LIFETIME LEARNING OF SKILLS: EVOLVING THE SKILL-LEARNING ALGORITHM ITSELF

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model agents can acquire reusable procedural knowledge as skills. Existing skill-learning methods typically use fixed refinement lifecycles that prescribe which execution evidence to inspect, how to revise skills, how to allocate the evaluation budget, and when to deploy or revert updates. These methods apply the same learning strategy across heterogeneous failures, domains, and stages of skill maturity. We introduce Lifetime Skill-Learning (LSL), a framework that evolves the algorithm for constructing, evaluating, and maintaining skills while leaving the foundation model and host agent unchanged. LSL represents the skill-learning lifecycle as an executable meta-program that controls trajectory selection, failure diagnosis, update-operator selection, candidate generation, budget allocation, testing, compression, deployment, and rollback. An inner loop executes each meta-program to construct and validate domain skills. An outer population-based loop evolves the meta-programs using cross-task regret, anytime performance, budget efficiency, transfer, lifecycle complexity, and safety-constraint violations. This separation allows the learning strategy to adapt and records an executable, replayable history of how each deployed skill was obtained. We evaluate LSL using a cross-domain meta-learning protocol with structurally diverse domains for meta-training and unseen domains for transfer evaluation. We compare LSL with fixed refinement loops, expert-designed policies, single-prompt meta-skills, and population-based alternatives. Controlled component ablations further test whether lifecycle evolution provides benefits beyond additional sampling and repeated skill rewriting.

1 INTRODUCTION

Large language model (LLM) agents increasingly operate in environments that require procedural knowledge beyond general linguistic and reasoning capabilities. Reliable execution depends on using specific interfaces, satisfying domain constraints, recovering from recurring failures, and verifying task completion. Agent skills encode this knowledge as reusable packages of instructions, scripts, examples, and resources that are loaded at inference time without modifying the underlying model. Curated skills improve performance on diverse, expertise-intensive tasks, although the gains vary across tasks, models, and agent harnesses.(Li et al., 2026) Because skills are explicit and editable artifacts, they also support persistent improvement: agents can collect execution experience, revise and validate skills, and reuse them in subsequent interactions.

Existing approaches automate skill improvement using a predefined learning algorithm. SkillEvolver implements skill learning as a meta-skill that authors, deploys, audits, and iteratively refines domain skills from execution trajectories.(Zhang et al., 2026) Controlled studies characterize multi-round skill evolution as a sparse, validation-filtered search process whose returns depend on the model, benchmark, feedback composition, and evolution round.(Liu et al., 2026) Related work extends the search space to prompts or complete agents. Promptbreeder co-evolves task prompts and mutation prompts,(Fernando et al., 2024) while automated design methods generate agent programs from archives of previous designs.(Hu et al., 2025) The Darwin Gödel Machine allows agents to modify their code and retain empirically useful descendants through open-ended search.(Zhang et al., 2025a) These methods optimize persistent artifacts and their update mechanisms, but typically specify the skill-learning lifecycle manually. Decisions about which trajectories to inspect, how to diagnose

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

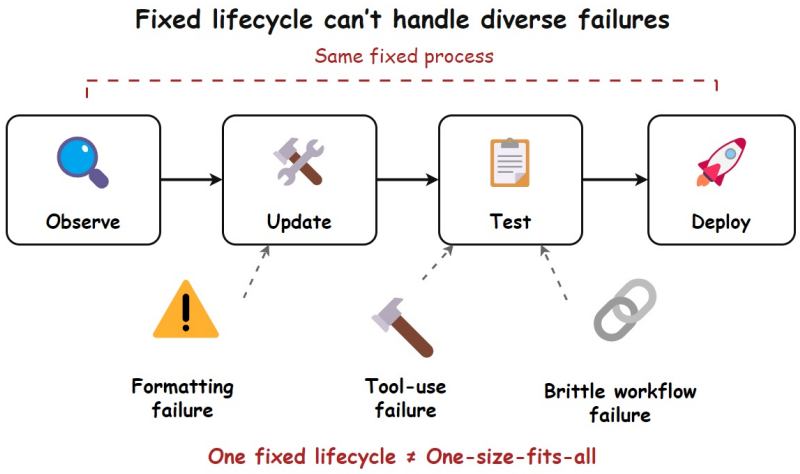


Figure 1: Mismatch between a fixed skill-learning lifecycle and heterogeneous failure modes.

failures, which mutation operator to apply, how many candidates to generate, how to allocate the evaluation budget, and when to stop or roll back are fixed before learning.

