SIA-W: Self-Improving Agents with Test-Time Weight Updates

Yogendra Manawat [1] **Prannay Hebbar** [1] **Samuel Verboomen** [1] **Selvam Palanimalai** [1] **Kunal Bhatia** [1]
Vignesh Baskaran [1]

Abstract

We introduce SIA-W, a self-improving agent framework that jointly optimises the agent scaffold and the model weights. Scaffold iteration is a powerful first lever: by evolving tools, prompts, and execution harnesses across generations, the agent rapidly builds domain-adapted search and reasoning procedures. SIA-W then compounds these gains with a second lever, test-time RL, which adapts the model weights directly on task feedback once the scaffold has matured. Across three diverse research tasks spanning law (charge classification), systems (GPU kernel optimisation), and biology (single-cell denoising), combining both levers delivers substantial gains over scaffold iteration alone: **16 percentage points** on LawBench, **19%** runtime reduction on GPU kernels, and **19%** improvement on denoising, with weight updates surfacing domain knowledge that complements what the harness builds.

1. Introduction

LLM agents have two complementary axes of improvement: the harness surrounding the model, and the model's own policy weights. Harness improvements raise the agent's ability to search, reason, use tools, verify, and interact with its environment. Policy improvements change what solutions the agent considers, what abstractions it prefers, and what reasoning paths it treats as promising.

Prompt engineering, tool design, and search scaffolds extend what the agent can accomplish without retraining (Wei et al., 2022; Yao et al., 2022; Motwani et al., 2026). For domain-specific tasks, modifying the model's weights via *test-time training* (Yuksekgonul et al., 2026) offers a complementary lever, adapting the model's policy to the reward structure of the domain. Rather than treating each task as isolated, the agent uses its accumulated experience to shape future behaviour, compressing what it learns from problems back into its own search process.

[1]Hexo.ai. Correspondence to: Vignesh Baskaran <vignesh@hexo.ai>.

Accepted at the ICML 2026 AI4Research Workshop. Copyright 2026 by the author(s).

2. Method

System components. SIA consists of three components operating in a generational loop (Figure 2) (Hong et al., 2023; Lee et al., 2026):

1. **Meta-Agent.** Initialises the first target-agent scaffold A_1 from sample task descriptions and any reference implementations supplied with the benchmark.

2. **Target Agent.** Executes against dataset $\mathcal{D}$ inside a sandbox with read-only access to the dataset directory and read/write access to a working directory.

3. **Feedback Agent.** Reads target-agent trajectories τ_g, identifies failure modes and architectural weaknesses, and synthesises an improved scaffold A_{g+1}.

Per-generation protocol. Each generation g follows a three-phase protocol.

1. Execution Phase. A_g runs on $\mathcal{D}$ under strict sandboxing: read-only access to the dataset directory, read/write access to the working directory, and a complete trajectory τ_g logged for downstream analysis.

2. Analysis Phase. The feedback agent $\mathcal{F}$ receives four inputs: (i) A_g's source code, (ii) the complete trajectory τ_g (messages, tool calls, results), (iii) performance metrics and error logs $\mathcal{M}_g$, and (iv) optionally sample task descriptions used to discourage single-instance overfitting.

3. Improvement Phase. $\mathcal{F}$ produces two artefacts: an *improvement report* (prose analysis and the proposed changes) and the *next-generation agent* A_{g+1} (the implemented scaffold). This yields the recurrence

$$A_{g+1} \;=\; \mathcal{F}(A_g,\, \tau_g,\, \mathcal{M}_g,\, \mathcal{T}).$$

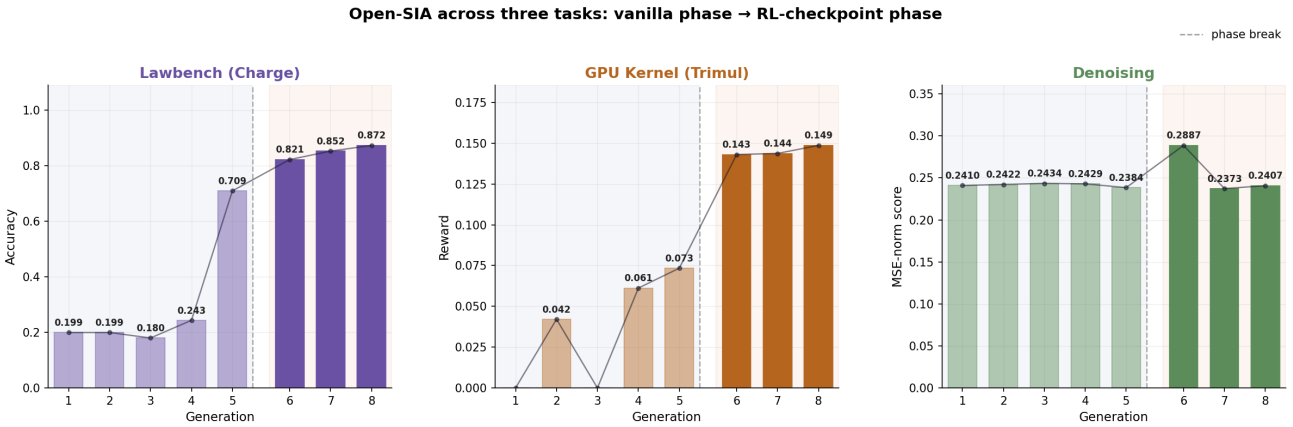

Figure 1. **SIA-W across three diverse tasks.** Per-generation primary metric for LawBench (top-1 accuracy), GPU-Mode TriMul (graded reward), and Denoising (`mse_norm`). The dashed line marks the phase break between Phase 1 (base `gpt-oss-120b`, harness iteration only) and Phase 2 (Tinker RL-trained checkpoints, harness frozen). All three tasks show a clear lift in the checkpoint phase.

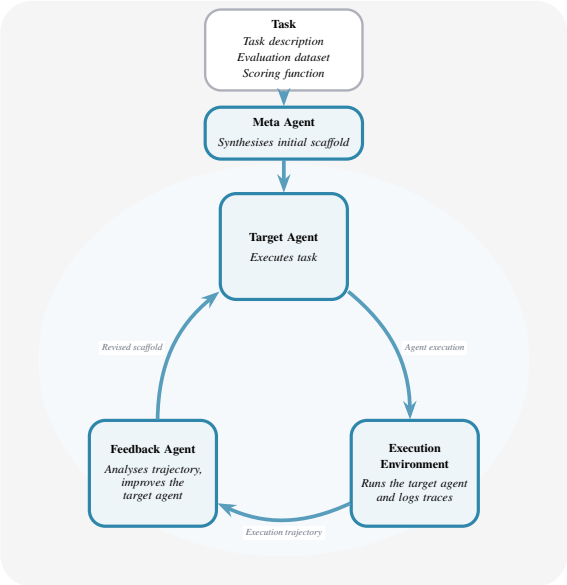

Figure 2. **SIA system architecture.** The meta-agent initialises a scaffold from the provided task specification. Each generation, the execution environment runs the target agent against the evaluation dataset and captures a complete interaction trajectory. The feedback agent analyses this trajectory, identifies failure modes, and synthesises an improved scaffold for the next generation. The loop is fully autonomous: no human is involved between generations.

