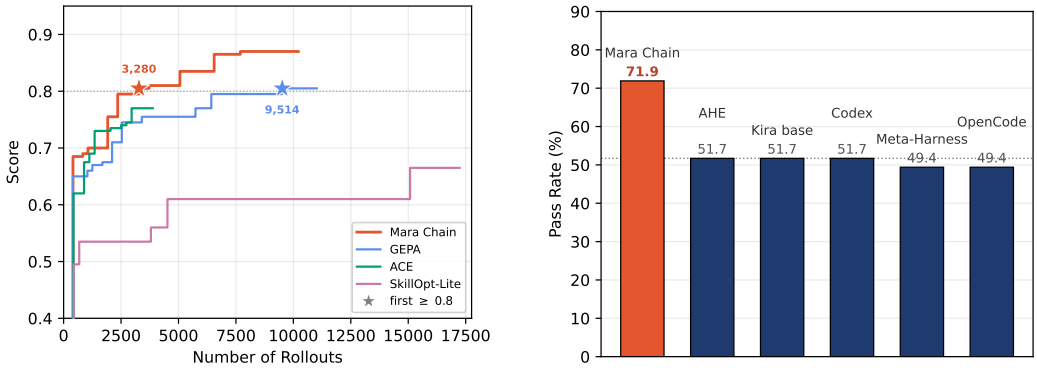
MARA CHAIN: RETHINKING FAILURE AS A STEPPING STONE FOR AI SYSTEM AUTO-EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Optimizing deployed AI systems increasingly amounts to editing prompts, skills, harnesses, and code rather than model weights. Existing approaches commonly optimize these artifacts through propose–evaluate–select procedures, where candidate configurations are evaluated and only those meeting an acceptance criterion are selected. Yet our analysis shows that discarded candidates often contain information critical for subsequent optimization. Discarding them causes later proposals to revisit the same failure modes. We introduce *Mara Chain*, a refinement procedure that turns rejected candidates into stepping stones. Rather than discarding a rejected candidate, Mara Chain retains and iteratively refines it using evidence accumulated across preceding attempts. The procedure limits each refinement chain to a fixed depth and applies Pareto-filtered Top-N selection to bound the candidate pool. Across AppWorld skill optimization, TerminalBench 2.1 harness optimization, and MuSiQue retrieval-pipeline optimization, Mara Chain delivers greater task-performance gains with fewer rollouts. It outperforms GEPA, ACE, and SkillOpt-Lite by up to $20.5\,\%$ in relative performance on AppWorld, reaching the target score with $65.5\,\%$ fewer rollouts than GEPA. It improves the pass rate by 20.2 and 22.5 percentage points over AHE and Meta-Harness on Terminal-Bench 2.1, respectively, and improves MuSiQue test nDCG@10 and Recall@10 by 0.104 and 0.131 over a hand-written retrieval pipeline.



(a) AppWorld, GLM-5 (b) TerminalBench-2.1, GLM-5

Figure 1: Comparison on AppWorld and TerminalBench-2.1. On AppWorld, Mara Chain reaches the target validation score using $65.5\,\%$ fewer rollouts than GEPA and achieves a higher final score, while ACE and SkillOpt-Lite do not reach the target score. On TerminalBench-2.1, Mara Chain achieves the highest pass rate among the compared methods.

1 INTRODUCTION

AI systems are increasingly optimized through mutable prompts, skills, harnesses, scripts, configurations, and code rather than model-weight updates. A standard approach is to place these artifacts in a propose–evaluate–select loop, where an LLM proposes a candidate configuration and an evaluator decides whether to select it. Such procedures have been applied to prompts (Yang et al., 2024; Zhou et al., 2022; Fernando et al., 2023; Guo et al., 2024; Pryzant et al., 2023), compound systems

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.

(Khattab et al., 2023; Cheng et al., 2024; Wu et al., 2025; Zhang et al., 2025b), code (Novikov et al., 2025), and agent skills and harnesses (Yang et al., 2026; Shen et al., 2026; Lee et al., 2026; Lin et al., 2026). Unlike weight fine-tuning, these artifact-optimizing methods can adapt a deployed system without changing its underlying model.

However, existing methods often suffer from *persistent failure barriers*: they repeatedly re-analyze and re-attempt the same failing task, making no real progress even as the rollout budget grows. Inspecting the rollout traces, we find that the culprit is how these methods handle candidates: any candidate that does not improve the score on the held-out set is discarded outright. Yet such a discarded candidate is rarely worthless. It may carry a partial fix that can be extended, implement a sound direction whose implementation contains a repairable bug, or be a wrong attempt whose measured effect rules out one route to the goal, prompting later proposals to try a different one. By throwing these candidates away, existing methods discard the very evidence needed to move past the barrier, forcing each subsequent proposal to reconstruct the same diagnosis from scratch. We argue that these discarded candidates are an underused resource: retaining and building on them, rather than throwing them away, is key to breaking the barrier and can substantially improve the performance of artifact optimization.

We introduce the Mara Chain[1], a procedure that, within the propose–evaluate–select loop for compound AI system optimization, refines rejected candidates instead of discarding them, forming a history-conditioned chain to exploit the evidence in failed attempts. In each epoch, when a candidate does not clear the configured acceptance bar, Mara Chain retains its rollout traces, residual failures, structured analysis, and change history. It then generates a sequence of descendants: at each step, the LLM analyzes the accumulated evidence and proposes the next candidate accordingly. To keep this bounded, the chain runs for only a fixed depth; if it produces no descendant that beats the initial candidate, the chain is discarded, so a failed chain adds no lasting cost. Furthermore, to avoid an ever-growing candidate pool, in which many candidates remain mutually non-dominated and cannot be pruned, the candidate pool applies a top-N truncation to the Pareto-filtered set, keeping the N candidates with the highest scores on the validation set. As a result, Mara Chain overcomes the stalling that traps existing methods while keeping its cost bounded.

We evaluate Mara Chain on three benchmarks—AppWorld (Trivedi et al., 2024), Terminal-Bench 2.1 (Merrill et al., 2026), and MuSiQue (Trivedi et al., 2022)—each optimizing a distinct class of tunable artifacts, across three LLM models: GLM-5 (Zeng et al., 2026), DeepSeek-V4-Pro, and Qwen3.5-397B-A17B. Figure 1 presents the main comparisons on AppWorld and Terminal-Bench 2.1. On **AppWorld**, whose artifacts form a *skill* configuration, Mara Chain improves over the strongest skill optimizers—GEPA (Agrawal et al., 2025), ACE (Zhang et al., 2026c), and SkillOpt-Lite (Shen et al., 2026)—by up to 20.5 %, reaches the target validation score using 65.5 % fewer rollouts than GEPA, and achieves a higher final score; ACE and SkillOpt-Lite do not reach the target score. On **TerminalBench 2.1**, an *agent-harness* configuration, the optimized harness improves the pass rate by 39.1 % over the harness optimizers AHE (Lin et al., 2026) and Meta-Harness (Lee et al., 2026). On **MuSiQue**, a *retrieval-pipeline* configuration, Mara Chain improves test nDCG@10 / Recall@10 by 34.6 % / 39.8 % over the hand-written default pipeline, showing the approach is not specific to any single artifact type.

2 Problem Statement

Tunable artifacts. Tunable artifacts are the modifiable behavioral components of a system under optimization, including prompts, scripts, harnesses, configuration files, and source code. We represent these artifacts as a concrete directory, whether the target system is an AI agent, a workflow, or a standalone algorithm. Rather than optimizing a constrained parameter vector or a domain-specific language, the optimizer directly adds, removes, and modifies files in this directory. For each target system, the user specifies an *artifact schema* comprising the directory structure and the semantic role of each file. This schema establishes the artifact mapping and defines the set of files that the proposer may modify (App. A). Consequently, any system with tunable artifacts that can be mapped

[1]The name alludes to Māra, the embodiment of the obstacles that, in Buddhist tradition, confront the practitioner on the path to awakening. In the same spirit, the method advances through repeated setbacks: each attempt that fails to clear the acceptance bar is retained rather than discarded, its failed experience accumulated as a stepping stone, until the accumulated attempts jointly reach the goal.

in this way and evaluated reproducibly—such as a skill, evaluation harness, retrieval pipeline, or standalone computational kernel—can be optimized by the same procedure.

Optimization objective. Given an artifact configuration σ and a task distribution $\mathcal{D}$, we seek

$$\sigma^\star \;=\; \arg\max_{\sigma}\; \mathbb{E}_{\tau\sim\mathcal{D}}\big[\,\text{score}\big(\text{exec}(\sigma,\tau)\big)\,\big] \quad \text{s.t.} \quad |\{\text{rollouts}\}| \le B, \tag{1}$$

where $\text{exec}(\sigma,\tau)$ executes the system instantiated with artifact configuration σ on task τ, and score $\in [0,1]$ is the per-instance performance measure. Because each $\text{exec}(\sigma,\tau)$ is a costly rollout, this is an optimization problem under a fixed rollout budget B: the goal is to attain as high an expected score as possible while issuing a limited number of rollouts.

Persistent failure barriers. A task τ imposes a set of conditions. Executing σ on τ yields a *residual* $\delta(\sigma,\tau)$: the unsatisfied conditions exposed by the execution. Across successive candidate configurations, this residual can stay invariant—unchanged no matter how much rollout budget is spent—rather than being a merely hard condition that additional candidates or rollouts would eventually resolve. We call this phenomenon a *persistent failure barrier*: a condition whose residual remains invariant across iterations and prevents τ from succeeding. Such barriers commonly arise under the standard propose–evaluate–select search even when a correct modification is reachable, especially on long-horizon tasks. As a result, the correct modification is repeatedly discarded and the same residual is re-diagnosed across iterations. Many factors drive this, such as a partial or well-directed change that yields no immediate score gain despite being on the right track, a correct modification that is never exercised at execution, or a target condition gated by unmet prerequisites. Regardless of the cause, additional rollouts leave the residual unchanged.

3 MARA CHAIN

We introduce Mara Chain, an improved propose–evaluate–select procedure that searches for a high-performing, task-specific artifact configuration. Mara Chain is built on a key insight: a rejected candidate is not a dead end but evidence for the next attempt. As shown in Figure 2, Mara Chain is embedded in an outer population-based loop that validates improvements and, through Pareto-filtered Top-N selection, maintains a bounded candidate pool. All rollout logs, scores, traces, and metadata are persisted in a filesystem-backed memory, which the LLM-based proposer can access to retrieve and analyze the complete history of prior attempts.

Outer Optimization Loop. Figure 2 overviews the outer propose–evaluate–select loop. The loop maintains a shared candidate pool $\mathcal{P}$ of artifact configurations, invokes Mara Chain refinement procedure only when a proposal fails the acceptance criterion, and runs multiple optimization slots in parallel until the rollout budget is exhausted. Let $B \subset \mathcal{D}_{\text{tr}}$ denote a training minibatch and V the validation dataset. Each candidate $P \in \mathcal{P}$ is an artifact configuration as defined in §2.

To select a parent for an available slot, Mara Chain favors candidates that lead on different score dimensions. For a candidate P, let $z_j(P)$ denote its score on dimension j, and define its leadership count as

$$\ell(P) = \sum_j \mathbb{I}\left[z_j(P) \ge \max_{Q\in\mathcal{P}} z_j(Q) - \varepsilon\right]. \tag{2}$$

For multiple slots, parent candidates are sampled without replacement with probabilities proportional to their leadership counts $\ell(P)$. This rule favors candidates that are competitive on complementary dimensions while retaining stochastic exploration.

The slot evaluates P on B to establish a minibatch baseline, then proposes a candidate P_n. A candidate clears the acceptance bar when its minibatch score improves sufficiently over that of its parent P under the threshold δ. If P_n clears the acceptance bar against P, it is evaluated on all validation tasks in V. Otherwise, the slot invokes Mara Chain refinement procedure. A candidate returned by Mara Chain Reflective Refinement is forwarded to validation only if it clears the same acceptance bar against P; otherwise, it is discarded. After validation, the outer loop updates $\mathcal{P}$ using Pareto-Filtered Top-N Selection. The loop stops when its rollout budget is exhausted.

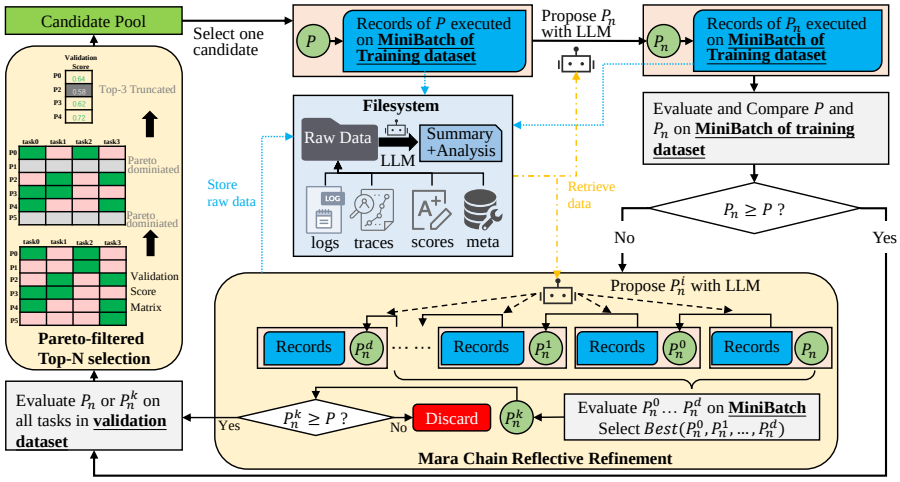


Figure 2: Overview of one Mara Chain optimization slot. A parent P is sampled from the candidate pool using multi-dimensional leadership weights and executed on a training minibatch. Rollout logs, traces, scores, and metadata are persisted to a filesystem-backed memory, where an analysis phase produces a structured summary. The mutation phase uses the parent and that summary to generate P_n. If P_n clears the configured acceptance bar against P on the same minibatch, it is evaluated on the validation set. Otherwise, the Mara chain successively generates $P_n^0, \ldots, P_n^d$ from the retained execution history, selects the best-scoring chain candidate on the minibatch, and validates it only when it clears that bar against P. Validated candidates undergo Pareto-filtered Top-N selection: dominated candidates are removed using validation score vectors, and a validation-score top-N truncation bounds the remaining pool. The process repeats until the rollout budget is exhausted. Multiple slots execute this procedure in parallel over the shared candidate pool.