A fixed lifecycle cannot address the heterogeneity of skill-learning problems, as illustrated in Figure 1. Formatting failures may require localized constraint patches, incorrect tool-use strategies may require structural replacement, and brittle workflows may require additional tests. Successful traces reveal reusable procedures, while failures expose missing preconditions or unsafe actions. The value of each type of evidence varies across domains and learning stages. Aggressive exploration can support early-stage search but wastes resources after a stable skill emerges. Revisions that improve training performance can also reduce transfer performance or violate deployment constraints. Adapting the skill alone therefore leaves a second-order problem: adapting *how skills are learned*. The learning process must select evidence and update operators based on observed failures, allocate a bounded evaluation budget, preserve validated behavior, and support auditing and rollback.

We introduce **Lifetime Skill-Learning** (**LSL**), which evolves the skill-learning algorithm while keeping the foundation model fixed. LSL represents a lifecycle as an executable meta-program that performs observation, diagnosis, operator selection, budget allocation, candidate generation, testing, selection, compression, deployment, and rollback. Given task executions and outcomes, the meta-program selects successful and failed trajectories, update operators, candidate counts, test environments, and termination or rollback conditions. An inner loop executes the meta-program to produce and validate domain skills. An outer population-based program-evolution loop searches over meta-programs using fitness criteria that capture cross-task regret, budget efficiency, transfer performance, program complexity, and safety-constraint violations. The evolution process therefore targets the algorithm for constructing, evaluating, and maintaining skills, rather than a single skill. LSL modifies neither the model weights nor the host agent implementation. Its decisions and artifacts remain executable and replayable, allowing each deployed skill to be audited and traced to the process that produced it.

We evaluate LSL through cross-domain meta-learning. Structurally diverse domains are used for meta-training, and unseen domains evaluate lifecycle transfer. We compare LSL with fixed refinement loops, manually designed expert policies, a single-prompt meta-skill, Population-Based Training,(Jaderberg et al., 2017) and program-evolution variants. In addition to final task performance, we measure anytime performance, cross-task regret, evaluation-budget efficiency, transfer, lifecycle complexity, and safety-constraint violations. Controlled ablations fix the update operator or budget, remove rollback, exclude successful trajectories, or restrict meta-training to a single domain. These experiments determine whether lifecycle evolution provides gains beyond additional sampling, a single iterative rewrite chain, or conventional population-level selection over fixed optimizers.

Our contributions are:

- We formulate skill self-improvement as lifetime meta-learning that explicitly optimizes the learning lifecycle and the resulting skill.
- We propose LSL, an auditable two-level framework that evolves executable meta-programs for selecting evidence, choosing skill-update operators, allocating evaluation budgets, validating candidates, and controlling deployment and rollback.
- We develop a cross-domain evaluation protocol that measures transfer, anytime performance, budget efficiency, lifecycle complexity, and safety through controlled comparisons and component-level ablations.

2 Related Work

2.1 Experience, Memory, and Reusable Skills for Language Agents

Language agents can retain information outside frozen foundation models. Reflexion converts feedback into verbal reflections,(Shinn et al., 2023) ExpeL extracts reusable insights from successful and failed trajectories,(Zhao et al., 2023) Generative Agents organize observations and higher-level reflections,(Park et al., 2023) and MemGPT manages multiple memory tiers through explicit control operations.(Packer et al., 2023) These approaches primarily preserve experience as declarative context.