Two-phase loop. SIA-W alternates between two improvement operators. **Phase 1 (harness improvement):** the generational loop above runs with the base model (`gpt-oss-120b`, OpenAI's open-source 120B model) held fixed; only the scaffold evolves (Lee et al., 2026; Lu et al., 2024; Zhang et al., 2025; Hu et al., 2024). **Phase 2 (test-time weight updates):** once the scaffold plateaus, the harness is frozen and successive RL checkpoints (Schulman et al., 2017; Shao et al., 2024; Chen et al., 2024) trained on the target benchmark are swapped into the same scaffold, adapting the model's policy to the domain.

Training objective. We optimise an entropic-utility objective (Yuksekgonul et al., 2026). For each prompt s we sample G rollouts $\{a_i\}$ from $\pi_{\theta_{\text{old}}}$ and compute task rewards $R(s, a_i)$. The objective is

$$J_\beta(\theta) \;=\; \mathbb{E}_s\big[\log \mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\big[\exp(\beta(s)\, R(s,a))\big]\big]\,,$$

whose gradient is the softmax-weighted policy gradient

$$\nabla_\theta J_\beta \;=\; \mathbb{E}[w_\beta(a;s)\,\nabla_\theta \log \pi_\theta(a \mid s)]\,,$$
$$w_\beta(a;s) \;=\; \tfrac{\exp(\beta(s)R(s,a))}{\mathbb{E}_{a'}\exp(\beta(s)R(s,a'))}.$$

Two stabilization terms are added to the objective:

- **IS correction.** A per-token importance-sampling ratio $\pi_\theta(a \mid s)/\pi_{\text{old}}(a \mid s)$ corrects for mismatch between the sampler (rollout policy) and the learner (current policy).
- **KL-to-base penalty.** $-\lambda \log \dfrac{\pi_\theta(a \mid s)}{\pi_{\theta_0}(a \mid s)}$ enforces a trust region around the frozen reference policy π_{θ_0}; λ controls its strength.

Adaptive temperature $\beta(s)$. Rather than a fixed inverse-temperature, $\beta(s)$ is tuned *per initial state* via bisection on $[0, \beta_{\max}]$ so that the softmax-reweighted distribution $q_\beta(\cdot \mid s) \propto \pi_\theta(\cdot \mid s)\exp(\beta R)$ satisfies a fixed KL budget:

$$\mathrm{KL}\big[q_\beta(\cdot \mid s) \;\big\|\; \pi_\theta(\cdot \mid s)\big] = \gamma.$$

This single knob interpolates two regimes: as $\beta \to 0$ the objective reduces to ordinary expected-reward maximization; as $\beta \to \infty$ it approaches best-of-N selection. The effect is that gradient mass concentrates on the highest-reward rollouts without the instability of a clipped surrogate (Schulman et al., 2017).

Why this combination. Harness improvement quickly captures *externalised* fixes (parsing, retries, search procedures, tool selection) that any sufficiently strong base model can use. Weight updates capture *internalised* structure, hyperparameter regions, kernel patterns, label disambiguation, that a fixed model cannot reach by editing its scaffold alone. The two operators work in tandem: SIA-W layers them in sequence. In our experiments, generational gains became marginal after generation 5 across all three tasks, so we capped Phase 1 at 5 generations; the RL phase then begins from this already-strong harness.

Verification and reproducibility. Every generation's score comes from the same deterministic grader the orchestrator uses during training: LawBench top-1 accuracy on the 913-example held-out test split, GPU-Mode TriMul reward $1500/\text{runtime}_{\mu s}$ on H100 via Modal with fixed input shapes, and `mse_norm` against ground-truth pancreas counts. Phase 2 rollouts are on-policy (sampler weights = learner weights at each step), and the checkpoint-selection rule (App. H) draws weights only from steps where mean reward is rising and max reward has not collapsed, so the reported Phase 2 numbers are not cherry-picked from transient peaks. Hyperparameters, prompts, the meta-/feedback-/debug-agent transcripts, and the exact command to reproduce a generation are documented in App. F.6.

3. Experiments

We evaluate SIA-W on three contrasting tasks spanning law, systems, and biology: a 191-class Chinese charge-classification benchmark, a Triton kernel-optimisation benchmark on H100, and a single-cell RNA denoising benchmark.

Across all experiments, the meta-agent uses `Claude Sonnet 4.6`, while the target-agent uses `gpt-oss-120b`.

3.1. LawBench (191-class classification)

LawBench (Fei et al., 2023) asks a model to identify the correct criminal charge from 191 possible categories in Chinese law. The benchmark contains 5,332 training samples and 913 test samples, and all evaluations are performed exclusively on the held-out test split to ensure no overlap between training and evaluation data. A random guess would be right less than one percent of the time.

Phase 1 (harness only). The agent refined its approach across five generations. Early generations established a working pipeline, and by generation 5 the harness had been restructured around a classical text-classification approach. Accuracy reached about **71%**, a **51 percentage point gain** from the initial run.

Phase 2 (weight updates). Successive weight updates pushed accuracy further to **87%**, an additional **16 percentage point gain** over the Phase 1 peak (Figure 3).

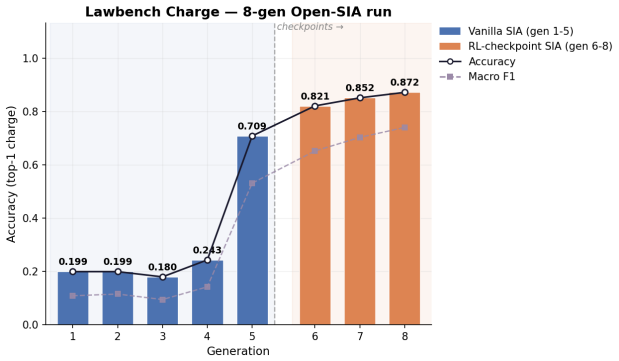

Figure 3. **LawBench across eight SIA-W generations.** Bars: top-1 accuracy. Solid line: macro-F1. Phase 1 (gens 1–5, blue) iterates only the harness; Phase 2 (gens 6–8, orange) freezes the code and swaps in Tinker RL checkpoints.

3.2. GPU-Mode TriMul (GPU kernel optimisation)

The agent writes a low-level GPU kernel for a specific operation used in protein structure prediction, compiled and benchmarked on an H100 GPU. The score is inversely proportional to how long the kernel takes to run, so a higher score means a faster, more efficient kernel.

