahc026	0.9671	0.9371	0.990	0.988	both
ahc027	1.2421	1.5329	1.590	1.541	both
ahc039	0.8748	0.9165	0.961	0.937	both
ahc046	0.9729	0.9602	1.022	1.028	both
circle†	0.9948	1.0001	0.9999	0.9994	sat.
cloudcast†	0.0016	0.0016	0.0016	0.0016	sat.
eplb	0.1519	0.1656	0.1643	0.1645	—
erdos†	0.9915	0.9997	0.9991	0.9966	sat.
first_ac†	0.9936	0.9959	0.9920	0.9942	sat.
llm_sql	0.7286	0.7322	0.7343	0.7436	both
prism†	26.273	26.364	26.282	26.274	sat.
second_ac	0.9914	1.0099	1.0130	1.0369	both
txn	4090.8	4392.7	4351	4419	Combo
Backend: GLM-5.2					
ahc008	0.3700	0.5646	0.5965	0.6288	both
ahc011	0.4213	0.3933	0.4483	0.4517	both
ahc015	0.8472	0.8835	1.0355	0.9145	both
ahc016	0.4476	0.4792	0.5985	0.6744	both
ahc024	0.7120	0.8250	0.6968	0.8376	Combo
ahc025	0.1803	0.2002	0.2734	0.2208	both
ahc026	0.9742	0.9522	0.9745	0.9829	both
ahc027	1.3306	1.1134	1.3183	1.2978	—
ahc039	0.5874	0.6232	0.7142	0.6408	both
ahc046	0.9964	0.9621	1.0154	1.0136	both
circle†	0.9939	0.9991	0.9969	0.9996	sat.
cloudcast†	0.0016	0.0016	0.0016	0.0016	sat.
eplb	0.1586	0.1678	0.1606	0.1611	—
erdos†	0.9996	0.9998	0.9995	0.9996	sat.
first_ac†	0.9931	0.9968	0.9973	0.9979	sat.
llm_sql	0.6797	0.7309	0.7407	0.7287	Base
prism†	26.542	26.511	26.546	26.610	sat.
second_ac	1.0106	1.0274	1.0339	1.0185	Base
txn	3990.6	4335.3	4374.7	4299.3	Base