Algorithm 1 Reflective refinement in Mara Chain.

Require: parent P with baseline rollout R_P, rejected direct candidate P_n with rollout R_n, minibatch B, acceptance bar δ, depth d

$\mathcal{C} \leftarrow [\langle P, R_P\rangle, \langle P_n, R_n\rangle]$ {Mara-chain nodes ⟨candidate, rollout⟩; root first}
for $i = 0, \ldots, d$ **do**
  $P_n^i \leftarrow$ `store.create_child`($\mathcal{C}$.last) {inherits $\mathcal{C}$.last's artifacts}
  $\Gamma_i \leftarrow$ `store.build_context`($\mathcal{C}$) {score history, prior analyses, attributed changelog, run records}
  $A_i \leftarrow$ `proposer.mara_analyse`($\mathcal{C}$.last, Γ_i) {analyses persisted to `store`}
  `proposer.mutate`(P_n^i, A_i) {edits P_n^i's artifacts in place}
  $R_i \leftarrow$ `rollout`(P_n^i, B); $s_i \leftarrow$ `score`(R_i)
  `store.persist`(P_n^i, R_i, s_i); $\mathcal{C}$.`add`($\langle P_n^i, R_i\rangle$)
  if *clears*($s_i, s_B(P), \delta$) **then break** {stop once the bar is met}
end for
($best$, s_{best}) $\leftarrow \arg\max_{\langle C,R\rangle \in \mathcal{C}[1:]}$ `score`(R)
if *clears*($s_{best}, s_B(P), \delta$) **then**
  return *best* {caller evaluates it on V}
end if
return NONE

Mara Chain Reflective Refinement. The Mara Chain is a sequential, history-conditioned refinement procedure. It is triggered only after the direct candidate P_n does not clear the acceptance bar against P on the current minibatch. Rather than treating that rollout as discarded information, the chain preserves the candidate, its execution records, the structured analysis, and the artifact-change history. At chain step i, the proposer reads this accumulated context and produces P_n^i; the rollout of P_n^i then extends the context for step $i+1$. Thus, all chain candidates form one lineage, and every proposal is conditioned on the observed consequences of its predecessors.

Algorithm 1 specifies the procedure. Each chain step assembles its context Γ_i from the store: the per-instance score history across all chain nodes, the structured analyses written for prior nodes, the attributed artifact-change history along the chain, and the raw run records. All chain nodes are scored on the same fixed minibatch, and the chain returns its highest-scoring node only when it clears the same configured acceptance bar as the direct candidate, stopping as soon as one does. The store is not a passive log: it retains raw rollout data and its LLM-generated analyses, which make the accumulated evidence available to subsequent proposals. The full store schema and analysis prompts are specified in App. A. This permits a later candidate to extend a partial fix, repair the implementation bug of an attempt whose direction was sound, or pursue the goal by a different route once an earlier attempt has ruled one out. The chain depth d bounds the additional rollout cost of a failed direct proposal.

Pareto-Filtered Top-N Selection. Validation produces a per-task score vector $\mathbf{z}_V(P) \in [0,1]^{|V|}$ for each candidate. The pool update uses two stages. First, a candidate P is removed if another candidate Q weakly Pareto dominates it:

$$Q \succeq P \quad \Longleftrightarrow \quad z_{V,j}(Q) + \varepsilon \geq z_{V,j}(P) \quad \text{for all } j. \tag{3}$$

For identical validation vectors, the more recent candidate is retained. This per-instance criterion preserves candidates that excel on complementary subsets of validation tasks, rather than prematurely reducing them to a single scalar score.

Second, if the non-dominated set contains more than N candidates, we retain the N candidates with the highest mean validation scores $\frac{1}{|V|}\sum_j z_{V,j}(P)$ and remove the rest. The first stage preserves complementary validation performance; the second stage bounds memory and evaluation cost because high-dimensional per-task score vectors leave many candidates mutually non-dominated, causing the candidate pool to grow without bound under Pareto filtering alone. Its implementation pseudocode and scheduler semantics are provided in App. A.1.

Practical Implementation. We implement Mara Chain in a modular optimization framework whose pluggable target-side, optimizer-side, and candidate-store components are instantiated per setting with the target system and artifact schema. Component interfaces and concrete artifact layouts are detailed in App. A.

4 Evaluation

We evaluate Mara Chain on three benchmarks spanning distinct tunable artifact configurations: AppWorld (Trivedi et al., 2024), which optimizes skills consisting of `SKILL.md` guidance, references, and scripts; TerminalBench 2.1 (Merrill et al., 2026), which optimizes an agent-harness configuration over 89 agentic-OS tasks; and MuSiQue (Trivedi et al., 2022), which optimizes a retrieval-pipeline configuration. We conduct experiments across three frozen target LLMs: GLM-5 (Zeng et al., 2026), DeepSeek-V4-Pro-0813, and Qwen3.5-397B-A17B; unless otherwise stated, GLM-5 is the default model. We use the Pi coding agent (Earendil Inc., 2025) as the LLM-based proposer. We compare against GEPA (Agrawal et al., 2025), ACE (Zhang et al., 2026c), and SkillOpt-Lite (Shen et al., 2026) for skill optimization, and AHE (Lin et al., 2026) and Meta-Harness (Lee et al., 2026) for agent-harness optimization; Codex (OpenAI, 2025) and OpenCode (Anomaly, 2025) provide additional agent-harness references. For MuSiQue, we compare against a hand-written default pipeline because existing optimizers do not provide directly comparable retrieval-pipeline implementations. App. A specifies the implementation interfaces and artifact schemas, while App. E documents the metrics, splits, hyperparameters, and evaluation budgets. Unless otherwise stated, we set the Pareto-filtered Top-N parameter to $N = 3$ and the maximum number of Mara Chain reflective refinement iterations to $d = 5$.

4.1 Main Results: Performance and Efficiency

AppWorld: Skills. The AppWorld tunable artifacts form a *skill configuration*: natural-language `SKILL.md` guidance, references, and runnable scripts. We evaluate on 585 AppWorld tasks: 168 `test_normal` and 417 `test_challenge`; the training-pool and validation-set construction is in App. E.

Table 1: AppWorld evaluation of optimized skill configurations on 585 recorded tasks (168 `test_normal`; 417 `test_challenge`). TGC = task-grade completion, SGC = scenario-grade completion; Normal and Challenge are the two test splits. The green number is the improvement over the empty-skill configuration (pp). Parentheses identify the slot configuration attaining each Mara Chain result. Bold marks the best value per column.

Optimized skill configuration	Normal TGC	Normal SGC	Challenge TGC	Challenge SGC
Empty skill	48.2	35.7	33.3	19.4
SkillOpt-Lite (Shen et al., 2026)	70.8 +22.6	51.8 +16.1	68.3 +35.0	46.0 +26.6
GEPA (Agrawal et al., 2025)	83.3 +35.1	67.9 +32.2	69.3 +36.0	44.6 +25.2
ACE (Zhang et al., 2026c)	83.9 +35.7	69.6 +33.9	74.1 +40.8	55.4 +36.0
Mara Chain	**89.9** +41.7 (2)	**83.9** +48.2 (2)	**76.7** +43.4 (1)	**59.0** +39.6 (1)

Table 1 reports final performance across the Normal and Challenge splits. Mara Chain attains the best result in every metric: 89.9/83.9 Normal TGC/SGC and 76.7/59.0 Challenge TGC/SGC. It exceeds the specialized skill optimizers GEPA, ACE, and SkillOpt-Lite across both task types, showing that retained refinement improves final performance beyond prompt-only and skill-only optimization baselines. Figure 1(a) shows that Mara Chain achieves both higher rollout efficiency and stronger final performance than the competing skill optimizers. It reaches a validation score of 0.8 after 3,280 rollouts, compared with 9,514 for GEPA—about one third as many rollouts as the strongest baseline; ACE and SkillOpt-Lite never reach this score. The wall-clock view of the same trajectories shows the same gap (Figure 8, App. F): Mara Chain reaches 0.8 in 11.9 h, about 3× faster than GEPA (35.8 h). Mara Chain ultimately reaches 0.87, while GEPA plateaus at 0.805. As shown in Figure 1(a), all methods improve rapidly during the early rollouts, when many readily diagnosed conditions remain. As these conditions are resolved, the remaining long-horizon problems require dependent modifications and expose persistent failure barriers. GEPA, ACE, and SkillOpt-Lite plateau in this regime, whereas Mara Chain retains and refines evidence from rejected candidates to continue improving. Mara Chain eventually plateaus because of the remaining model-capability limits and its bounded depth d.

TerminalBench 2.1: Harnesses. The TerminalBench 2.1 tunable artifacts form an *harness configuration*. The harness optimizers AHE and Meta-Harness are the baselines, with off-the-shelf CLI agents Codex and OpenCode and the un-optimized Kira base as references. The system model is GLM-5 across rows; the reported comparison remains end-to-end under the stated configurations.

Figure 1(b) shows that Mara Chain substantially outperforms both specialized harness optimizers and off-the-shelf coding agents on the 89-task TerminalBench 2.1 evaluation. The Mara-Chain-optimized harness attains a 71.9% pass rate, exceeding the next-best results of 51.7% from AHE, Codex, and the Kira base by 20.2 percentage points, and surpassing Meta-Harness and OpenCode (49.4%) by 22.5 points. The per-task binary rewards underlying this comparison are provided in App. G.

MuSiQue: Retrieval Pipelines (Beyond Agentic Systems). The MuSiQue tunable artifacts form a *retrieval-pipeline configuration*: a DAG of retrieval nodes with tunable node configurations, node scripts, and a prompt file (App. H). The baseline is a hand-written default pipeline, against which we report the end-to-end optimized pipeline. On MuSiQue (Table 2), the optimized pipeline improves test nDCG@10 by +0.104 and Recall@10 by +0.131 over the default, with validation gains of +0.134/+0.210.

These results show that Mara Chain is not limited to self-optimizing agentic systems: whenever a system's tunable behavior can be mapped to editable artifacts and evaluated on a target task distribution (§2), the same procedure applies. MuSiQue instantiates this broader setting as a retrieval DAG whose configurations, node scripts, and prompts are optimized together.

4.2 Ablation and Case Study

Effects on the final score. Figure 3(*left*) reports the final scores of the full setting (Mara Chain + Top-N) versus the two ablations that disable the Mara Chain flow or the Top-N selection. Mara

Table 2: MuSiQue retrieval (4-hop). The optimized artifact is a retrieval-pipeline configuration; the baseline is a hand-written default configuration. Metrics over top-10 retrieved documents. Test is disjoint from the val split used for development. w/o Mara Chain removes retained refinement (Mara Chain depth 0) at the same 2-slot budget. The green number is the gain over the default baseline; Bold = best per column.

Artifact configuration	nDCG@10 (test)	Recall@10 (test)	nDCG@10 (val)	Recall@10 (val)
Default (hand-written)	0.301	0.329	0.419	0.395
w/o Mara Chain (2-slot)	0.371 +0.070	0.421 +0.092	0.505 +0.087	0.560 +0.165
Mara Chain (2-slot)	**0.405** +0.104	**0.460** +0.131	**0.553** +0.134	**0.605** +0.210

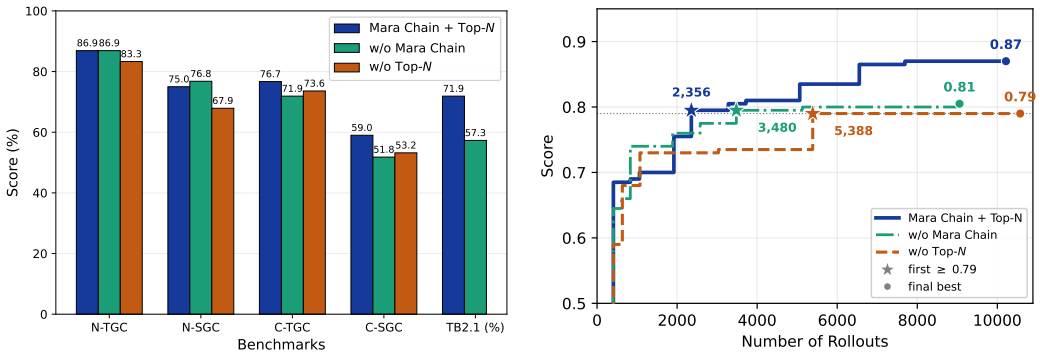


Figure 3: **Left:** Ablation study of Mara Chain Reflective Refinement and Top-N selection. Mara Chain + Top-N enables both mechanisms; each ablation disables one of them—the Mara Chain flow or the Top-N selection—and is compared against the both-on setting. All settings use the 1-slot configuration. Top-N selection uses $N = 3$ on AppWorld and $N = 1$ on TerminalBench 2.1 (no w/o Top-N result for TerminalBench 2.1). **Right:** AppWorld validation score versus optimization rollouts for the Mara Chain and Top-N ablation. Mara Chain with Top-N reaches 0.79 after 2,356 rollouts and a final score of 0.87. w/o Mara Chain reaches 0.79 after 3,480 rollouts and finishes at 0.81, whereas w/o Top-N reaches 0.79 after 5,388 rollouts. Combining Mara Chain with Top-N is both the most sample-efficient and the highest-scoring setting.