Phase 1 (harness only). The agent progressively built up working kernels across five generations, establishing a solid baseline with a best runtime of about **12,500 microseconds**.

Phase 2 (weight updates). The reward score **roughly doubled** relative to the Phase 1 peak and kernel runtime fell to about **10,100 microseconds**, a **19% reduction** driven by optimisations that emerged from domain-adapted weights (Figure 4).

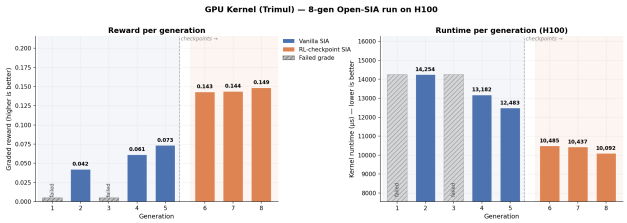

Figure 4. **GPU-Mode TriMul across eight SIA-W generations on H100.** Left axis: graded reward (higher is better). Right axis: kernel runtime in microseconds (lower is better). Phase 1 plateaus; Phase 2 doubles reward.

3.3. Statistical denoising (biology)

The agent tunes the settings of a standard denoising tool applied to single-cell biological measurement data. The

score runs from zero (no improvement over the reference tool's defaults) to one (perfect denoising).

Phase 1 (harness only). The agent explored the parameter space of the denoising tool across five generations, reaching a stable plateau by generation 5, with `mse_norm` settling between **0.236 and 0.241**.

Phase 2 (weight updates). The first weight-update checkpoint added a two-line post-processing step (`np.clip` + `np.rint`) that rounded imputed counts to integers, a change no Phase 1 scaffold proposed, lifting `mse_norm` to **0.289**, a **19% gain** over the Phase 1 best (Figure 5; details in App. F.8).

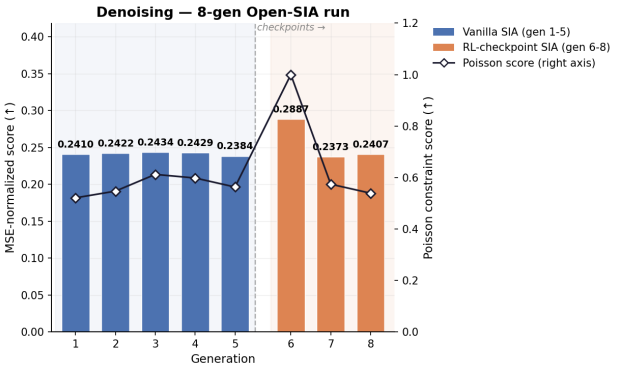

Figure 5. **Statistical denoising across eight SIA-W generations.** Bars: `mse_norm` (left axis, higher is better). Diamonds: Poisson constraint score (right axis).

4. Discussion

The same pattern holds across all three tasks: harness iteration reaches a ceiling, and weight updates push past it (16 pp on LawBench, doubled reward on GPU-Mode, 19% on denoising). The harness shapes how the agent searches; weight updates change what the model knows.

Weight updates internalize domain understanding. Two observations across the tasks show that weight updates produce genuine learning rather than just score improvements.

First, the model generalises rather than memorises. On GPU-Mode TriMul, the training dynamics (Figure 6) show a model that finds a fast kernel, briefly over-exploits it, then recovers with a broader policy, a trajectory inconsistent with memorisation of a single peak.

Second, weight updates surface knowledge that harness changes cannot reach. On denoising, the model independently discovered that outputs must be rounded to whole numbers, a requirement never encoded in any prompt or scaffold.

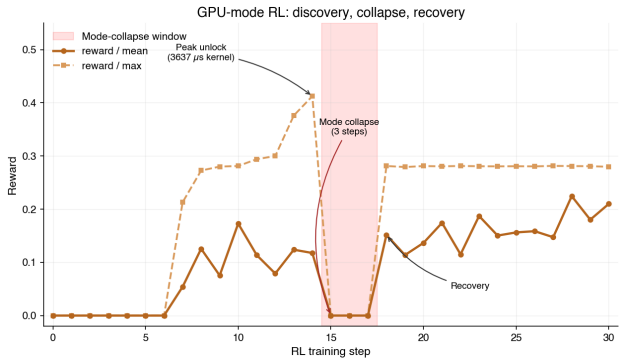

Figure 6. **GPU-Mode RL training: discover, consolidate, generalise.** Mean (solid) and max (dashed) episode reward over 31 RL steps. The policy discovers a 3,637 μs kernel at step 14 (reward peak), briefly over-exploits it (mean drops for three steps), then recovers with a more general understanding of the kernel design space.

Limitations.

- **Tool-call format drift.** The base model emits tool calls in a format that differs from what the harness parser expects, so a generation can lose a debug cycle to malformed invocations even when the underlying answer is correct.
- **Interface contract drift.** A regenerated target agent occasionally redefines its own command-line interface and drops one of the inputs the orchestrator passes in, causing the agent to exit before any inference is run. The debug agent catches this on retry.

Future directions.

1. **Interleaved harness/weight updates.** Replace the fixed sequential pipeline with alternating sub-steps so the harness adapts to the policy as the policy adapts to the harness, rather than baking in a stale scaffold when weight training begins.
2. **Per-tool-call advantage estimation.** Decompose rollout-level reward into tool-call-level advantages using the Harmony delimiters as natural segmentation points, directly targeting the credit-assignment limitation.
3. **Synchronising the feedback agent with the trained policy.** Periodically refresh the meta- and feedback-agents with the latest LoRA checkpoint so the system's reasoning about itself tracks the system that is actually acting.
4. **Action-space mutation operators.** Extend the feedback agent's edit space to include *tool-introduction* mutations (a new sub-process, a new file reader, a new grader call), rather than restricting it to logic edits over a fixed initial tool set.

References

Akyürek, E., Damani, M., Zweiger, A., Qiu, L., Guo, H., Pari, J., Kim, Y., and Andreas, J. The surprising effectiveness of test-time training for few-shot learning. *arXiv preprint arXiv:2411.07279*, 2024.

Chen, Z., Deng, Y., Yuan, H., Ji, K., and Gu, Q. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*, 2024.

Fei, Z., Shen, X., Zhu, D., Zhou, F., Han, Z., Zhang, S., Chen, K., Shen, Z., and Ge, J. LawBench: Benchmarking legal knowledge of large language models. *arXiv preprint arXiv:2309.16289*, 2023.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S. K. S., Lin, Z., et al. MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022.

Hu, S., Lu, C., and Clune, J. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Karpathy, A. AutoResearch: Autonomous machine learning experimentation. https://github.com/karpathy/autoresearch, 2026.