Procedural systems instead encode reusable behavior. Voyager maintains executable code skills,(Wang et al., 2024) SkillWeaver discovers and refines web-agent skills,(Zheng et al., 2025) Skill-Pro represents skills through activation, execution, and termination conditions,(Mi et al., 2026) and SkillsBench evaluates curated skill packages across heterogeneous tasks.(Li et al., 2026) Self-Refine supplies iterative critique and revision,(Madaan et al., 2023) while ReAct,(Yao et al., 2023b) Tree of Thoughts,(Yao et al., 2023a) and Graph of Thoughts(Besta et al., 2023) expand within-task reasoning through interaction or structured search. SkillEvolver closes the persistence loop by authoring, auditing, and revising skills from trajectories,(Zhang et al., 2026) while multi-round analyses characterize evolution as validation-filtered search with model-, benchmark-, feedback-, and round-dependent returns.(Liu et al., 2026)

2.2 Automatic Optimization of Prompts and Agentic Programs

Automatic Prompt Engineer generates and selects instructions,(Zhou et al., 2022) OPRO proposes candidates from prior solutions and scores,(Yang et al., 2023) ProTeGi applies critique-guided edits and candidate selection,(Pryzant et al., 2023) and TextGrad propagates textual feedback through computation graphs.(Yuksekgonul et al., 2024) Evolutionary variants use semantic genetic operators(Guo et al., 2023) or co-evolve task and mutation prompts.(Fernando et al., 2024) Structured approaches optimize modular programs,(Khattab et al., 2024) semantic prompt sections,(Juneja et al., 2025) or executable prompting and tool-use configurations.(Spiess et al., 2025)

Broader methods search over complete workflows. ADAS generates code-represented agents from an evaluated archive,(Hu et al., 2025) and AFlow searches workflow graphs using tree search and execution feedback.(Zhang et al., 2025b) The Darwin Gödel Machine permits code-level self-modification and archival selection,(Zhang et al., 2025a) extending the Gödel machine's proof-authorized self-rewriting formulation to empirical evaluation.(Schmidhuber, 2003) Increasing expressiveness enlarges the design space but raises evaluation costs and complicates behavioral attribution.

2.3 Meta-Learning, Learned Optimization, and Population-Based Search

MAML learns parameters suitable for rapid gradient adaptation,(Finn et al., 2017) whereas recurrent meta-learners encode adaptation in hidden state.(Duan et al., 2016)(Wang et al., 2016) Learned optimizers directly produce parameter updates,(Andrychowicz et al., 2016) scale meta-training across diverse optimizees,(Metz et al., 2022) or optimize learned loss functions.(Shu et al., 2019)

Resource-aware black-box methods include Hyperband,(Li et al., 2016) BOHB,(Falkner et al., 2018) and Population-Based Training.(Jaderberg et al., 2017) Evolutionary approaches preserve behavioral diversity through MAP-Elites,(Mouret & Clune, 2015) novelty search,(Lehman & Stanley, 2011) quality-diversity optimization,(Zolla et al., 2018) or co-evolving environments and agents.(Wang et al.,

2019) Continual-learning methods address persistence through parameter constraints,(Kirkpatrick et al., 2017) experience replay,(Rolnick et al., 2018) or constraints derived from stored examples.(Lopez-Paz & Ranzato, 2017) Across these families, adaptation depends on predefined update mechanisms, evaluation budgets, mutation operators, retention policies, and stopping criteria.

3 METHOD

We propose Lifetime Skill-Learning (LSL), a bilevel framework that evolves task-specific skills and the executable algorithms that learn them while keeping the language-model parameters frozen. The outer loop searches over auditable lifecycle programs, and the inner loop executes each program to generate, evaluate, compress, deploy, and roll back versioned skills.