Chain's benefit grows with task difficulty. On the easier Normal split its effect is barely visible: Normal TGC is unchanged (86.9 vs 86.9 w/o Mara Chain) and Normal SGC moves by less than two points (75.0 vs 76.8). On the harder Challenge split the effect turns clearly positive: Challenge TGC 76.7 vs 71.9 (+4.8 pp) and Challenge SGC 59.0 vs 51.8 (+7.2 pp). The gain is largest on the hardest benchmark, TerminalBench 2.1, where the pass rate rises from 57.3% (w/o Mara Chain) to 71.9% (+14.6 pp). Mara Chain's gain extends beyond agentic benchmarks: on MuSiQue, disabling Mara Chain at the same 2-slot budget lowers test nDCG@10 by 0.034 and Recall@10 by 0.039 (Table 2). Disabling Top-N selection instead degrades every AppWorld metric (Normal TGC/SGC 83.3/67.9, Challenge TGC/SGC 73.6/53.2), so both mechanisms contribute to the full setting's scores.

Effects on sample efficiency. Figure 3(*right*) compares the optimization sample efficiency of the three settings. The full setting reaches a validation score of 0.79 after only 2,356 rollouts and continues to a final score of 0.87. The w/o Mara Chain ablation (Top-N only) reaches 0.79 after 3,480 rollouts and finishes at 0.81, while the w/o Top-N ablation (Mara Chain only) needs 5,388 rollouts to reach 0.79. Mara Chain and Top-N are therefore synergistic: together they are both more sample-efficient and higher-scoring than either mechanism alone, and no single-mechanism setting attains the full setting's final score at any comparable budget.

Case study. Figure 4 records the `sanitize-git-repo` trace on TerminalBench 2.1. It shows when Mara Chain Reflective Refinement matters: the task needs several corrections, yet no single proposal clears its parent's acceptance bar. In the displayed w/o Mara Chain setting, a candidate that contains a partial fix but fails to clear the bar is discarded; the next propose therefore starts from a different proposal instead of extending the partial fix, and repeated iterations leave the task failing. Along the recorded Mara Chain lineage, by contrast, candidates that miss the direct bar are

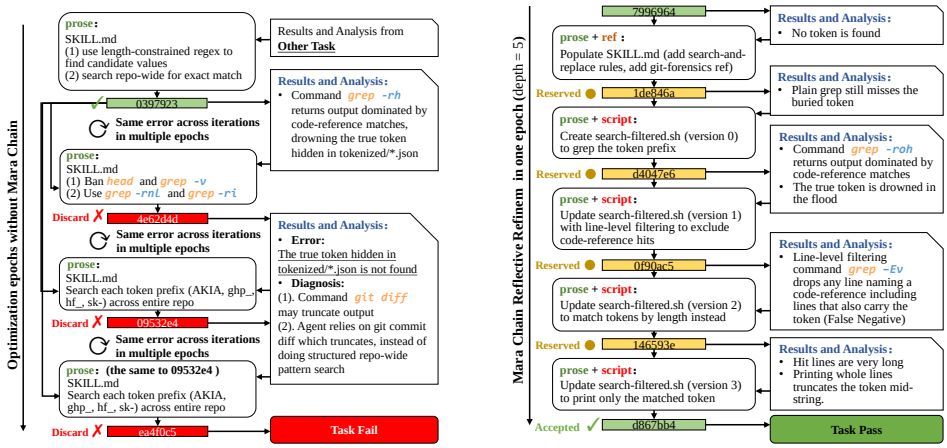


Figure 4: Illustrative recorded case study of `sanitize-git-repo`. **Left (without Mara Chain):** proposals introduce partial fixes, but each candidate that fails to clear the acceptance bar is discarded; later iterations therefore do not continue from these partial fixes, and the task remains failed. **Right (with Mara Chain):** a direct candidate that does not clear the bar is retained with its execution evidence and refined along one lineage. Successive descendants build on partial fixes, move from prose guidance to a script, correct the script's observed behavior, and finally pass the task. Green, yellow, and red bars indicate accepted, reserved, and discarded candidates.

Table 3: Paired AppWorld evaluation of Mara Chain optimized skills versus their empty-skill configurations across three LLMs. Δ is the mean improvement across the four metrics.

LLM Model	Method	N-TGC	N-SGC	C-TGC	C-SGC	Δ
GLM-5	Empty skill	48.2	35.7	33.3	19.4	—
	Mara Chain	86.9	75.0	76.7	59.0	+40.3
DeepSeek-V4-Pro	Empty skill	66.7	39.3	60.9	30.9	—
	Mara Chain	93.5	83.9	81.5	69.8	+32.7
Qwen3.5-397B-A17B	Empty skill	63.7	48.2	50.1	30.2	—
	Mara Chain	82.7	73.2	73.6	56.1	+23.4

retained with their rollout evidence, residuals, and change history. Successive steps build on the partial fixes, first moving from prose guidance to a script and then correcting the script's observed behavior, until a candidate passes the task. Notably, each refinement step along the Mara Chain lineage tends to escalate from low-reliability prose guidance toward a higher-reliability script, so that the final candidate's behavior is enforced by executable code rather than by advisory text alone; in the w/o Mara Chain setting, by contrast, successive proposals typically remain prose-level edits, whose effect is harder to enforce reliably. The trace thus highlights Mara Chain's advantage on long-horizon, hard problems: instead of discarding failed candidates, it accumulates their partial fixes along a lineage until the task passes, yielding solutions that are both higher-performing and more reliable. The complete recorded case material is provided in App. J.

4.3 Cross-Model Generalization

We evaluate Mara Chain optimized skills and their corresponding empty-skill configurations on AppWorld across three system models: GLM-5, DeepSeek-V4-Pro-0813, and Qwen3.5-397B-A17B. Table 3 shows paired evaluation differences across all three models. On average over the four metrics, Mara Chain improves the optimized skill over its empty-skill baseline by $+40.3$ pp on GLM-5, $+32.7$ pp on DeepSeek-V4-Pro, and $+23.4$ pp on Qwen3.5-397B-A17B. Across all three models, Mara Chain improves every metric over the corresponding empty-skill baseline, demonstrating that the optimization is not tied to any single model and generalizes across different LLM models.

5 Related Work

Prompt evolution. Over a *prompt string*, AutoPrompt (Shin et al., 2020) searches discrete tokens via input gradients; OPRO (Yang et al., 2024) casts the LLM as optimizer over a meta-prompt, APE (Zhou et al., 2022) generates and selects instructions, PromptBreeder (Fernando et al., 2023) and EvoPrompt (Guo et al., 2024) apply evolutionary mutate–select, PromptWizard (Agarwal et al., 2024) self-evolves via critique and synthesis, and ProTeGi (Pryzant et al., 2023) uses a textual gradient with beam search. GEPA (Agrawal et al., 2025) applies Genetic-Pareto evolution with reflection to the system's prompt strings. Compound-system optimizers like TextGrad (Yuksekgonul et al., 2025), Trace (Cheng et al., 2024), Optimas (Wu et al., 2025), Symbolic Learning (Zhou et al., 2024), DSPy (Khattab et al., 2023), MIPRO (Opsahl-Ong et al., 2024), and AFlow (Zhang et al., 2025b) optimize pipelines or workflows, and AlphaEvolve/FunSearch (Novikov et al., 2025; Romera-Paredes et al., 2024) and ADAS (Hu et al., 2024) evolve code; MAP-Elites (Mouret & Clune, 2015) is the quality-diversity family these draw on.

Harness evolution. AHE (Lin et al., 2026) and Meta-Harness (Lee et al., 2026) search harness code with an agentic proposer over filesystem-stored traces—AHE wrapping analyze–mutate in a falsify-and-rollback loop, Meta-Harness leaving *how* to evolve to the agent itself. Self-Harness (Zhang et al., 2026a), AutoHarness (Lou et al., 2026), and DarwinX (Zhang et al., 2026d) are related variants. ACE (Zhang et al., 2026c) evolves contexts as playbooks via generate–reflect–curate (building on dynamic cheatsheet (Suzgun et al., 2026)), and SkillOpt / SkillOpt-Lite / SkillCAT (Yang et al., 2026; Shen et al., 2026; Chen et al., 2026) optimize agent skills.

Reflection and self-refinement. Reflecting on failures to drive the next attempt underlies Reflexion (Shinn et al., 2023) and Self-Refine (Madaan et al., 2023). Experiential agents accumulate reusable skills or workflows—Voyager (Wang et al., 2023), ExpeL (Zhao et al., 2024), Agent Workflow Memory (Wang et al., 2024)—and self-referential evolution targets the agent itself (Zhang et al., 2025a; 2026b).

6 Limitation and Future Work

Limitations. Mara Chain diagnoses one batch at a time and can overfit that batch's failure modes, which per-instance Pareto selection across batches only mitigates. Its context construction and diagnosis logic are also a hand-designed first version that remains fixed throughout the search.

Future work. One line tightens efficiency: chain depth adapts to problem difficulty—deeper on hard barriers, earlier termination on easy ones—with cheaper proposer models and budgeted depth reducing wall-clock and token cost. A second widens exploration: diversity-aware selection beyond a single test score, and branching the serial lineage into a parallel *Mara-tree*. A third makes the chain trustworthy and transferable: per-node confidence estimates with fallback to the baseline, and effective strategies distilled into a shared experience bank meta-learned across artifact configurations and domains.

7 Conclusion

We introduced the *Mara Chain*, a history-conditioned refinement procedure for the propose–evaluate–select optimization of tunable artifacts: instead of discarding a rejected candidate, it retains the candidate's rollout traces, residual failures, structured analysis, and artifact-change history and refines them on the same training minibatch along a single lineage, converting evidence from failed attempts into context for later proposals. Across skills, agent harnesses, and retrieval pipelines (AppWorld, TerminalBench 2.1, MuSiQue), Mara Chain outperforms specialized optimizers on both final performance and sample efficiency, and the gains persist across GLM-5, DeepSeek-V4-Pro, and Qwen3.5-397B-A17B. Ablations credit both mechanisms: retained refinement drives the largest gains on the hardest tasks, and the outer loop's Pareto-filtered Top-N selection further improves final scores and sample efficiency. These results indicate that retained, evidence-conditioned refinement is a general, model-agnostic mechanism for AI-system auto-optimization. Code will be released upon acceptance.

AI USE STATEMENT

In this work, we used generative AI tools—the open-source OpenCode CLI and an internal coding-assistant CLI, running the Kimi-K3 and GLM-5.2 models—to implement methods: the optimization framework behind our experiments was co-developed with AI assistance through repeated human-directed iterations, each experimental setup was built with AI assistance, and the frontend was built with AI-generated code. We also used these tools to organize and interpret experimental results, including reading run logs, analyzing failure trajectories, and compiling experimental data into the result tables, and to provide feedback on the research methodology: a first draft of the methodology was written by AI from phenomena the authors had observed in repeated experiments through our run-visualization frontend, and AI was further used to inspect problem trajectories and critique the methodology prompts. We have not used generative AI tools to generate synthetic data sets or to formulate the mathematical claims or their proofs.

Additionally, we used generative AI tools to search for and organize related literature and formatting requirements, to suggest the structure of this paper, to draft and edit parts of the manuscript, and to create the figures and tables.

The research idea and the overall architecture of the framework were conceived by the authors; AI assistance filled in implementations and executed code modifications under human direction, and all refactoring was initiated and described by the authors. We have reviewed all AI-assisted work: AI-drafted manuscript text was reviewed, revised, or rewritten by the authors, and AI-written code was run, observed, and debugged through repeated iterations before use. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Eshaan Agarwal, Joykirat Singh, Vivek Dani, Raghav Magazine, et al. PromptWizard: Task-aware prompt optimization framework. *arXiv preprint arXiv:2405.18369*, 2024.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, et al. GEPA: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025. ICLR 2026 (Oral). Code: `https://github.com/gepa-ai/gepa`.

Anomaly. OpenCode: an open-source ai coding agent. `https://github.com/anomalyco/opencode`, 2025.

Beijing Academy of Artificial Intelligence. BAAI/bge-reranker-v2-m3. `https://huggingface.co/BAAI/bge-reranker-v2-m3`, 2024.

Kunfeng Chen, Qihuang Zhong, Juhua Liu, and Bo Du. Skillcat: Contrastive assessment and topology-aware skill self-evolution for llm agents. *arXiv preprint arXiv:2606.13317*, 2026.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next AutoDiff: Generative optimization with rich feedback, execution traces, and LLMs. *arXiv preprint arXiv:2406.16218*, 2024.

Gordon V Cormack, Charles LA Clarke, and Stefan Buettcher. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pp. 758–759, 2009.

Earendil Inc. Pi Coding Agent. `https://pi.dev`, 2025. Code: `https://github.com/earendil-works/pi`; npm `@earendil-works/pi-coding-agent`.

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

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. 2024:34133–34156, 2024.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

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

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

Jiahang Lin, Shichun Liu, et al. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026.

Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P. Murphy. AutoHarness: Improving LLM agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329*, 2026.

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

Mike A. Merrill et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

Alexander Novikov et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. Codex CLI: an open-source agentic coding tool for the terminal. `https://github.com/openai/codex`, 2025.

Krista Opsahl-Ong, Michael J. Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, et al. Optimizing instructions and demonstrations for multi-stage language model programs. *arXiv preprint arXiv:2406.11695*, 2024.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*, 2023. EMNLP 2023.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Yifei Shen, Bo Li, and Xinjie Zhang. SkillOpt-Lite: Better and faster agent self-evolution via one line of vibe. *arXiv preprint arXiv:2607.03451*, 2026. Code: `https://github.com/EvolvingLMMs-Lab/SkillOpt-Lite`.

Taylor Shin, Yasaman Razeghi, Robert L Logan Iv, Eric Wallace, and Sameer Singh. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. pp. 4222–4235, 2020.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *NeurIPS*, 36:8634–8652, 2023.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. pp. 7080–7106, 2026.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. MuSiQue: Multihop questions via single-hop question composition. *arXiv preprint arXiv:2108.00573*, 2022. TACL 2022.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. pp. 16022–16076, 2024.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

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

Shirley Wu, Parth Sarthi, Shiyu Zhao, et al. Optimas: Optimizing compound AI systems with globally aligned local rewards. *arXiv preprint arXiv:2507.03041*, 2025. ICLR 2026.