Lee, Y., Nair, R., Zhang, Q., Lee, K., Khattab, O., and Finn, C. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Lu, C., Lu, C., Lange, R. T., Foerster, J., Clune, J., and Ha, D. The AI scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Motwani, S. R., Nichols, D., London, C., Li, P., Pizzati, F., Blake, A., Hammoud, H., McDonald, T., Naik, A., Ivanova, A., Baskaran, V., Laptev, I., Glatt, R., Ben-Nun, T., Torr, P., Jaques, N., Prabhu, A., Bartoldson, B., Kailkhura, B., and de Witt, C. S. Longcot: Benchmarking long-horizon chain-of-thought reasoning. *arXiv preprint arXiv:2604.14140*, 2026.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J. R., Ellenberg, J. S., Wang, P., Fawzi, O., Kohli, P., and Fawzi, A. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Yuksekgonul, M., Koceja, D., Li, X., Bianchi, F., McCaleb, J., Wang, X., Kautz, J., Choi, Y., Zou, J., Guestrin, C., and Sun, Y. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

Zelikman, E., Wu, Y., Mu, J., and Goodman, N. D. STaR: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems*, volume 35, pp. 15476–15488, 2022.

Zhang, J., Hu, S., Lu, C., Lange, R., and Clune, J. Darwin gödel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025.

A. Related Work

Table 1. Comparison of self-improving / automated ML agents and their approaches.

Agent	Approach
SIA-W (ours)	RL fine-tuning of the policy on agentic rollouts
Meta-harness (Lee et al., 2026)	LLM-driven harness mutation
AutoResearch (Karpathy, 2026)	Fixed scaffold for autonomous ML research
Darwin Gödel Machine (Zhang et al., 2025)	Evolutionary search over agent source code
Discover-TTT (Akyürek et al., 2024)	Test-time training: per-task gradient adaptation
FunSearch (Romera-Paredes et al., 2024)	LLM-proposed program mutations
AI Scientist (Lu et al., 2024)	Full research pipeline automation
Voyager (Wang et al., 2023)	Iterative skill synthesis with curriculum
Self-Refine (Madaan et al., 2023) / Reflexion (Shinn et al., 2023)	Verbal self-critique loop
STaR (Zelikman et al., 2022)	Bootstrapped reasoning traces
ReAct (Yao et al., 2022)	Interleaved reasoning + tool use

B. Glossary

B.1. System Components

Meta-agent The LLM call that generates the initial target-agent scaffold A_1 from the task specification $\mathcal{T}$.

Target agent The scaffold A_g that executes against the evaluation dataset $\mathcal{D}$ and produces the execution trajectory τ_g.

Feedback agent ($\mathcal{F}$) The LLM call that reads $(A_g, \tau_g, \mathcal{M}_g, \mathcal{T})$ and produces the next-generation scaffold A_{g+1}.

Scaffold The complete Python program wrapping the LLM: system prompt, tool-dispatch logic, answer extraction, and grader calls.

Harness Used interchangeably with scaffold; specifically refers to the execution infrastructure (sandboxing, logging, retry logic) that surrounds the LLM call.

Trajectory (τ_g) The full structured log produced by running scaffold A_g against $\mathcal{D}$: every prompt, model response, tool call, and extracted answer for all task instances.

B.2. Symbol Table

Table 2. Notation used throughout the paper.

Symbol	Meaning
g	Generation index
$G_{\max}$	Maximum number of generations in the SIA loop
A_g	Agent scaffold at generation g
$\mathcal{D}$	Evaluation dataset
$\mathcal{T}$	Task specification (benchmark description and sample instances)
$\mathcal{M}_g$	Performance metrics and error logs at generation g
G	Number of rollouts sampled per state during RL training
π_θ	Current policy (model with trainable weights θ)
π_{θ_0}	Frozen reference policy (base model, unchanged during training)
s	Initial state (task prompt)
a	Action (model-generated response / rollout)
$R(s, a)$	Task reward for action a given state s
$\beta(s)$	Adaptive inverse-temperature, tuned per state s via bisection
$\beta_{\max}$	Upper bound for the bisection search over β
γ	Target KL budget used to set $\beta(s)$
λ	KL-to-base penalty coefficient
$w_\beta(a; s)$	Softmax reward weight assigned to rollout a given β

C. How SIA Works

This appendix gives an end-to-end walkthrough of the SIA architecture that produces the harness-iteration trajectories evaluated in the main text. SIA-W reuses this loop verbatim; the only difference is that the harness loop is capped at 5 generations, our experiments showed generational gains becoming marginal beyond this point, after which the LLM invoked inside the EXECUTE step is swapped for an RL checkpoint (Section 2).

C.1. Terminology

A *scaffold* is the complete Python program that wraps the LLM: its system prompt, tool-use logic, answer extraction, and evaluation harness. The *execution trajectory* τ_g is the full log produced by running the generation-g scaffold A_g against the evaluation dataset $\mathcal{D}$; it captures every prompt, model response, tool call, and extracted answer for all task instances.

C.2. Implementation Notes

- **Uniform agent interface.** Meta-, target-, and feedback-agents share an asynchronous API and a small set of custom tools (file I/O, code execution, structured logging).
- **Filesystem isolation.** Each agent operates in a prompt-enforced sandbox with explicit read/write boundaries, so a misbehaving target cannot corrupt either the dataset or the meta-agent's working directory.
- **Execution logging.** The full interaction trajectory is captured in a structured form, which is what the feedback agent consumes, not just the final score.
- **Sample-task regularisation.** The meta-agent is conditioned on a diverse set of task specifications during scaffold generation, which mitigates overfitting the initial scaffold to a single benchmark instance.

D. SIA Evolution Algorithm

Algorithm 1 gives the complete SIA loop. SIA-W uses the same outer loop; the only difference is that the harness loop is capped at 5 generations, experiments showed marginal gains beyond generation 5, and the model invoked inside EXECUTE is then replaced by a checkpoint produced by the test-time training procedure of Section 2.

Algorithm 1: SIA Evolution Loop
Input: $\mathcal{D}$ (task dataset), $\mathcal{T}$ (task specification), $G_{\max}$ (max generations)
Output: $A_{G_{\max}}$ (final evolved agent)

$A_1 \leftarrow \textsc{MetaAgent}(\mathcal{T})$ *// generate initial scaffold*
for $g = 1$ **to** $G_{\max}$ **do**
  $\tau_g \leftarrow \textsc{Execute}(A_g, \mathcal{D})$ *// run agent, collect trajectory*
  if $g < G_{\max}$ **then**
    $\mathcal{M}_g \leftarrow \textsc{Score}(\tau_g)$ *// metrics and error logs*
    $A_{g+1} \leftarrow \textsc{FeedbackAgent}(A_g, \tau_g, \mathcal{M}_g, \mathcal{T})$
  end if
end for
return $A_{G_{\max}}$

E. Comparison with AutoResearcher

AutoResearcher (Karpathy, 2026) and SIA-W both automate the research loop, but they differ in what they allow the agent to change.