3.1 PROBLEM FORMULATION AND FRAMEWORK OVERVIEW

Let $\mathcal{D}$ denote a distribution over domains, and let $\mathcal{T}_d = \{\tau_i\}_{i=1}^{N_d}$ denote a collection of tasks from $d \sim \mathcal{D}$. Each task $\tau = (x, \mathcal{E}, V)$ comprises an instruction x, an execution environment $\mathcal{E}$, and a verifier V. Given a skill s, a frozen agent with parameters θ produces $z \sim \mathrm{Agent}_\theta(x, \mathcal{E}; s)$ with reward $r = V(z) \in \mathbb{R}$, where z records actions, tool calls, observations, and the final output. A skill is a versioned inference-time artifact that contains procedural instructions and optional executable resources. It affects agent behavior through context and tools without updating θ.

The objective is to learn a lifecycle program π that maps task experience and an existing skill repository to an improved repository under a bounded execution budget. The program controls evidence selection, failure diagnosis, editing, evaluation allocation, acceptance, and rollback. During meta-training on $\mathcal{D}_{\mathrm{tr}}$, task-level verifiers guide lifecycle search without requiring gradient access to Agent_θ. The learned lifecycle is evaluated for transfer to unseen domains $\mathcal{D}_{\mathrm{te}}$.

Figure 2 summarizes LSL. The inner loop receives tasks, a current skill version, and an execution budget. It collects traces, diagnoses skill deficiencies, generates and evaluates candidates in isolated sessions, and uses validation outcomes to select, compress, deploy, or roll back skills. The inner loop returns a versioned artifact and a replayable provenance record. The outer loop evaluates a population of lifecycle programs across domains according to task reward, resource use, transfer, safety, and complexity, and then evolves their control logic and operator choices.

3.2 EXECUTABLE LIFECYCLE PROGRAMS

At inner-loop step t, a lifecycle operates on

$$h_t = (\mathcal{T}_t^{\mathrm{seen}}, \mathcal{Z}_t, \mathcal{S}_t, b_t, q_t), \qquad a_t = \pi(h_t) \in \mathcal{A}_{\mathrm{life}}, \tag{1}$$

where $\mathcal{T}_t^{\mathrm{seen}}$ contains the encountered tasks, $\mathcal{Z}_t$ is the trace buffer, $\mathcal{S}_t$ is the versioned repository, b_t is the remaining budget, and q_t records the audit and safety status. The action space covers observation, diagnosis, editing, evaluation, selection, compression, deployment, rollback, and termination. This definition also specifies the lifecycle policy. The state evolves according to $(h_{t+1}, o_t, c_t) = \mathrm{Transition}(h_t, a_t; \mathrm{Agent}_\theta)$, where o_t is the observation and $c_t \geq 0$ is the consumed resource. Programs support typed branches, bounded loops, and predefined operators. They cannot update θ or bypass verification, versioning, or resource accounting.

Each execution emits a provenance graph $\mathcal{G}_\pi$ that links states, traces, diagnoses, skill versions, evaluations, and deployment decisions. Deployment requires an identified parent, a reproducible edit record, and an admissible validation record:

$$\mathrm{Valid}(\pi, \mathcal{G}_\pi) = \mathbb{I}[\mathcal{G}_\pi \text{ satisfies provenance, budget, and safety rules}]\,. \tag{2}$$

Executions that violate these requirements receive infeasible outer fitness and cannot deploy skills.

3.3 INNER-LOOP SKILL EVOLUTION

Given the current skill s_t, the lifecycle selects tasks and collects scored trajectories $\mathcal{Z}_t = \{(z_{t,j}, r_{t,j})\}_{j=1}^{n_t}$, where $z_{t,j} \sim \mathrm{Agent}_\theta(x_j, \mathcal{E}_j; s_t)$ and $r_{t,j} = V_j(z_{t,j})$. A diagnosis operator contrasts successful and failed traces to produce $\delta_t = g_t(\mathcal{Z}_t, s_t) = (e_t^{\mathrm{miss}}, e_t^{\mathrm{wrong}}, e_t^{\mathrm{unused}}, e_t^{\mathrm{risk}})$, which