Chengrun Yang et al. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2024.

Yifan Yang, Ziyang Gong, Weiquan Huang, et al. SkillOpt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative ai by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Hangfan Zhang, Shao Zhang, Kangcong Li, et al. Self-Harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026a.

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *arXiv preprint arXiv:2603.19461*, 2026b.

Jenny Zhang et al. Darwin gödel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025a.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. 2025:34040–34077, 2025b.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. 2026:86069–86100, 2026c.

Yifan Zhang, Yutong Dai, Juntao Tan, et al. DarwinX: Evolving agent harnesses through natural selection. *arXiv preprint arXiv:2608.07545*, 2026d.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, et al. ExpeL: LLM agents are experiential learners. *arXiv preprint arXiv:2308.10144*, 2024. AAAI 2024.

Wangchunshu Zhou, Yixin Ou, Shengwei Ding, Long Li, Jialong Wu, et al. Symbolic learning enables self-evolving agents. *arXiv preprint arXiv:2406.18532*, 2024.

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

A FRAMEWORK DETAILS

Figure 5 shows our evolutionary framework for optimizing a system by evolving its *tunable artifacts* with an agentic proposer, persisting all state to a filesystem. We represent tunable artifacts (§2)—a system's editable prompts, scripts, configurations, and runtime guards—as a real directory of files that the optimizer mutates by adding, deleting, and editing files in place, not a constrained parameter set or DSL—so that the *same* optimization loop can optimize systems of arbitrary form (a skill, an agent harness, a retrieval pipeline) without re-wiring the optimizer for each new family. The Mara Chain (§3) runs on this framework and is the focus of the paper; this appendix details the substrate—the artifact configuration, the seven pluggable components, the two-phase proposer, the concurrent scheduler, and the filesystem memory. Figure 5 overviews the architecture and data flow, and Algorithm 2 gives the full concurrent evolution loop that invokes the chain.

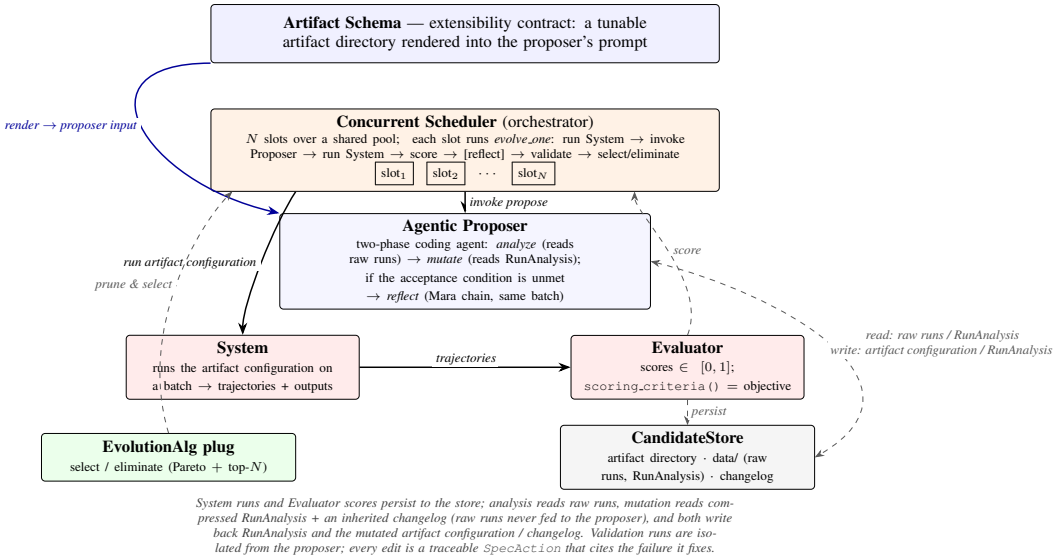


Figure 5: Architecture and data flow (complements Algorithm 2). Each slot runs the candidate artifact configuration through the system runner and scores it with the evaluator (whose scoring criteria is the objective); the proposer is a two-phase coding agent whose analyze phase reads raw runs and whose mutate phase reads the compressed `RunAnalysis` plus an inherited changelog (raw runs are not fed to the mutation phase), writing the mutated artifact configuration and `RunAnalysis` back to the candidate store. The scheduler runs up to N slots over a shared pool backed by the evolution-algorithm plug and the candidate store. Only the artifact-configuration instantiation and the plugins change across task families.

Seven pluggable components. The framework exposes seven pluggable components across three roles. The *target side* describes the system being optimized: the *system runner* runs the artifact configuration on a batch and returns trajectories and outputs; the *evaluator* scores a result and gives the reason; the *data instance* is the per-task contract. The *optimizer side* drives optimization: the *optimizer* (the evolutionary loop) orchestrates selection, proposing, evaluation, the Mara Chain, validation, and elimination under a budget (its default is the concurrent reflective loop); the *proposer* analyzes a failure and rewrites the artifact configuration into a child; and the *evolution algorithm* selects and eliminates candidates. The *candidate store* is the persistence layer. All seven are pluggable; adding a system means subclassing the target side and defining an artifact configuration.

Artifact schema and reliability. An artifact configuration σ contains artifacts $A(\sigma)$, each with a qualitative reliability prior $d(a) \in [0, 1]$. Our concrete schemas order representations as `prose` $<$ `script` $<$ `extension`: natural-language instructions may be ignored, scripts are deterministic once invoked but may not be invoked, and runtime extensions are deterministic when reached. When rollout evidence indicates that an intended modification was bypassed, the proposer may replace or

augment it with a higher-reliability representation permitted by the schema. The concrete directory layouts and reliability instantiations are given in App. B.

A.1 PARETO-FILTERED TOP-N SELECTION

Selection and elimination pseudocode. The `ParetoFrontierEvolutionAlgorithm` implements multi-dimensional leadership-weighted parent sampling and *Pareto-filtered Top-N selection*. Below is compressed pseudocode from `evolution_algorithm/pareto_frontier.py`.

select(num):

$P \leftarrow$ candidates with state=`pending`
if $P = \emptyset$ **then**
  return [] {no work; scheduler waits}
end if
if no candidate scored yet **then**
  return `random.sample`(P, k)
end if
for all candidate $c \in P$ **do**
  $d_c \leftarrow$ #dimensions where c achieves max score (with ε tolerance)
end for
$k \leftarrow \min(\text{num}, |P|)$
$S \leftarrow$ `random.choices`$(P, \text{weights}=\{d_c\}, k)$ {weighted without replacement}
return S

eliminate():

$Q \leftarrow$ candidates with state $\in$ {`pending`, `evolving`}
if $|Q| \leq 1$ **then**
  return []
end if
sort Q by (*generation*, *created_at*) desc {Phase 1: weak Pareto dominance}
for all $a \in Q$ (alive) **do**
  for all $b \in Q$ (alive, $b \neq a$) **do**
    if a weakly dominates b **then**
      mark b as dominated
    end if
  end for
end for
$F \leftarrow$ alive candidates after Phase 1 {Phase 2: top-N truncation}
if $|F| >$ max_candidate_num **then**
  sort F by (*avg_score*, *generation*, *created_at*) desc
  retire F[max_candidate_num :]
end if{only retire `pending`; `evolving` retires after finishing}
return eliminated pairs (cid, reason)

Weak-Pareto dominance definition. Candidate a *weakly dominates* candidate b (denoted $a \succeq b$) if and only if a's score is at least as good as b's on every per-instance dimension, modulo a numerical tolerance ε:

$$a \succeq b \iff \forall\, i:\ a_i + \varepsilon \geq b_i, \qquad \varepsilon = 10^{-8}.$$

This $\geq$ (not $>$) relation is used so that candidates with identical scores are resolved by the pre-sort: deeper generation and newer creation time are preferred, keeping the frontier biased toward more-evolved candidates. The dominator a need only be *no worse* on any dimension—it need not be strictly better. A candidate is removed only if *some* other alive candidate weakly dominates it.

Why top-N truncation. Per-instance Pareto selection, adopted from GEPA, preserves complementary winners but ceases to prune when the validation dimension D grows. Under a simple Bernoulli model, the probability that one candidate weakly dominates another is $P(A \succeq B) = [1 - p(1-p)]^D$, where $p \in (0, 1)$ is a candidate's per-instance pass probability. Because the optimized system is itself a probabilistic model (an LLM agent), an instance outcome is stochastic

Table 4: Component ablations on AppWorld, TerminalBench 2.1, and MuSiQue. The full setting combines Mara Chain with Pareto-filtered Top-N selection. The *−Mara Chain* setting removes retained refinement (single-shot search, Mara-chain depth 0) while keeping Pareto-filtered Top-N; the *−Top-N* setting keeps Mara Chain but relies on Pareto filtering alone. AppWorld and Terminal-Bench 2.1 use the 1-slot configuration; MuSiQue uses 2-slot. −Top-N was not run on Terminal-Bench 2.1 or MuSiQue. Bold marks the best result per metric.

Metric	Mara Chain + Top-N	−Mara Chain	−Top-N
N-TGC	**86.9**	**86.9**	83.3
N-SGC	75.0	**76.8**	67.9
C-TGC	**76.7**	71.9	73.6
C-SGC	**59.0**	51.8	53.2
TB2.1 (%)	**71.9**	57.3	—
MuSiQue nDCG@10 (test)	**0.405**	0.371	—
MuSiQue Recall@10 (test)	**0.460**	0.421	—
MuSiQue nDCG@10 (val)	**0.553**	0.505	—
MuSiQue Recall@10 (val)	**0.605**	0.560	—

rather than certain—$p \notin \{0, 1\}$—so $p(1-p) > 0$ and the domination probability decays exponentially with D, falling below 1% at $D \geq 20$. The Pareto frontier then degenerates to the whole population, which bloats unchecked under GEPA's Pareto-only selection. The top-N Phase 2 truncation bounds the population in this regime. Pairing the cap with per-instance dominance yields *Pareto-filtered Top-N selection*.

Probability-scoring caveats under pass/fail. The `select` operation weights candidates by their max-score dimension count d_c (how many dimensions they lead). Under probabilistic task scoring (each dimension is a Bernoulli trial with pass probability p), two effects degrade this heuristic as the validation dimension D grows:

- *Frontier degeneracy.* $P(a \succeq b) = [1 - p(1-p)]^D$ falls below 1% at $D \geq 20$, so nearly all candidates survive Phase 1 and the population bloats; the top-N Phase 2 truncation bounds it, but the dimension-count weighting in `select` becomes noisy when many candidates share the same d_c (ties are common under p near 0 or 1).
- *Sampling noise.* The weighted-without-replacement sampling distributes selection probability proportional to d_c, but d_c estimates are themselves noisy: a candidate that passed k dimensions on a lucky batch may have d_c inflated, causing it to be selected more often than its true quality warrants. Conversely, a candidate that would dominate on a *different* batch may have $d_c = 0$ and be starved of selection opportunities.

These effects are inherent to any dominance-based EA under stochastic, sparse rewards (pass/fail). The top-N truncation mitigates population bloat; future work could investigate adaptive dimension weighting or Thompson sampling to address the sampling-noise issue.

A.2 Concurrent Scheduler

Concurrent scheduler. The scheduler runs up to $N =$ `num_proposals` candidates concurrently over a shared pool, so a slow slot (e.g., a hard batch in a long Mara Chain) does not block the others. Records carrying 1-slot and 2-slot labels differ, but do not establish separate, additive, or chain-only effects; matched controls are required for component attribution. Its correctness relies on the following invariants.

Scheduler invariants. Three invariants hold throughout the concurrent loop: (1) *no-select back-off*—if `EA.select()` returns [], the scheduler waits on in-flight slots rather than deadlocking; (2) *atomic state-machine flips*—a candidate is flipped to `evolving` before its slot is spawned, preventing double-selection (should `eliminate` ever become `await`-able, serialization must be reintroduced); and (3) *checkpoint safety*—`evolving` candidates are reset to `pending` on restart, and any candidate resumes from its $(epoch, dataset_index)$ cursor, with append-only JSONL preserving full history.

Acceptance predicate and full concurrent loop. For a child score c, parent score p, and acceptance bar δ, write $\mathit{clears}(c, p, \delta)$ for the benchmark- and configuration-dependent predicate that determines admission to validation. Here δ names the acceptance bar; the available documentation does not establish the predicate's raw comparison semantics. The direct child is validated exactly when it clears the bar, and the chain fires exactly when it does not; a chain winner is also validated only when it clears the same bar.

Algorithm 2 is the compressed main loop each slot runs. The Mara Chain (Algorithm 1) is the only non-mechanical step: it fires when a child misses the acceptance bar on its batch and the depth cap $K > 0$; $K{=}0$ disables the Mara Chain and recovers single-shot search. Parents and Mara-chain children are all scored on the *same* batch for a fair comparison, and the budget $\mathcal{B}$ (iterations / rollouts / system_runs / tokens / elapsed) bounds the run, with in-flight slots draining when it is exhausted.

Algorithm 2 Main loop (compressed; full version and concurrent-scheduler invariants in this appendix). The Mara Chain (Algorithm 1) fires when a child misses the acceptance bar.