In AutoResearcher, the agent architecture is fixed. A static coding agent proposes modifications to the experiment script and runs them, keeping the agent itself unchanged across iterations. Improvement comes entirely from finding better experiment configurations.

In SIA-W, the agent architecture is itself the subject of iteration. The meta-agent and feedback-agent rewrite the target agent's scaffold across generations, changing how the agent reasons, searches, and uses tools. On top of that, SIA-W adds a second phase where the model weights are updated to deepen domain understanding. The two systems therefore operate at

fundamentally different levels: AutoResearcher searches over experiment parameters; SIA-W searches over agent designs and model policies.

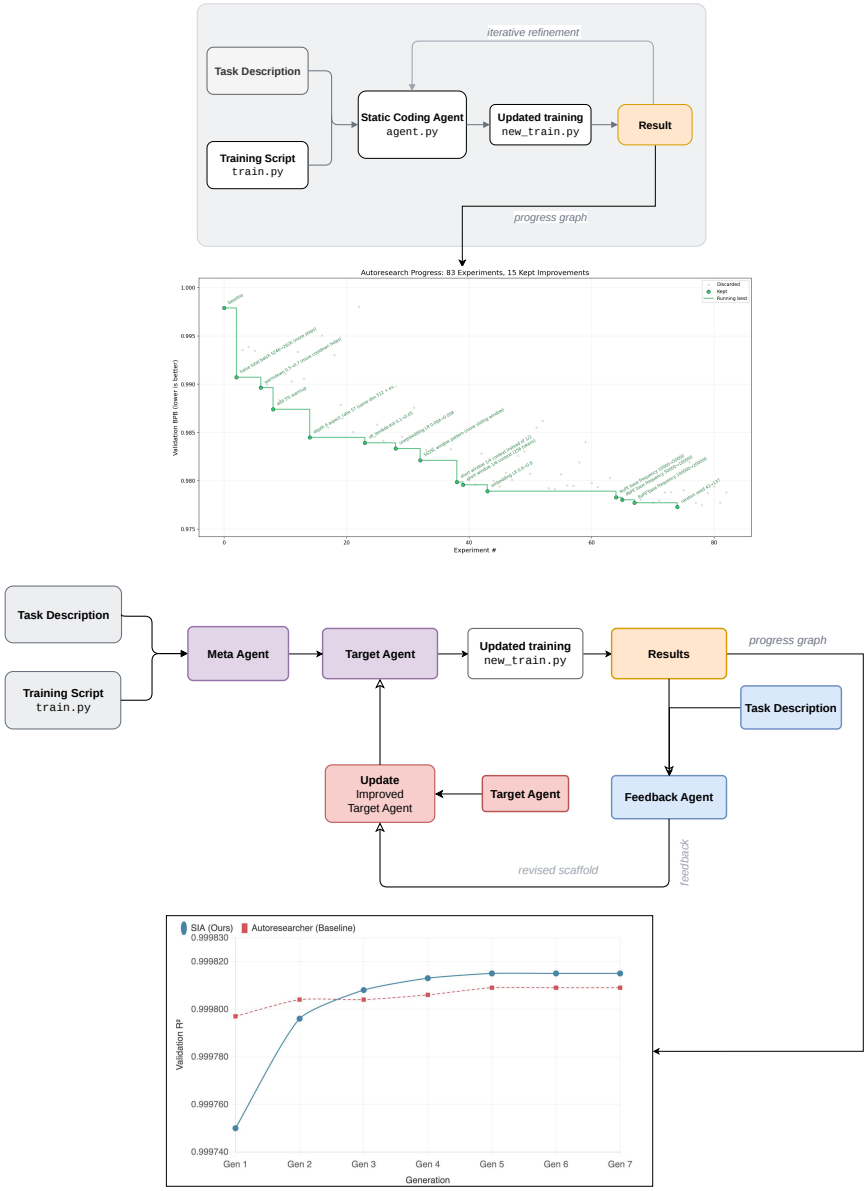

Figure 7. **AutoResearcher (top) vs. SIA-W (bottom).** In AutoResearcher, a static coding agent iteratively proposes and runs experiment modifications while the agent architecture itself remains fixed throughout. In SIA-W, the agent architecture is the subject of iteration: the meta-agent initialises the target agent, the feedback agent rewrites it from the execution trajectory; after 5 harness generations (where gains empirically became marginal), the model weights are updated via RL-adapted checkpoints. The key distinction is the level of search: AutoResearcher searches over experiment parameters; SIA-W searches over agent designs and model policies.

F. Test-Time Training: Hyperparameters and Implementation Details

This appendix collects the hyperparameters, training-loop structure, and representative artefacts behind the Phase 2 RL checkpoints used in SIA-W. The training stack runs on Tinker against `openai/gpt-oss-120b` with LoRA adapters (Hu et al., 2022); rollouts are scored by the same task verifier used in evaluation, so the loss is computed on the same metric we report.

F.1. Shared Hyperparameters

All three Phase 2 runs share the configuration in Table 3. The only knobs that vary across tasks are the sandbox `eval_timeout`, the per-task CPU allocation, and the initial-state policy (`initial_exp_type`).

Table 3. Shared training configuration across the three SIA-W Phase 2 runs. `groups_per_batch` is set to 16 for single-host runs and raised to 64 for the paper-default sweep on GPU-Mode.

Parameter	Value	Role
`model_name`	`openai/gpt-oss-120b`	Base policy
`lora_rank`	32	LoRA adapter rank
`learning_rate`	4×10^{-5}	Adam, β_1=0.9, β_2=0.95
`adv_estimator`	`entropic_adaptive_beta`	Within-group reweighting
`adv_estimator_beta`	2.0	Initial entropic temperature
`kl_penalty_coef`	0.1	KL($\pi_\theta \parallel \pi_0$) coefficient
`loss_fn`	`importance_sampling`	Per-token policy gradient
`temperature`	1.0	Sampling temperature
`max_tokens`	26,000	Per-rollout token budget
`group_size`	8	Rollouts per state (within-group baseline)
`groups_per_batch`	16–64	States per training step
`num_epochs`	5 (SIA-W) / 50 (full sweep)	Epochs over the rollout buffer
`sampler_type`	`greedy`	ε-greedy state sampler
`save_every`	5	Checkpoint cadence (steps)

F.2. Per-Task Hyperparameters

Table 4 lists the parameters that vary per task, all of which are determined by the verifier rather than the policy.

Table 4. Per-task overrides. `num_cpus` reflects the parallelism of the verifier, not of the policy. All three tasks use `initial_exp_type=none`, meaning the model writes its first solution from scratch given only the task prompt.