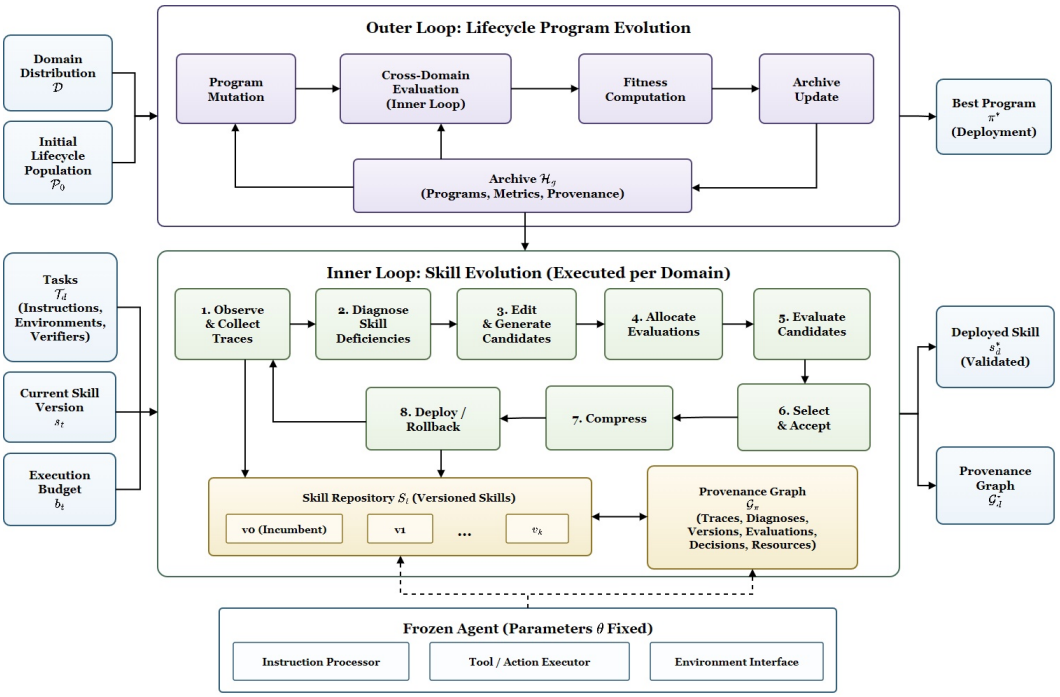


Figure 2: Overview of the LSL framework and its outer and inner evolution loops.

represents missing procedures, misleading instructions, unused components, and safety risks. The edit operators condition on this diagnosis record to constrain skill revision.

The lifecycle selects $u_t \in \mathcal{U}$ and generates $\mathcal{C}_t = u_t(s_t, \delta_t; k_t) = \{\tilde{s}_{t,1}, \ldots, \tilde{s}_{t,k_t}\}$, where the remaining budget constrains k_t. The operator library includes patching, restructuring, check insertion, resource revision, and compression. Evaluation allocations satisfy $\sum_j \beta_{t,j} \leq b_t$. In isolated agent sessions, candidate j receives

$$\widehat{R}_{t,j} = \frac{1}{|\mathcal{V}_{t,j}|} \sum_{\tau_i \in \mathcal{V}_{t,j}} V_i(\mathrm{Agent}_\theta(x_i, \mathcal{E}_i; \tilde{s}_{t,j})) \,. \tag{3}$$

The lifecycle can apply low-fidelity screening before higher-fidelity evaluation, with all costs recorded. Among the candidates satisfying $\mathrm{Safe}(\tilde{s}_{t,j}) = 1$, the highest-reward candidate replaces the incumbent only when $\mathrm{Accept}(\widehat{R}_{t,j}, \widehat{R}_{t,0}) = 1$. Otherwise, $s_{t+1} = s_t$, which implements the rollback rule.