Require: determinism function $d(\cdot)$, initial configuration σ_0, candidate pool $\mathcal{P}$, budget $\mathcal{B}$, acceptance bar δ, K {Mara-chain depth cap; $K{=}0$ disables the Mara Chain }

concurrently over up to N slots, sharing $\mathcal{P}$ (open a slot only while $\mathcal{B}$ is unexhausted; in-flight slots drain):
for each slot that becomes free **do**
  $\mathit{parent} \leftarrow$ `select()` {pick a pending candidate; if [], wait on in-flight slots}
  flip $\mathit{parent} \rightarrow$ `evolving` {leave the selectable pool}
  $B \leftarrow$ `derive_batch`($\mathit{parent}.\mathit{cursor}$)
  $p \leftarrow$ `evaluate`(parent, B) {baseline: parent on B}
  if p is full **then**
    skip mutation {parent already perfect on B}
  else
    $\mathit{child} \leftarrow$ `create_child`(parent) {inherits artifact; cursor $\rightarrow$ next}
    propose(child, B) {analyze$\rightarrow$mutate the child}
    $c \leftarrow$ `evaluate`(child, B) {same batch: fair comparison}
    if *clears*(c, p, δ) **then**
      validate(child); flip $\mathit{child} \rightarrow$ `pending` {cleared the bar; joins the pool}
    else
      if $K > 0$ **then**
        $\mathit{best} \leftarrow$ *reflection_chain*(child, B, δ) {child missed $\Rightarrow$ reflect; Alg. 1}
        if a winner was found **then**
          validate(best); flip $\mathit{best} \rightarrow$ `pending`
        end if
      else
        retire(child) {missed the bar, no Mara Chain }
      end if
    end if
  end if
  advance $\mathit{parent}.\mathit{cursor}$; flip $\mathit{parent} \rightarrow$ `pending`
  eliminate($\mathcal{P}$) {weak Pareto + top-N; every slot}
  persist(pool) {memory + checkpoint}
end for

A.3 Proposer and Tunable-Artifact Edits

Pi proposer extensions: anti-cheating and reliability guards. The proposer is an autonomous coding agent (we use the Pi coding agent (Earendil Inc., 2025) for its unbypassable extension guards) with filesystem and shell access. Two concerns arise from this autonomy, and the framework addresses both through the Pi runtime's extension guard—the `extension` guard of the schema (§2), which always applies and the agent cannot bypass.

(i) Anti-cheating: validation isolation (`block-val-system-run.ts`). The proposer could otherwise read the held-out validation trajectories and overfit the validation set. The extension intercepts

every `read/write/edit/bash/grep/find/ls` tool call and blocks any whose target path matches `val_system_run/`:

```
pi.on("tool_call", (event) => {
  const pathToCheck =
    event.input.file_path ?? event.input.command ?? /* ... */;
  if (/val_system_run/.test(pathToCheck))
    return { block: true,
           reason: "Access to val_system_run is not allowed ..." };
});
```

The proposer therefore cannot read validation-set trajectories, so the validation score reflects generalization, not leakage.

(ii) Reliability: enforced changelog discipline (`enforce-changelog-cli.ts`). Because LLM instruction-following is probabilistic, the proposer frequently ignores the instruction to use the sanctioned CLI and writes `changelog.jsonl` itself, often producing malformed JSONL (e.g., several JSON objects concatenated on one line) that breaks downstream parsing. The extension blocks direct writes to `changelog.jsonl`—via `write/edit` or shell redirection/`tee`—and forces the proposer to record every edit through the `append-changelog` CLI, which emits one well-formed JSONL object per edit. This minimizes the probability of a malformed lineage record rather than relying on the model to follow the format.

These guards constrain the proposer's access to the validation signal and its lineage bookkeeping. They are a concrete instance of the "hard runtime guard the model cannot bypass" instantiated by the schema's `extension` guard.

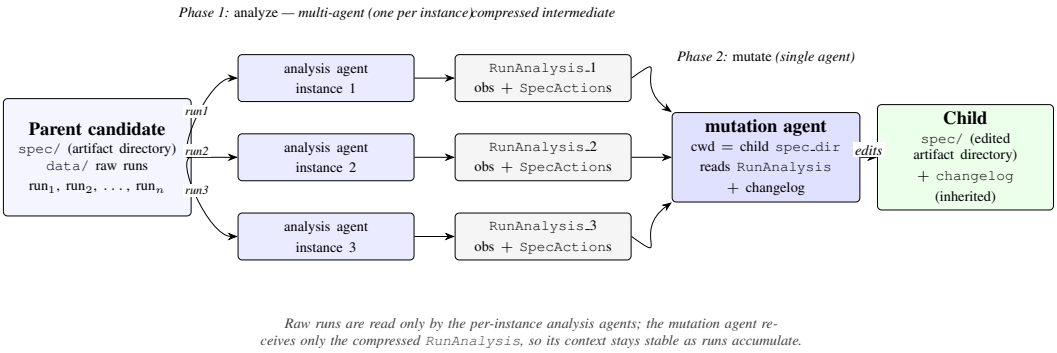


Figure 6: Two-phase propose. *Analyze* is multi-agent: one small analysis agent per data instance reads that instance's raw run (in the parent's data directory) and emits a structured `RunAnalysis`. *Mutate* is a single agent that reads only the compressed `RunAnalysis` set (plus an inherited changelog) and edits the child's tunable artifacts—raw runs are never fed to it, so its context stays stable as runs accumulate.

ArtifactAction and ChangeLog. Every mutation is a structured `ArtifactAction` that must cite at least one traced failure—an AHE-style falsifiable contract: an action with non-empty `resolves` referencing no observed failure is rejected. Each accepted action is appended to the candidate's `changelog.jsonl` as a `ChangeLogEntry`, and a child inherits its parent's changelog, so the attributed edit history accumulates down the lineage. Their schemas:

```
ArtifactAction { // one tunable-artifact edit
  file: str // concrete path relative to the artifact root
  operation: "add" | "delete" | "modify"
  artifact_issue: str // the defect in this file (what is wrong / missing)
  change: str // exact change; BEFORE/AFTER format for "modify"
  resolves: list[int] // indices into RunAnalysis.trajectory_analysis;
                      // non-empty -- must cite >=1 traced failure
}

ChangeLogEntry { // one line in changelog.jsonl (inherited by children)
  timestamp: datetime
  type: "feat" | "fix" | "refactor" | "perf" | "docs" | "style" | "chore"
  subject: str // imperative summary, <= 72 chars
```

```
  body: str // observed failure pattern + change + expected impact
  diff: str // unified (git) diff of the actual changes
  files_modified: list[str]
  author: str // proposer identity (e.g. pi, claude_code)
}
```

The candidate state machine is `pending` ↔ `evolving` → `unavailable`: `pending` candidates are selectable by the evolution algorithm, `evolving` means a slot owns the candidate, and `unavailable` is used by in-flight Mara Chain children.

Proposer input and output. The mutation phase's operating prompt is assembled at render time from four sources, none hardcoded in proposer code: (i) the *rendered schema*—the component tree plus each component's role and determinism; (ii) the *objective*, taken from the evaluator's `scoring_criteria()`; (iii) the *diagnosis contract*, the `RunAnalysis`/`ArtifactAction` schemas the analysis phase must produce; and (iv) the *compressed inputs from the analyze phase*—the per-instance `RunAnalysis` set and the inherited changelog, not raw runs. The mutation agent runs with the child's `artifact_dir` as its working directory. The determinism ordering travels with the schema: when a fix at the current component is bypassed, the rendered prompt instructs the proposer to move it to a higher-determinism asset rather than restating it, so the instruction is not hardcoded in the loop. This is the weld between the schema and the Mara Chain—the chain drives edits down the determinism gradient, and the gradient itself is injected from the schema.

What the proposer ultimately delivers is a new child candidate: the edited `artifact/` directory (the genome, mutated by applying the committed `ArtifactActions`) plus an appended `ChangeLogEntry` recording each edit and the failure it resolves. The candidate is then scored by the evaluator and, if accepted, enters the population with this attributed lineage inherited by its own children. Full prompt templates and the rendering pipeline are in the released code.

A.4 Filesystem as Full-Fidelity Memory

Filesystem as full-fidelity memory. The filesystem is the substrate that makes *optimize anything* tractable. An evolution run produces far more data—full execution trajectories, per-instance scores, the analyzer's structured `RunAnalysis`, the proposer's attributed edits, and the lineage linking each child to its parent—than could be held in memory or stuffed into an LLM context window. Keeping all of it on a filesystem lets the analysis phase access the raw records and the mutation phase access its structured outputs, including layouts that are hard to anticipate in advance. Persistence sits behind the candidate-store interface: the default is a local filesystem, but it can be reimplemented to sync to a remote or cloud backend, so the loop is not coupled to a single machine or storage medium. This follows the filesystem-as-memory idea of AHE and Meta-Harness (Lin et al., 2026; Lee et al., 2026); the on-disk layout is shown in Fig. 7 and detailed below.

We keep this data on a filesystem rather than in an in-memory or prompt-compressed representation for two reasons, each grounded in what we observed during evolution runs:

- *No information loss, and drill-down on demand.* Keeping run and analysis data on disk means the analysis phase can drill down from a compressed `RunAnalysis` into the raw trajectory when a diagnosis is uncertain; the mutation phase receives the structured analysis rather than raw trajectories.
- *Reproducibility and resumability.* Append-only logging makes every run reproducible and resumable from any checkpoint, which matters for evolution runs that span days and must survive crashes.

Licensed compression. The mutation phase reads a *compressed* `RunAnalysis`, not the raw runs, so its context stays stable as runs accumulate. This compression is *licensed*: each `RunAnalysis` is produced by a small per-instance analysis agent whose job is small enough that distillation is deliberate, and the raw runs remain on disk for drill-down—so it is not the trace-destroying compression Meta-Harness (Lee et al., 2026) warns against, but a structured evidence layer on top of full-fidelity logs.

On-disk layout of a run's workspace. A run is a timestamped workspace; Figure 7 shows its directory tree. Under `candidates/{candidate_id}/`, `artifact/` is the genome (inher-

ited and mutated) and `data/` is the phenotypic memory, holding `meta.json` (lineage, lifecycle state `pending/evolving/unavailable`, and the per-candidate progress cursor), an append-only `changelog.jsonl` (the attributed edit history), `summary.json` (per-instance scores), the raw system and validation runs, and the proposer's analysis and mutation artifacts. The `val_system_run/` directory is hard-isolated from the proposer, enforcing the validation isolation above. The sibling `logs/` directory holds run-level artifacts that span the whole optimization: `statistics.json` (best score, score history, candidate counts, usage), an append-only `iteration_records.jsonl` (one record per iteration), and `parameters.jsonl` (the run's parameters).

```
workspace_dir/
+-- candidates/
|   \-- {candidate_id}/
|       +-- artifact/              # genome: inherited & mutated
|       \-- data/                  # phenotypic memory
|           +-- meta.json          # lineage, state, progress cursor
|           +-- changelog.jsonl    # append-only attributed edit history
|           +-- summary.json       # per-instance scores
|           +-- system_run/{data_id}/      # system run trajectories
|           +-- val_system_run/{data_id}/  # validation runs (proposer-forbidden)
|           \-- proposer_run/               # proposal agents' artifacts
|               +-- analysis/
|               |   +-- trajectory/{child_id}.json  # analyzer trajectory
|               |   \-- result/{data_id}.json       # RunAnalysis (diagnoses)
|               \-- mutation/{child_id}.json        # proposer trajectory & edits
\-- logs/                          # run-level, global across candidates
    +-- statistics.json            # optimization stats (best score, history, counts)
    +-- iteration_records.jsonl    # append-only, one record per iteration
    \-- parameters.jsonl           # run parameters
```

Figure 7: On-disk layout of a run's workspace in our store. `candidates/{candidate_id}/` holds each candidate's `artifact/` genome and `data/` phenotypic memory (run data, the analyzer's diagnoses in `analysis/result/`, the proposer's edits in `mutation/`, and the lineage); `logs/` holds the run-level statistics, iteration records, and parameters.

B ARTIFACT SCHEMAS

Each of the three benchmarks instantiates a different artifact configuration—a directory of files with an artifact-reliability structure (§2). Below we give the on-disk layout and the `TunableArtifactSchema` code for each, and Table 5 makes the artifact-reliability instantiation concrete: *prose* artifacts have $d < \theta$ and may be bypassed, *script* artifacts execute deterministically once invoked (but invocation may be skipped), and *extension* artifacts execute deterministically when reached. Mara Chain inherits the artifact configuration along one lineage and can replace a repeatedly bypassed prose artifact with a script or extension when supported by the rollout evidence.

AppWorld (skill artifact configuration).

```
appworld-skill/
+-- skill/                       # NL instructions (agent may ignore)
|   +-- SKILL.md                 # AppWorld strategy instructions
|   +-- references/              # API docs, examples (loaded on demand)
|   +-- scripts/                 # verification helpers
|   +-- (additional files)
```

`APPWORLD_SKILL_TUNABLE_ARTIFACT_SCHEMA` in `appworld_tunable_artifact_def.py`:

```
APPWORLD_SKILL_TUNABLE_ARTIFACT_SCHEMA: TunableArtifactSchema = FolderSchema(
    name="appworld-skill",
    files=[
        FolderSchema(name="skill", files=[
            FileSchema(name="SKILL.md",
                       description=_APPWORLD_SKILL_MD_DESCRIPTION),
            FolderSchema(name="scripts",
                         description=_SCRIPTS_DIR_DESCRIPTION),
            FolderSchema(name="references",
                         description=_APPWORLD_REFERENCES_DIR_DESCRIPTION),
            FileSchema(name="...",
                       description=_ADDITIONAL_FILES_DESCRIPTION),
        ]),
```

Table 5: Asset/determinism instantiation of each artifact configuration. $d(a)$ is a qualitative prior, not a measured probability. Scripts are deterministic after invocation, and "on-demand" assets are loaded only when the model invokes them.

Artifact configuration	Asset	Type	$d(a)$	Role at runtime
AppWorld	`skill/SKILL.md`	prose	$< \theta$	strategy instructions (agent may ignore)
	`skill/references/`	prose	$< \theta$	API docs/examples, on-demand
	`skill/scripts/`	script	$\geq \theta$	verification helpers (deterministic when invoked)
TerminalBench	`agent/kira_agent.py`	extension	≈ 1	harness code: loop/tools/middleware, cannot bypass
	`agent/prompt-templates/`	prose	$< \theta$	system prompt template (NL)
	`skill/SKILL.md`	prose	$< \theta$	task-solving strategy
	`skill/scripts/`	script	$\geq \theta$	verification helpers
	`skill/references/`	prose	$< \theta$	detailed docs, on-demand
MuSiQue	`pipeline.json`	config	≈ 1	DAG: node order + params, executed by runner
	`nodes/*.py`	script	$\geq \theta$	node implementations, executed, cannot bypass
	`prompt/rewrite.md`	prose	$< \theta$	rewrite prompt, read but not enforced

```
    ],
)
```

The skill contains prose, references, and optional scripts. A script is deterministic after invocation but does not guarantee invocation; the observed case trace records one progression to a script, not a necessary repair rule.