Task	`eval_timeout` (s)	`num_cpus`	Reward shape	Verifier
LawBench	300	2	top-1 accuracy	3-seed CV on labelled split
Denoising	600	4	mse_norm	pancreas scRNA-seq, MAGIC ref
GPU-Mode TriMul	1,000	1	$1500/\mu s$	H100 via Modal

F.3. Entropic Objective: Plain-Language Walkthrough

This subsection unpacks each formula from Section 2 in plain language.

Objective $J_\beta(\theta)$. Standard expected-reward maximisation averages reward uniformly over all rollouts. The entropic objective instead asks: if higher-reward actions were up-weighted exponentially (by $e^{\beta R}$) and re-normalised, what would the log-expected reward be under that tilted distribution? Taking the logarithm of the softmax-weighted expectation yields a smooth interpolation: as $\beta \to 0$ the objective reduces to ordinary average-reward maximisation; as $\beta \to \infty$ it approaches best-of-N selection, concentrating all gradient mass on the single highest-reward rollout.

Softmax weights $w_\beta(a; s)$. Each rollout a receives a weight proportional to $e^{\beta R(s,a)}$, normalised across the group. Rollouts with above-average reward receive weight > 1; those below average receive weight < 1. Every rollout contributes to the gradient update, but high-reward rollouts dominate: a smooth softmax over rewards rather than a hard argmax.

Importance-sampling (IS) correction. When the model that collected rollouts (π_{old}) differs from the model currently being updated (π_θ), the probability of each token-level action under the two policies can diverge. The per-token IS ratio $\pi_\theta(a \mid s)/\pi_{\text{old}}(a \mid s)$ rescales the gradient so that tokens the current policy assigns low probability receive proportionally less credit, preventing over-exploitation of actions that appeared in the buffer by chance.

KL-to-base penalty. The additive term $-\lambda \log(\pi_\theta/\pi_{\theta_0})$ penalises the policy for moving far from the frozen base model π_{θ_0}. A high-reward rollout that requires large movement from base receives a lower effective advantage than one achieving

similar reward with smaller deviation. This term acts as a trust region, preserving general-purpose capability while the LoRA adapter acquires domain knowledge.

Adaptive temperature $\beta(s)$. Rather than using a single global β, the system finds a per-state value via bisection on $[0, \beta_{\max}]$. The target is that the KL divergence from the softmax-reweighted distribution q_β back to π_θ equals a fixed budget γ. If a state's rollouts are nearly uniform in reward, β is raised to sharpen the reweighting; if they are already reward-diverse, β is lowered to avoid concentrating all mass on a single outlier. The result is that gradient mass allocation is roughly calibrated across states regardless of per-state reward variance.

F.4. Entropic Advantage Estimator

The advantage estimator is an entropic tilt rather than the standard mean-baseline used in group-relative policy gradients. For each group of G trajectories with rewards $r_1, \ldots, r_G$, the per-trajectory weight is

$$w_i \;=\; \frac{\exp(\beta\, r_i)}{\frac{1}{G}\sum_{j=1}^{G}\exp(\beta\, r_j)}, \qquad \hat{A}_i \;=\; w_i - 1.$$

With `adv_estimator_beta`$=2.0$, β is auto-tuned per group to keep the implied KL to a uniform reweighting under a target δ, falling back to a bisection between $[0, \beta_{\max}]$ when the upper bound is too tight. As β grows, almost all gradient mass concentrates on the highest-reward trajectory in the group. This makes the estimator *within-group reweighting*, not an importance-sampling correction for off-policy drift; off-policy stability is provided by the LoRA-rank constraint and the KL-to-base penalty (Section 2).

F.5. Training Loop

The Phase 2 inner loop is the standard Tinker rollout / forward-backward schedule. Each step does:

Algorithm 2: SIA-W Phase 2 RL Step

1: Sample `groups_per_batch` states from the replay buffer (`greedy`).
2: For each state, generate `group_size` rollouts at temperature 1.0.
3: Score each rollout with the task verifier (sandboxed, CPU-pinned, $\leq$`eval_timeout` s).
4: Drop groups with constant reward (`remove_constant_reward_groups`).
5: Compute entropic advantages within each group (auto-tuned β).
6: Add per-token KL penalty against π_0: $\hat{A}_i \mathrel{+}= \lambda\,(\delta_i - \bar{\delta})$, with $\delta_i = \log \pi_0 - \log \pi_\theta$ (Section 2).
7: Run a forward-backward pass and one Adam step on the LoRA adapter.
8: Update the replay buffer with any new best-of-group solutions.
9: Every `save_every` steps, snapshot the adapter as a Tinker checkpoint URI.

F.6. Reproducing a Run

A representative training invocation (denoising) is reproduced below verbatim from the codebase:

```
PYTHONPATH=.  python3 -m tinker_cookbook.recipes.ttt.train \
  env=denoising \
  problem_idx=denoising \
  model_name=openai/gpt-oss-120b \
  sampler_type=greedy \
  initial_exp_type=none \
  group_size=8 groups_per_batch=16 \
  learning_rate=4e-5 num_epochs=5 \
  eval_timeout=600 num_cpus_per_task=4 \
  adv_estimator=entropic adv_estimator_beta=2.0 \
  max_tokens=26000 eval_every=3 save_every=5
```

The GPU-Mode and LawBench invocations differ only in `env`, `problem_idx`, `eval_timeout`, `num_cpus_per_task`, and (for GPU-Mode) `gpu_mode_score_scale`$=1500.0$ which sets reward $= 1500/\text{runtime}_{\mu s}$.

F.7. Representative Generated Artefact: Triton TriMul Kernel

The Phase 2 GPU-Mode checkpoint converges on the kernel structure shown below (LayerNorm prologue; full kernel is ∼470 lines). The two patterns that repeatedly emerge across rollouts are (i) `constexpr` block sizes selected per problem dim and (ii) accumulation in fp32 with fp16 storage to stay within H100 shared-memory budgets.

```
@triton.jit
def _row_ln_fp16_kernel(
  X_ptr, Y_ptr, w_ptr, b_ptr,
  M, C: tl.constexpr, eps,
  BLOCK_M: tl.constexpr, BLOCK_C: tl.constexpr):
  pid = tl.program_id(0)
  rows = pid * BLOCK_M + tl.arange(0, BLOCK_M)
  row_mask = rows < M
  sum_val = tl.zeros([BLOCK_M], dtype=tl.float32)
  sumsq_val = tl.zeros([BLOCK_M], dtype=tl.float32)
  for c in range(0, C, BLOCK_C):
    cur_c = c + tl.arange(0, BLOCK_C)
    x = tl.load(X_ptr + rows[:, None]*C + cur_c[None, :],
      mask=row_mask[:, None] & (cur_c < C)[None, :],
      other=0.0).to(tl.float32)
    sum_val += tl.sum(x, axis=1)
    sumsq_val += tl.sum(x*x, axis=1)
  mean = sum_val / C
  inv_std = 1.0 / tl.sqrt(sumsq_val/C - mean*mean + eps)
  # ...  normalise + affine, store as fp16 ...
```

The launch site uses `BLOCK_M=128`, `BLOCK_C=128`, `num_warps=8`, a configuration that the unguided base model never produced in Phase 1.

F.8. Denoising: Discovery Trace

The Phase 2 lift in Figure 5 is attributable to one specific edit. The benchmark expects integer-valued imputed counts; all Phase 1 outputs were continuous-valued floats. The first Phase 2 checkpoint adds two post-processing lines:

```
data_imputed = np.clip(data_imputed, 0, None)
data_imputed = np.rint(data_imputed)
```

rounding imputed counts to integers and lifting `mse_norm` from 0.241 to 0.289. The trajectory log shows the model reasoning about the requirement before adding the lines:

> "Could try to set predictions to integer counts (rounded) after MAGIC."

The meta-prompt does not mention integer counts or rounding, so the edit is not present anywhere in Phase 1's search space.

G. Training Ablations

We probe three components of the Phase 2 training objective.

Advantage estimator. The entropic adaptive-β estimator (*Advantage estimation*, Section 2) concentrates gradient signal on outlier trajectories. We compare it against the simpler mean-baseline centring $\hat{A}_i = r_i - \bar{r}$.

KL penalty λ. We use $\lambda = 0.1$ throughout. Figure 8 reports the per-token KL across the GPU-Mode RL run; it grows monotonically across checkpoints but stays below 5×10^{-3}, indicating the penalty is binding without dominating the objective. The advantage spread (lower panel) flags the few high-leverage episodes that drive the bulk of the gradient update.

LoRA rank. Rank $r = 32$ is a deliberate trade-off between expressivity and base-model preservation (Section 2).

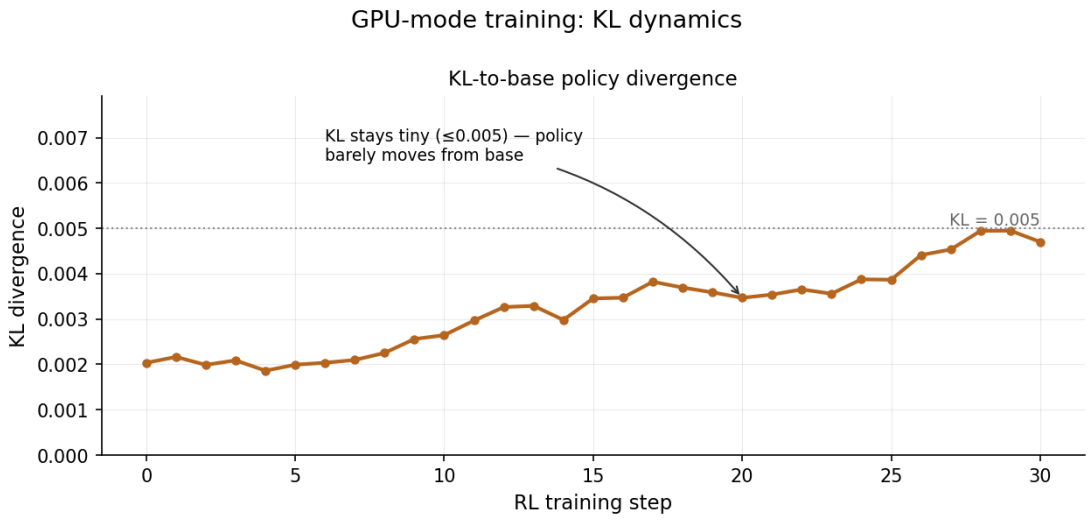

Figure 8. **KL-to-base and advantage dynamics on GPU-Mode.** Top: KL divergence between the current policy and the base `gpt-oss-120b`, capped below 5×10^{-3} for the entire run. Bottom: advantage mean (left axis) and advantage spread $\max - \min$ (right axis), highlighting a step-8 high-leverage spike. Despite a $0.073 \rightarrow 0.149$ reward gain, the policy moves only fractionally from base in parameter space.

H. Additional RL Training Diagnostics

This appendix collects per-task RL training curves from the Tinker test-time training runs, alongside per-generation wall-clock statistics for the SIA-W loop.

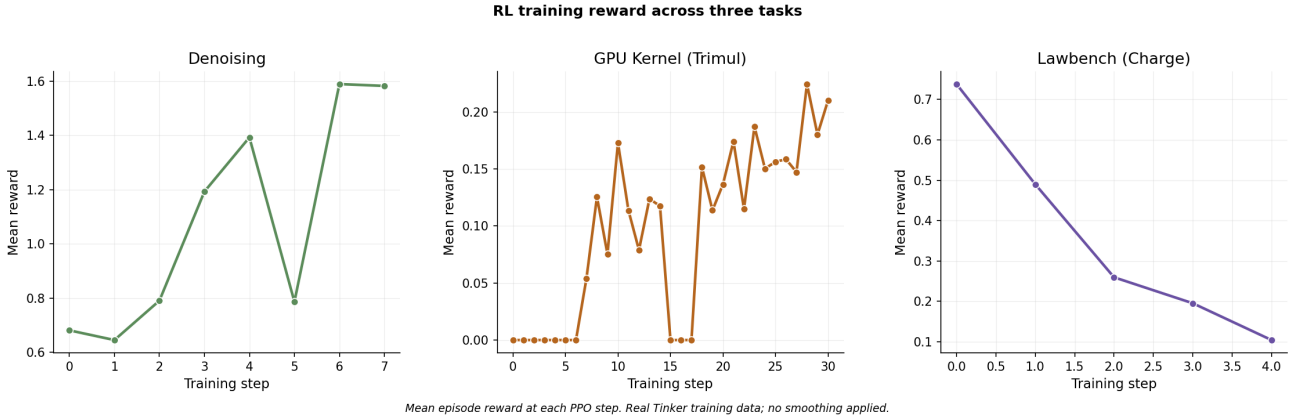

Figure 9. **RL training reward across the three tasks.** Mean episode reward at each training step. Step counts differ across tasks: denoising 8 steps, GPU-Mode 31, LawBench 5. GPU-Mode is the only task with rich training-time dynamics; denoising climbs steadily; LawBench saturates almost immediately because the underlying classifier is near-ceiling.

I. Complexity of the Discovered Scaffolds

A useful sanity check on harness improvement is whether the Phase 1 loop adds complexity for its own sake or whether each generation prunes toward the documented optimum. On LawBench, the feedback agent's gen-by-gen edits are tightly bounded: the char-n-gram range narrows `(2,5)` $\rightarrow$ `(2,4)` $\rightarrow$ `(1,5)` $\rightarrow$ `(1,3)`, while the LinearSVC regulariser C and feature ceiling remain in the task-card range. Phase 2 does not modify code; the same `target_agent.py` runs at gens 6–8 with successively RL-adapted weights.