TerminalBench 2.1 (agent-harness artifact configuration).

```
terminalbench-kira/
+-- agent/                      # harness code (deterministic, cannot bypass)
|   +-- kira_agent.py           # AgentHarness class: agent loop, tools,
|   |                           # reminder/middleware, budget enforcement
|   +-- prompt-templates/
|   |   \-- terminus-kira.txt   # system prompt template (str.format)
|   +-- (additional files)
+-- skill/                      # NL instructions (agent may ignore)
    +-- SKILL.md                # task-solving strategy
    +-- scripts/                # verification helpers
    +-- references/             # detailed docs (loaded on demand)
    +-- (additional files)
```

`TERMINALBENCH_KIRA_TUNABLE_ARTIFACT_SCHEMA` in `terminalbench_kira_tunable_artifact_def.py`:

```
TERMINALBENCH_KIRA_TUNABLE_ARTIFACT_SCHEMA: TunableArtifactSchema = FolderSchema(
    name="terminalbench-kira",
    description=_KIRA_ARTIFACT_DIR_DESCRIPTION,
    files=[
        FolderSchema(
            name="agent",
            description=_AGENT_DIR_DESCRIPTION,
            files=[
                FileSchema(name="kira_agent.py",
                           description=_AGENT_MAIN_DESCRIPTION),
                FolderSchema(name="prompt-templates",
                             description=_PROMPT_TEMPLATES_DIR_DESCRIPTION,
                             files=[
                    FileSchema(name="terminus-kira.txt",
                               description=_PROMPT_TEMPLATE_DESCRIPTION),
                ]),
                FileSchema(name="...",
                           description=_ADDITIONAL_FILES_DESCRIPTION),
            ],
```

```
        ),
        FolderSchema(
            name="skill",
            description=_KIRA_SKILL_DIR_DESCRIPTION,
            files=[
                FileSchema(name="SKILL.md",
                           description=_KIRA_SKILL_MD_DESCRIPTION),
                FolderSchema(name="scripts",
                             description=_SCRIPTS_DIR_DESCRIPTION),
                FolderSchema(name="references",
                             description=_REFERENCES_DIR_DESCRIPTION),
                FileSchema(name="...",
                           description=_ADDITIONAL_FILES_DESCRIPTION),
            ],
        ),
    ],
)
```

`agent/` is the unbypassable extension guard (harness code always runs); `skill/` is bypassable prose. These types permit edits such as `kira_agent.py` code at extension and `SKILL.md` rules at prose.

MuSiQue (retrieval-pipeline artifact configuration).

```
rag-pipeline/
+-- pipeline.json               # DAG config: node order + params (k, top_m, ...)
+-- nodes/                      # node implementations (deterministic Python)
|   +-- recall_bm25.py          #   BM25 retriever
|   +-- recall_dense.py         #   dense retriever (embedding)
|   +-- rerank_cross.py         #   cross-encoder rerank
|   +-- graph_rescore.py        #   graph-based rescore
|   +-- truncate.py             #   top-k truncation
|   +-- query_rewrite.py        #   LLM query rewrite (was skeleton)
|   +-- fuse_rrf.py             #   RRF fusion (added by optimizer)
|   +-- graph_expand.py         #   entity-overlap expansion (added)
\-- prompt/
    \-- rewrite.md              # query rewrite prompt (NL, node reads on demand)
```

`RAG_PIPELINE_TUNABLE_ARTIFACT_SCHEMA` in `rag_pipeline_tunable_artifact_def.py`:

```
RAG_PIPELINE_TUNABLE_ARTIFACT_SCHEMA: TunableArtifactSchema = FolderSchema(
    name="rag-pipeline",
    description=_ARTIFACT_DIR,
    files=[
        FileSchema(name="pipeline.json",
                   description=_PIPELINE_JSON),
        FolderSchema(name="nodes",
                     description=_NODES, files=[]),
        FolderSchema(name="prompt",
                     description=_PROMPT_DIR, files=[]),
        FileSchema(name="...",
                   description=_ADDITIONAL_FILES_DESCRIPTION),
    ],
)
```

`pipeline.json` and `nodes/` are deterministic (the runner executes them, so $d \geq \theta$); the `prompt/` directory contains NL prompts the system reads but does not enforce. The before/after evolved DAG is in App. H.

C Formal Conditions and Proofs

These statements are generic to any proposal process satisfying the following assumptions; they do not prove that Mara Chain improves any probability. For a task τ, let $G_\tau = (C, \prec)$ be a finite directed acyclic graph of conditions, where predecessors of c must be satisfied before c is observable. Let $\mathrm{Sat}(\sigma) \subseteq C$ be the conditions satisfied by configuration σ, and let

$$F(\sigma) = \{c \in C \setminus \mathrm{Sat}(\sigma) : \mathrm{pred}(c) \subseteq \mathrm{Sat}(\sigma)\}.$$

We assume *residual completeness*: the evaluator reports every currently unsatisfied observable condition, so $\delta(\sigma, \tau) = F(\sigma)$. An accepted update is required to be non-regressing: $\mathrm{Sat}(\sigma) \subseteq \mathrm{Sat}(\sigma')$. These are assumptions about the abstraction, not properties established by the experiments.

Theorem 1 (Persistent residual under unsupported accepted updates)**.** Let $c^\star \in \delta(\sigma, \tau)$. Under residual completeness and non-regressing accepted-update semantics, if every accepted proposal distribution assigns zero probability to updates that satisfy $c^\star$, then $c^\star \in \delta(\sigma_t, \tau)$ after every accepted update t.

Proof. An accepted update cannot remove $c^\star$ by assumption. Non-regression preserves the predecessors that make it observable, and residual completeness reports it. Induction over accepted updates proves the claim. □

Realizability and conditional support. For every condition c that becomes exposed under a configuration satisfying its predecessors, *realizability* means that at least one admissible, accepted update exists that satisfies c while preserving all currently satisfied conditions. Let $\mathcal{H}_{c,j}$ be the sigma field generated by the complete history before retained attempt j at exposed condition c.

Theorem 2 (Conditional repeated-attempt bound)**.** Suppose residual completeness, realizability, and non-regressing accepted updates hold. Further assume, rather than infer, that for every exposed c and attempt j, $\Pr(\text{attempt } j \text{ proposes and accepts an update satisfying } c \mid \mathcal{H}_{c,j}) \geq p > 0$. If G_τ has at most n conditions and

$$m \geq \left\lceil \frac{\log(n/\eta)}{-\log(1-p)} \right\rceil,$$

then allocating m attempts to each exposed condition satisfies all conditions with probability at least $1 - \eta$, using at most nm attempts.

Proof. For an exposed condition, the conditional probability that all m attempts fail is at most $(1-p)^m \leq \eta/n$. Non-regression and residual completeness expose successors in topological order. A union bound over at most n conditions gives the result. □

D MARA CHAIN: FULL DETAIL

D.1 CHAIN CONTEXT

For a direct proposal, the documented analysis phase reads the candidate's raw run records, while the direct mutation phase receives only the resulting compressed `RunAnalysis` and inherited changelog. For a Mara-chain proposal, the documentation specifies the following lineage material for the analysis phase, all pinned to the chain and the single failing instance:

- **Lineage path** ($v_0 \to \cdots \to v_{k-1}$): the ordered sequence of prior failed children the node is continuing.
- **Per-instance score table with two delta columns**: each prior node's score plus $\Delta_{\text{root}\to\text{cid}}$ (cumulative gain) and $\Delta_{\text{prev}\to\text{cid}}$ (gain over the preceding node).
- **Attributed changelog**: the inherited edit history with every entry tagged by the candidate that authored it.
- **Prior nodes' `RunAnalysis` diagnoses**: inherited diagnosis content used to condition the next refinement step.
- **Prior nodes' run files**: the latest trajectory of each prior node on this instance, for cross-version comparison.

The chain mutation phase receives the resulting analyses and changelog rather than raw run files. The documentation does not specify how any pre-loading, direct file access, and store-mediated access interact within the analysis phase; this scope is therefore unspecified. The node also receives the inherited residual δ_{k-1}.

D.2 FIVE-METHOD DIAGNOSIS AND THE CATEGORY TABLE

The Mara-chain analyze phase forces five diagnostic methods, each a concrete operation on the cross-attempt context:

1. **Delta-guided evidence tracing.** Attack the inherited residual.

2. **Success extraction.** Lift what a passing roll did right.
3. **Oscillation detection.** A symptom that flips across nodes signals a regression to undo.
4. **Inconsistency-pattern discovery.** Variance under the *same* artifact configuration is nondeterminism, not an artifact bug.
5. **Run-evidence corroboration.** Every diagnosis must cite a span in the raw run files.

Each prior-attempt diagnosis is classified into a structured category table:

A *never-triggered*—the new content was never reached; fix the trigger or move it where it always fires.

B *content-wrong*—fired but wrong; REMOVE or REPLACE.

C *vague*—correct in spirit but unactionable; replace with a concrete WHEN/DO/VERIFY procedure.

D *conflict*—overridden by other content; resolve and cite the conflicting candidate.

E *root-cause-irrelevant*—the failure has a different cause.

F *direction-exhausted*—tried repeatedly with no gain; switch component or mechanism.

G *inconsistent*—same artifact configuration, different outcome; harden the winning path.

H *lost-win*—an effective change was removed or overridden; RESTORE and guard it.

I *mechanism-too-weak*—the artifact configuration already contains the correct content but the system bypassed it; *raise the fix to a higher-reliability artifact representation*.

Preferences: prefer REMOVE/REPLACE over ADD; cite the responsible chain candidate in each `artifact_issue`; make no tit-for-tat additions; and hunt *suppression* edges and *lost-win* entries so a fix for one instance does not re-break another.

D.3 Single-Shot Baseline

Single-shot search. By *single-shot search* we mean the standard mutate–evaluate–select loop: mutate a parent into a candidate, evaluate the candidate once on a batch of tasks (a few trials only to denoise its pass rate), and admit it to the population only if *clears*$(s_B(\text{child}), s_B(\text{parent}), \delta)$ for the benchmark- and configuration-dependent acceptance predicate—otherwise discard or revert it and propose a fresh candidate next iteration. A rejected candidate is never re-run on the same tasks to learn why it failed. This is the regime of prior artifact and prompt optimizers (§5) and the baseline the Mara Chain improves; Algorithm 2 with $K{=}0$ recovers it.

Theorems 1 and 2 are generic conditional statements, not a comparison of the recorded configurations. The first concerns any accepted-update process whose proposal support excludes a condition; the second adds realizability, non-regression, and a history-conditional positive-support assumption. Neither assumption is established by the empirical comparisons.

E Experiment Protocol

Data selection. We pick three benchmarks that span three classes of tunable artifacts (§2): AppWorld (skill artifact configuration), TerminalBench 2.1 (agent-harness artifact configuration), and MuSiQue (retrieval-pipeline artifact configuration). Documented comparisons use harness optimizers (AHE, Meta-Harness), human CLI agents (Codex, OpenCode), and the un-optimized base on TerminalBench 2.1, and the hand-written default pipeline on MuSiQue. AppWorld records are static evaluations of optimized skill configurations, not documented optimizer runs or a method comparison.

Data, splits, and benchmark versions. *AppWorld* (v0.2.0.dev0, commit `e9325d3`): we use the benchmark's `train_val_split.json`, which merges the official `train` and `dev` tasks into a **147**-task training pool from which optimization minibatches are drawn. The validation set contains **200** tasks sampled from the `test_normal`/`test_challenge` pools (**100** each, seed 42), stratified so that $\sim$10% are solvable by the empty-skill baseline agent (23 pass / 177 fail). Final static evaluations are reported on the full test splits: **585** tasks (**168** `test_normal` and

417 `test_challenge`; Table 1), which therefore include the 200 validation tasks; validation scores are used only for optimization guidance and climbing-speed curves, never reported as final results. `max_steps=` 50 throughout. *MuSiQue*: `tools/prepare_musique.py` converts the raw `.jsonl` into BEIR-style files (`corpus.jsonl` with globally unique `doc_id`; `queries.jsonl` with gold from `is_supporting=True`). We use the 4-hop subset: **300** train (from `musique_full_v1.0_train`, 4-hop, first 300 with gold) / **100** val (from `musique_full_v1.0_dev`, 4-hop, 100 for development) / **300** test (from the `dev` full set, 4-hop, excluding the 100 val ids; test and val have zero overlap) (Table 2). *TerminalBench 2.1*: **89** tasks; there is no separate val set—all 89 tasks serve as both the training set for the optimizer and the evaluation set. This follows the standard protocol used by prior work on Terminal-Bench 2.0/2.1 (including AHE and Meta-Harness), which optimize directly on the full task set and report pass rate over the same 89 tasks. Because there is no held-out set, the reported TerminalBench results are in-sample optimized-task rates rather than held-out scores (Figure 1(b)).

Metrics. *AppWorld:* strict pass/fail per task. Each task has multiple test requirements (assertions on database state, answer correctness, side effects). A task scores 1.0 only if *all* requirements pass; 0.0 if any fails. The test suite is deterministic (time frozen, DB state compared exactly). **Task Goal Completion (TGC)** = average of per-task scores across all tasks. **Scenario Goal Completion (SGC)** = for each scenario (a group of related tasks sharing state), take the minimum score among its tasks (an entire scenario passes only if *every* task in it passes), then average across scenarios. TGC and SGC are reported separately on the Normal and Challenge splits (Table 1). *TerminalBench 2.1:* pass/fail (binary) per task; reported as pass rate over all 89 tasks (Figure 1(b)). *MuSiQue:* nDCG@10 and Recall@10, both computed over the top-10 retrieved documents per query (Table 2). nDCG@10 (normalized Discounted Cumulative Gain at cutoff 10) measures ranking quality: it rewards placing golden documents higher in the list, with logarithmically diminishing returns for lower ranks. Recall@10 measures coverage: the fraction of all golden documents that appear in the top-10 retrieved results. Both metrics are standard information-retrieval measures and are computed deterministically from the ranked output of the retrieval pipeline.

The archived AppWorld final-result records provide TGC and SGC values over the reported Normal and Challenge task sets. They do not establish whether the final evaluator, optimization reward, minibatches, or validation gate used the same interface, or how any optimized artifact configuration was produced; no stronger reproducibility claim is made here.

Per-method launch parameters. The documented framework settings for TerminalBench 2.1 and MuSiQue use a GLM-5.1 Pi coding-agent proposer; AppWorld proposer settings are unavailable. The documented settings share `max_turns=` 200, `timeout=` 3600 s, `skip_perfect_score_runs`=False, `last_n_analysis=` 10, `epochs=` 3.

AppWorld. The archived local records establish only the 168 `test_normal` and 417 `test_challenge` static final evaluations of optimized skill configurations. No optimizer manifests exist; optimization-run settings, system and proposer assignments, minibatches, validation procedures, budgets, and baseline launch parameters are unavailable. The labels 1-slot and 2-slot have no documented historical meaning. No AppWorld protocol beyond those static records is claimed. *TerminalBench 2.1*:

- MARA CHAIN (system GLM-5, system concurrency 4, evaluator concurrency 4, proposer concurrency 20, `max_candidate_num=` 1, `batch_size=` 4, budget 100 h wall-clock):
  - +Mara Chain: `num_proposals=` 1, `max_reflection_iterations=` 5.
  - −Mara Chain: `num_proposals=` 1, `max_reflection_iterations=` 0.
- AHE (Lin et al., 2026): system model GLM-5, analyzer/improve model GLM-5.1 (max tokens 64000, temperature 0.2), max iterations 10, target pass rate 0.95, pass@k with k=2, eval concurrency 4, auto-rollback-harmful True, real NexAU evolve-agent enabled, budget 100 h wall-clock. AHE's method (AgentDebugger analyze → NexAU evolve-agent mutate → falsify with verdict → optional auto-rollback) is faithfully reproduced from the open-source repo (`https://github.com/china-qijizhifeng/agentic-harness-engineering`) on TerminalBench 2.1, evolving the kira harness. The base model is GLM-5 (vs. the paper's GPT-5.4); absolute scores are not directly comparable to the paper, only same-environment

comparisons are valid. For a fair same-environment comparison with MARA CHAIN (which does not use external knowledge), we *disabled AHE's external-knowledge acquisition*—its retrieval of outside context during harness evolution—in our runs.

- Meta-Harness (Lee et al., 2026): reproduced from the open-source Meta-Harness repo (`https://github.com/stanford-iris-lab/meta-harness`), which we adapt to the kira harness and evaluate under local Docker on the Terminal-Bench 2.1. Launch parameters: system model GLM-5, proposer model GLM-5.1 run as a Claude Code coding agent, pass@k with k=2 (two trials per task), eval concurrency 4, and a total budget of 100 h wall-clock.

MuSiQue:

- MARA CHAIN (non-agentic pipeline; evaluator concurrency 8, proposer concurrency 10, `max_candidate_num`= 3, budget 20 h wall-clock): `num_proposals`= 2 (2-slot), `batch_size`= 6, `min_improvement_per_batch`= 2.0; the reported optimized pipeline uses `max_reflection_iterations`= 3. The pipeline's internal node models (BAAI/bge-m3 for dense recall, BAAI/bge-reranker-v2-m3 for cross-encoder rerank (Beijing Academy of Artificial Intelligence, 2024), GLM-5.2 for `query_rewrite`) are part of the evolved pipeline (App. H).

F APPWORLD SCORE VERSUS OPTIMIZATION TIME

Figure 8 plots the same AppWorld optimization trajectories as Figure 1(a), but against effective wall-clock time instead of optimization rollouts. The efficiency gap is not an artifact of per-rollout cost differences: Mara Chain reaches the 0.8 validation score in 11.9 h, about 3× faster than GEPA (35.8 h), while ACE and SkillOpt-Lite do not reach 0.8 at any point in the recorded runs and plateau at 0.770 and 0.665, respectively. Mara Chain also attains the highest final score (0.870 versus GEPA's 0.805), matching the rollout-based view.

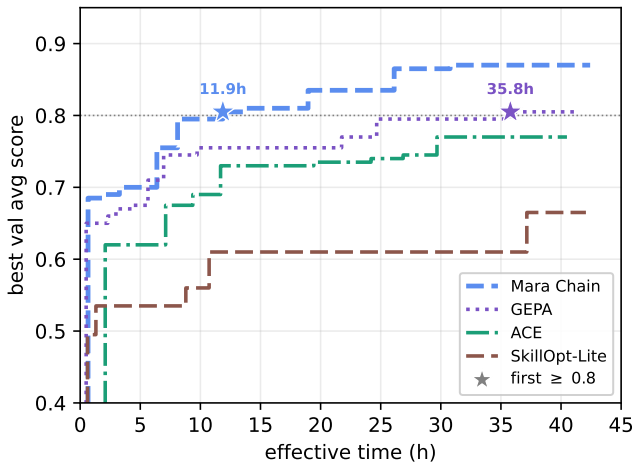


Figure 8: AppWorld validation best score versus effective optimization time (hours); the wall-clock companion of Figure 1(a). The dotted line marks the 0.8 validation score and ★ annotates each method's first recorded point at or above it. Mara Chain reaches 0.8 in 11.9 h, about 3× faster than GEPA (35.8 h), and attains the highest final score; ACE (final 0.770) and SkillOpt-Lite (0.665) never reach 0.8.

G TERMINALBENCH 2.1 PER-TASK REWARD

Table 6 lists the per-task reward ($r \in \{0, 1\}$) on the 89-task set behind Figure 1(b). Column means reproduce the reported pass rates: 51.7, 49.4, 51.7, 71.9, 57.3, 51.7, 49.4%. Columns (left → right):

Codex, OpenCode, Kira base, MARA CHAIN (ours), −Mara Chain ablation, AHE, Meta-Harness. All seven use the same frozen GLM-5 system model.

Table 6: TerminalBench 2.1 per-task reward (1 = task passes, 0 = otherwise).

Task	*Codex*	*OpenCode*	*Kira*	*MARA CHAIN*	*−Mara Chain*	*AHE*	*Meta-Harness*
bn-fit-modify	1	1	1	1	1	1	1
build-pmars	1	1	1	1	1	1	1
cobol-modernization	1	1	1	1	1	1	1
constraints-scheduling	1	1	1	1	1	1	1
count-dataset-tokens	1	1	1	1	1	1	1
crack-7z-hash	1	1	1	1	1	1	1
custom-memory-heap-crash	1	1	1	1	1	1	1
distribution-search	1	1	1	1	1	1	1
feal-differential-cryptanalysis	1	1	0	1	0	0	0
financial-document-processor	1	1	1	1	0	1	0
fix-code-vulnerability	1	1	1	1	1	1	0
fix-ocaml-gc	1	1	0	1	1	1	0
git-leak-recovery	1	1	1	1	1	1	1
headless-terminal	1	1	1	1	1	1	1
hf-model-inference	1	1	1	1	1	1	1
large-scale-text-editing	1	1	1	1	0	1	1
largest-eigenval	1	1	0	1	1	1	1
llm-inference-batching-scheduler	1	1	1	1	0	1	1
log-summary-date-ranges	1	1	1	1	1	1	0
mcmc-sampling-stan	1	1	1	1	1	1	1
merge-diff-arc-agi-task	1	1	1	1	0	0	1
modernize-scientific-stack	1	1	1	1	1	1	1
multi-source-data-merger	1	1	1	1	1	1	1
nginx-request-logging	1	1	1	1	1	1	1
openssl-selfsigned-cert	1	1	1	1	1	1	1
portfolio-optimization	1	1	0	1	1	1	0
prove-plus-comm	1	1	1	1	1	1	1
pytorch-model-cli	1	1	1	1	1	1	1
pytorch-model-recovery	1	1	1	1	1	1	1
regex-log	1	1	1	1	0	1	1
sparql-university	1	1	1	1	1	0	0
sqlite-db-truncate	1	1	1	1	1	1	1
sqlite-with-gcov	1	1	1	1	1	1	1
tune-mjcf	1	1	1	1	0	0	0
vulnerable-secret	1	1	1	1	1	1	1
cancel-async-tasks	1	0	1	1	1	0	0
code-from-image	1	0	0	1	0	1	1
compile-compcert	1	0	1	1	0	0	0
configure-git-webserver	1	0	0	1	1	1	1
fix-git	0	1	1	1	1	1	1
git-multibranch	1	0	1	1	1	1	1
overfull-hbox	1	0	1	1	1	1	1
password-recovery	0	1	1	1	1	1	1
path-tracing-reverse	0	1	1	1	0	0	0
pypi-server	0	1	1	1	1	1	1
qemu-alpine-ssh	0	1	0	1	1	1	1
qemu-startup	0	1	0	1	1	0	1
query-optimize	1	1	0	0	1	0	0
reshard-c4-data	1	0	1	1	0	0	1
sanitize-git-repo	1	0	0	1	0	0	0
torch-tensor-parallelism	1	0	0	1	1	0	0

continued on next page

Table 6: TerminalBench 2.1 per-task reward (continued).

Task	*Codex*	*OpenCode*	*Kira*	*MARA CHAIN*	*−Mara Chain*	*AHE*	*Meta-Harness*
winning-avg-corewars	0	1	0	1	1	0	0
adaptive-rejection-sampler	0	0	0	1	1	0	0
break-filter-js-from-html	0	0	0	1	1	1	1
build-pov-ray	0	1	1	0	1	0	0
circuit-fibsqrt	0	0	0	1	1	0	0
db-wal-recovery	0	0	0	1	1	0	0
extract-elf	0	0	0	1	0	1	1
install-windows-3.11	0	0	0	1	0	0	0
kv-store-grpc	0	0	1	1	1	1	1
mailman	0	0	1	1	1	1	1
model-extraction-relu-logits	0	0	0	1	0	0	0
mteb-retrieve	0	0	0	1	1	0	0
polyglot-c-py	0	0	1	1	1	1	1
protein-assembly	0	0	0	1	0	0	0
rstan-to-pystan	0	0	0	1	1	0	1
write-compressor	1	0	0	0	0	0	0
build-cython-ext	0	0	0	0	0	0	0
caffe-cifar-10	0	0	0	0	0	0	0
chess-best-move	0	0	0	0	0	0	0
dna-assembly	0	0	0	0	0	0	0
dna-insert	0	0	0	0	0	0	0
extract-moves-from-video	0	0	0	0	0	0	0
feal-linear-cryptanalysis	0	0	0	0	0	0	0
filter-js-from-html	0	0	0	0	0	1	0
gcode-to-text	0	0	0	0	0	0	0
gpt2-codegolf	0	0	0	0	0	0	0
make-doom-for-mips	0	0	0	0	0	0	0
make-mips-interpreter	0	0	0	0	0	0	0
mteb-leaderboard	0	0	0	0	0	0	0
path-tracing	0	0	0	0	0	0	0
polyglot-rust-c	0	0	1	0	1	1	0
raman-fitting	0	0	0	0	0	0	0
regex-chess	0	0	0	0	0	0	0
sam-cell-seg	0	0	0	0	0	0	0
schemelike-metacircular-eval	0	0	1	0	0	0	0
torch-pipeline-parallelism	0	0	0	0	0	0	1
train-fasttext	0	0	0	0	0	0	0
video-processing	0	0	0	0	0	0	0

H MuSiQue Pipeline Artifact Configuration (Before/After)

The MuSiQue retrieval-pipeline artifact configuration is a JSON DAG of nodes, each with a type, per-node parameters, and (for the rewrite node) a prompt file. MARA CHAIN evolves this directory.

Default (baseline).

```
recall_bm25(k=10) -> recall_dense(k=10, bge_m3) ->
rerank_cross(top_m=100, bge-reranker-v2-m3) -> graph_rescore(a=0.3) ->
truncate(k=10)
```

Evolved.

```
query_rewrite(mode=llm, GLM-5.2, prompt_file=rewrite, T=0, max_tokens=128) ->
recall_bm25(k=30, use_rewritten=true) ->
recall_dense(k=30, bge_m3, use_rewritten=true, use_dual_query=true) ->
fuse_rrf(rrf_k=60) ->
```

```
graph_expand(top_n=40, min_entity_overlap=1, max_expand_per_group=10) ->
rerank_cross(top_m=30, bge-reranker-v2-m3, use_rewritten=false) ->
graph_rescore(a=0.2) -> truncate(k=30)
```

The proposer edited three kinds of assets: *node code* (added `query_rewrite`, `fuse_rrf` (Cormack et al., 2009), `graph_expand`; modified recall k 10 $\rightarrow$ 30, rerank `top_m` 100 $\rightarrow$ 30, `graph_rescore` α 0.3 $\rightarrow$ 0.2; rewired the DAG), a *prompt* (the rewrite node's synonym/term-expansion prompt), and *hyperparameters* (RRF `rrf_k`= 60, `graph_expand top_n`= 40 / `max_expand_per_group`= 10).

Ablation: the chain's contribution on MuSiQue. The ablation in Table 4 (App. A) isolates the Mara Chain's contribution on this retrieval-pipeline artifact configuration. At the same 2-slot budget, disabling the chain (−Mara Chain: single-shot search, Mara-chain depth 0) still improves over the hand-written default (+0.070 test nDCG@10), but trails the full setting by 0.034 (test) and 0.048 (val) nDCG@10 and by 0.039 (test) and 0.045 (val) Recall@10. The chain therefore accounts for roughly a third of the total test gain, mirroring the ablation pattern on AppWorld and TerminalBench 2.1.