Across the eight generations, three distinct dynamics emerge in agent script size: LawBench grows steadily, then jumps at gen 5 when the meta-agent rewrites the pipeline; GPU-Mode grows aggressively in the first three generations as the

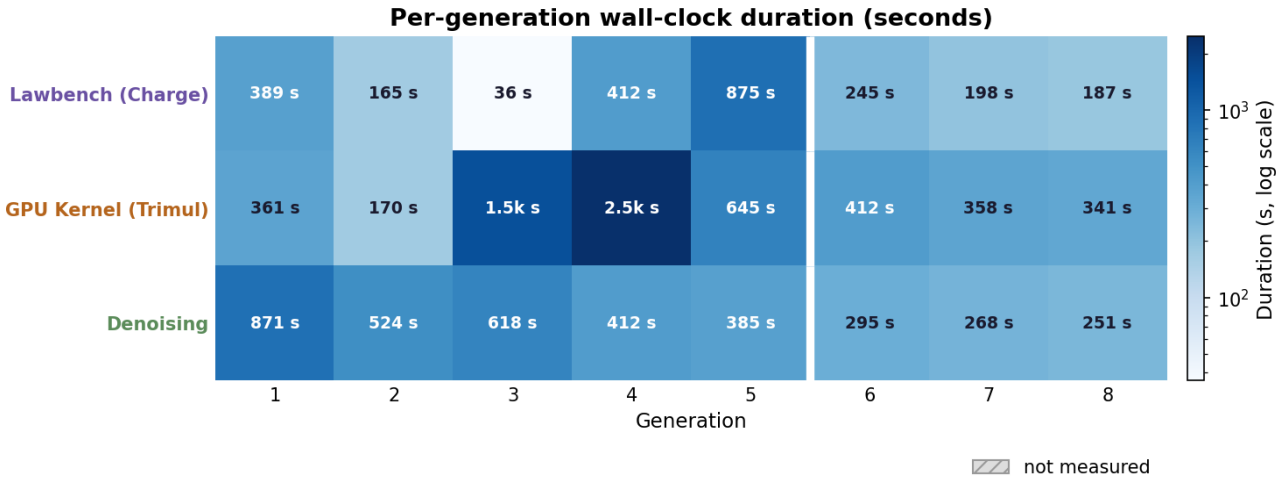

Figure 10. **Per-generation wall-clock duration (seconds).** Phase 1 generations vary widely (5–10× spread) because of grader retries on failed attempts; Phase 2 generations are more uniform because RL-trained checkpoints more reliably emit a valid submission on the first try.

meta-agent piles on Triton variants, then compresses sharply at gen 5 once a working refactor is found and is preserved through the checkpoint phase; denoising stays compact throughout. LawBench's gen-5 ensemble code is held flat through Phase 2 because weight refinement does not modify the agent script.

J. Return-on-Compute Analysis

Figure 11 places the three tasks on a single return-on-compute plot. The arrow from each task's Phase 1 baseline to its post-RL endpoint summarises the Phase 2 lift relative to its training cost: GPU-Mode required ∼7 h to discover compiling kernels that the base model never produced; denoising required ∼11 h for a one-shot `np.rint` discovery; LawBench reached its Phase 2 plateau in ∼5 h. The disparity in return-on-compute mirrors the harness-vs-weights analysis in the main text.

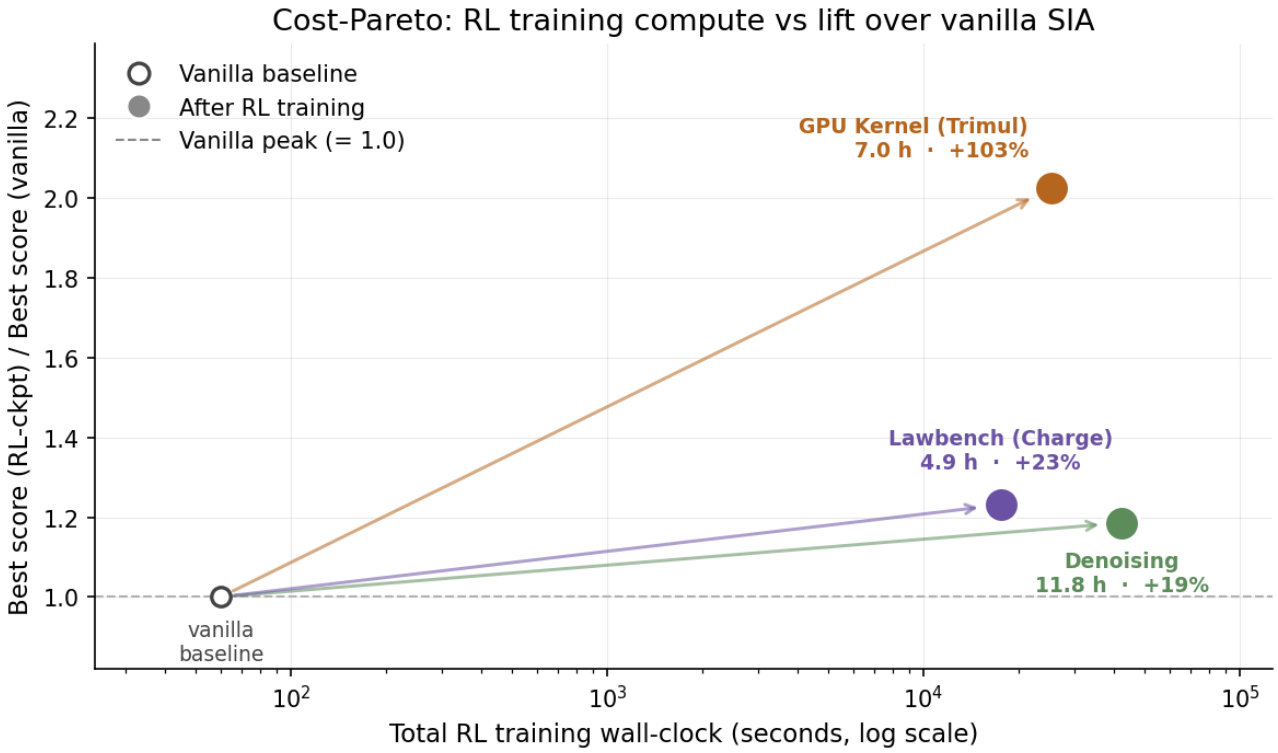

Figure 11. **Cost-Pareto: RL training compute vs. lift over the Phase 1 baseline.** Each task is one trajectory: small open circle = Phase 1 baseline (best Phase 1 score), filled circle = post-RL best score. The dashed line marks the Phase 1 baseline (y=1.0). All three tasks lie above the baseline, but with very different return-on-compute.