I Analogy to Classical Optimization

The framework main loop of is a recognizable instance of classical iterative optimization, with the artifact configuration in place of the parameter vector and the Mara Chain in place of *momentum*—an operational, not decorative, analogy. A classical optimizer carries a correction term across steps so the next update builds on the last rather than restarting from zero, and the Mara Chain plays exactly this role: accumulating diagnoses and edits down a single lineage so the next attempt starts from the residual the previous one left, instead of discarding it the way single-shot search does (Thms. 1–2). What is *not* classical is the substrate: the "gradient" is a rollout-derived edit attributed to a traced failure, and the "correction term" is a diagnosis in natural language and code, carried on a filesystem—which is why a coding agent, not a fixed update rule, is the optimizer.

Table 7: The evolutionary-optimization analogy is *operational rather than decorative*: each row has a functional equivalent in the loop.

Known concept	framework functional equivalent
parameter	artifact configuration (artifact directory tree)
search space	schema-instantiated artifact tree (not a fixed DSL)
gradient direction	rollout-derived edit direction (`ArtifactAction`)
learning rate	edit budget per epoch
optimizer	reflective proposer (two-phase coding agent)
momentum	Mara Chain (correction accumulated across a chain of failures)
early stopping / checkpoint	candidate state machine + $(epoch, dataset_index)$ resume

J Persistent Failure Barrier Case Studies (Full)

This appendix presents four illustrative traces relevant to §J.1; they do not establish the assumptions of Thms. 1–2. Because tasks run in batches (`batch_size`= 4), a propose that earns reward on one task may be accepted even if another task in the batch still scores 0, so the other task's partial fix is *incidentally merged* into the lineage on someone else's reward. Such merges are unstable—they occur by coattail, not because the failing task was targeted—and fall far short of the targeted re-diagnosis that resolves the persistent failure barriers below.

J.1 Persistent failure barrier case studies

The recorded traces contain four depth-4 passing descendants after zero-scoring ancestors; they are illustrative, not causal identification. On TerminalBench 2.1, the recorded −Mara Chain configuration does not pass the listed tasks while a recorded retained lineage does. Each listed lineage has zero-scoring ancestors and a later passing descendant. We trace

four cases (App. J for full traces): three TerminalBench persistent failure barriers resolved at depth 4—`sanitize-git-repo`, `model-extraction-relu-logits` (Table 8), and `protein-assembly` (Table 9)—and one AppWorld scenario, `29caf6f` (Table 10, Fig. 9). In all four, every depth-< 4 ancestor scores 0 and only the depth-4 child passes; in three the breakthrough asset is general and family-reusable (`protein-assembly` is cleared by a protein-specific rule). On the AppWorld persistent failure barrier, +Mara Chain is stable and −Mara Chain unstable (Fig. 9, App. J). The following illustration traces one refinement lineage end to end; the other three persistent residuals are traced below.

J.2 Without Mara Chain: Persistent Residuals

Theorem 1 applies only under its residual-completeness and accepted-update assumptions. The recorded −Mara Chain run retires 78 of 104 candidates, including candidates with partial corrections.

`sanitize-git-repo`. More than ten propose batches analyze this persistent failure barrier, each adding another partial prose fix (a length-constrained-pattern rule, a no-truncation rule, a `grep -roh` procedure, a repo-wide-search rule, a git-checkout caveat). Several are re-made—the same diagnosis recurs in three separate batches. Yet every sanitize run scores 0 (9 candidates). The partials either merge for the wrong reason (other tasks in the batch pass while sanitize stays 0) or are discarded; in both cases the partial stays at prose (never raised to a runnable script), so the buried token is never surfaced. The partial correction is neither refined on the same evidence nor transferred to a higher-reliability script, so the residual persists.

`model-extraction-relu-logits`. Four propose entries each re-add essentially the same partial—a prose "black-box extraction" rule. The partial is always prose (no runnable extraction script), so all 9 runs score 0. Independent proposals repeatedly recreate the same prose partial, but do not retain its evidence long enough to construct the extraction algorithm the task requires.

`protein-assembly`. 8 candidates, all score 0 on `test_gblock`—the agent uses HIV-1 capsid instead of the FLAG tag, or strips tags from the specified source. −Mara Chain does diagnose specific errors ("HIV-1 capsid instead of FLAG tag", "no gBlock reference"), but these candidates are retired; none builds the full correction chain (use the specified source verbatim, handle X-residues, verify the first amino acid, check the fusion order). Because the candidates are retired independently, these partial corrections are not accumulated into a complete sequence.

AppWorld `29caf6f`. −Mara Chain solves only 0.16/0.21/0.21 of runs across the scenario's three tasks, every failure tripping the *same* sub-condition—"the reply names a non-requested movie"—reproduced verbatim $54\times$. The skill artifact configuration already tells the agent to extract the requested director and filter by it, but the rule is a bypassable prose/regex asset ($d < \theta$); the agent discards the director and sends all 21 movies $54\times$. The extract-and-filter rule is a useful but incomplete low-reliability artifact; its execution evidence is discarded rather than used to refine the candidate or strengthen the artifact representation.

J.3 With Mara Chain: convergence

The Mara Chain restructures search as a single lineage $\kappa = (\sigma_0, \ldots, \sigma_T)$. Each step conditions on the preceding residual, rollout analysis, and artifact-change history; it may also select a higher-reliability artifact representation when the evidence indicates bypass. Newly observable failures become the residual for the next refinement step. Theorem 2 gives a bound only when its positive-support assumption is made. In these recorded traces, four passing descendants occur at depth 4 after zero-scoring ancestors. The illustration gives the `sanitize-git-repo` workflow, Tables 8– 9 give the remaining TerminalBench traces, and Table 10 gives the AppWorld scenario.

`sanitize-git-repo`: a 4-step refinement builds the script. Each node conditions on the preceding residual; the artifact changes from prose to script; and the depth-4 "print only the matched token" edit corrects the script's own output behavior, which became observable only after the script

was introduced. Retaining each residual allows the three failure modes to be addressed sequentially rather than rediscovered independently.

`model-extraction`: a 4-deep bloodline rewrites the algorithm (Table 8). Winner score 1.0, 20/20 rows match, cosine > 0.99; every ancestor scores 0. The artifact evolves from prose to reference material and then to a script. The two script fixes address distinct retained residuals—"too slow" (execution) and "over-counts" (algorithmic correctness)—and the final rewrite corrects behavior observable only after a runnable script exists.

Table 8: Model extraction: one bloodline, depth $0 \rightarrow 4$, every ancestor scoring 0. The artifact evolves from prose to reference material and then to a script; the depth-3/4 algorithm rewrite corrects behavior exposed only after the script is runnable.

rd	candidate	form	asset edited	residual δ (drives next)
0	`f95e6890cfa1`	(start)	(inherits parent, no task-specific edit)	agent `cats forward.py` (not black-box)
0	`dd5d0e8a3382`	prose+ref	`SKILL` black-box rule + extraction ref	evaluator `np.random.seed` pollution
1	`38899c80b3fe`	ext+ref	random-state isolation	correct method, no runnable impl
2	`fe8a3fbab6c9`	ref+scr	`extract-relu-weights.py` v0 (freq. clustering)	too slow; agent abandons
3	`d87d45d2e963`	ref+scr	efficiency: 2000 $\rightarrow$ 800 samples	over-counts: 47 vs 20 neurons
4	`38b2f87817e4` ✓	ext+ref+scr	algorithm rewrite: line-scan	**PASS**

`protein-assembly`: a 4-deep bloodline corrects the fusion sequence (Table 9). It inherits the model-extraction winner's artifact configuration, then goes its own depth-4 route on the protein persistent failure barrier. The barrier is resolved by a *protein-specific* X-residue rule—the one genuinely per-task patch—though the same winning node also banks a general cross-compiler that serves other tasks.

Table 9: Protein assembly: one bloodline, depth $0 \rightarrow 4$, every ancestor scoring 0. † marks a protein-specific fix and $^{\text{gen}}$ a general (reusable) fix.

rd	candidate	form	asset edited	residual δ (drives next)
0	`899d9d350389`	ext+prose	FC fix in `kira_agent`$^{\text{gen}}$	uses HIV-1 capsid instead of FLAG tag
0	`52aa7b868cdc`	ext+prose	terminal spirals in `kira_agent`$^{\text{gen}}$	wrong source, strips tags from data
1	`51e925ee9e77`	ext+prose	data-source checklist in `kira_agent`† + SKILL rule†	assembly still wrong
2	`216373b50662`	ext+prose	verbatim / His-tag rule in SKILL† + runtime mod$^{\text{gen}}$	X-residue handling wrong
3	`c0f841ac9754`	ext+prose+scr	data-source rule in SKILL† + tool alias$^{\text{gen}}$ + sanitizer$^{\text{gen}}$	length offset, first amino acid
4	`fddaf3b3e11c` ✓	ext+prose+scr	X-residue rule in SKILL† + cross-compiler$^{\text{gen}}$	**PASS**

AppWorld `29caf6f`: the chain hardens the scenario's shared `SKILL.md` (Table 10, Fig. 9). Early fixes (gen 3–7) already make +Mara Chain's _2/_3 pass rate far exceed −Mara Chain's 0.21 (0.60–0.75 at gen 2–3), while −Mara Chain stays at 0.21 throughout—the early fixes, not just the late chain, are what makes +Mara Chain stable. The late chain (gen 15–16) adds the requester-retrieval gate + director isolation (peak val 0.855). Because the three tasks share one `SKILL.md`, the hardening fixes _2/_3 *and* stabilizes _1. Figure 9 shows the pass/fail pattern: +Mara Chain is stable, −Mara Chain unstable.

Generality. Three of the four breakthrough assets are general and family-reusable: `search-filtered.sh` takes any prefix set and prints only matched tokens;

Table 10: AppWorld `29caf6f`: key RC fixes on the shared `SKILL.md` that harden the scenario, in generational order. Early fixes (gen 3–7) already make WITH's pass rate far exceed −Mara Chain's 0.21; the late chain (gen 15–16) adds the requester-retrieval gate + director isolation.

gen	rd	node	edit	solved
3	1	`39102e8295c9`	fix login rule, pagination order, substring collisions	_3
4	1	`ec65a28a8e6e`	step-6 banking: paginate all contacts + client-side filter (prerequisite; seeds checkpoint)	locates msg
4	2	`f66bfcd1ab39`	prevent context overflow from exhaustive message pagination	_2, _3
7	0	`9ccdf0fbfefe`	add `pagination_helper.py` + prevent restart loops (+Mara Chain-only checkpoint)	sustains
14	1	`a810bbf06036`	fix fragile parsing (line-by-line extraction)	_2, _3
15	1	`6ebfc3cbb94b`	enforce requester-message retrieval gate (root cause: agent never retrieved the request)	_2
16	2	`35a92e7e209d` ✓	isolate requested director in a variable; filter by it (peak val 0.855)	all three

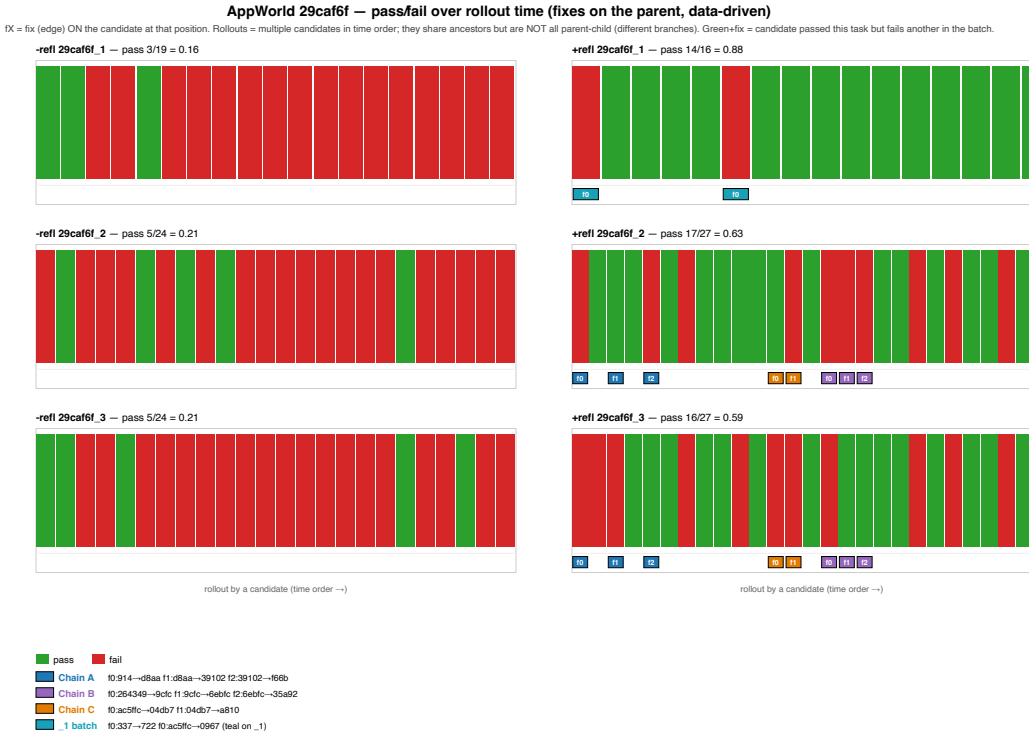


Figure 9: AppWorld `29caf6f`: pass/fail of each task's in-run rollouts during the ±Mara Chain ablation (green=pass, red=fail; left-to-right in rollout time). −Mara Chain (left) is unstable; +Mara Chain (right) is stable. Colored squares mark fixes (first rollout only); same color = same chain. $f0/f1/f2$ = a fix producing an rd-$0/1/2$ candidate.

`extract-relu-weights.py` auto-detects input dim and scans random 1-D lines for any black-box ReLU net; the AppWorld extract-and-filter serves any "reply with items matching a requested attribute" task. `protein-assembly` is the exception (a protein-specific rule).

Four persistent residuals, one mechanism. Under the assumptions of Theorem 1, unsupported accepted updates preserve a residual. These traces show retained rollout evidence and artifact history across later edits, but do not establish that retention alone caused the observed passes or that budget played no role.