Compression uses the same validity and acceptance interface. At stopping time T, the lifecycle deploys the validated repository skill with the highest estimated reward, $\mathcal{S}^\star_{\pi,d} = \mathrm{Deploy}\left(\arg\max_{s \in \mathcal{S}_T} \widehat{R}(s)\right)$. The lifecycle terminates when the budget is exhausted, no admissible action remains, or the program's stopping condition is satisfied.

3.4 Outer-Loop Evolution and Objective

At generation g, the outer loop maintains a population $\mathcal{P}_g$ and an archive $\mathcal{H}_g$. Each archive entry stores a program, its lineage, domain-level metrics, resource use, validity, and the provenance of its generated skills. The outer loop samples a parent π_p and mutates it into $\pi' = m(\pi_p; \xi)$. The mutations modify operator calls, branches, budget rules, stopping conditions, or rollback conditions. Static type checks and bounded-execution checks precede empirical evaluation.

Each valid descendant executes its complete inner loop on sampled meta-training domains under a common budget. For domain d, the outcome $\mathbf{y}_{\pi,d} = (R^{\mathrm{final}}_{\pi,d}, G^{\mathrm{any}}_{\pi,d}, C_{\pi,d}, U_{\pi,d}, K_\pi)$ contains the final reward, anytime performance, resource cost, safety-violation rate, and program complexity.

Table 1: Success rate (%, mean±std over five seeds) under matched inner-task execution budgets. The upper panel reports meta-training domains; the lower panel reports meta-test domains. Best results are **bold**; second-best results are underlined.

Method	Sheet	Document	Code	Planning	Train Average
FIXEDLOOP	54.3±1.9	57.8±2.1	39.6±1.7	48.1±2.4	50.0
EXPERTPOLICY	59.1±1.6	63.0±1.8	42.7±1.5	52.6±2.0	54.4
METAPROMPT	56.8±2.5	60.2±2.7	40.9±2.1	50.7±2.9	52.2
GREEDYPROGRAM	61.0±1.8	64.1±1.6	44.0±1.8	53.4±2.2	55.6
PBT	62.4±1.5	65.8±1.5	45.3±1.4	55.1±1.9	57.2
LIFETIME	**66.1±1.3**	**69.4±1.2**	**49.8±1.3**	**59.0±1.7**	**61.1**
Method	Science	Web	Kernel	Test Average	
FIXEDLOOP	52.4±2.2	55.7±1.8	40.8±2.5	49.6	
EXPERTPOLICY	56.8±1.9	59.6±1.7	42.1±2.3	52.8	
METAPROMPT	54.5±2.8	57.3±2.4	41.7±3.0	51.2	
GREEDYPROGRAM	57.7±2.0	60.8±1.9	42.9±2.4	53.8	
PBT	59.4±1.7	62.1±1.6	40.8±2.1	54.1	
LIFETIME	**63.5±1.5**	**66.8±1.4**	**45.7±1.9**	**58.7**	

Table 2: Ablations on unseen meta-test domains (%, mean±std over five seeds).

Variant	Science	Web	Kernel	Average
Fixed budget	61.2±1.7	64.0±1.6	43.5±2.2	56.2
Fixed operator	59.9±1.8	62.8±1.7	42.0±2.3	54.9
No rollback	60.5±1.9	63.4±1.8	39.8±2.8	54.6
No success traces	61.7±1.8	64.5±1.5	42.7±2.4	56.3
Single-domain meta-train	58.8±2.3	61.7±2.0	40.9±2.7	53.8
LIFETIME	**63.5±1.5**	**66.8±1.4**	**45.7±1.9**	**58.7**

Archive updates retain programs that improve fitness or increase structural and behavioral diversity:

$$\mathcal{H}_{g+1} = \mathrm{ArchiveUpdate}(\mathcal{H}_g, \pi', \mathbf{y}_{\pi'}), \qquad \mathcal{P}_{g+1} \sim Q_{g+1}(\mathcal{H}_{g+1}). \tag{4}$$

Given a reference reward R_d^{ref}, the cross-task regret is $\mathrm{Regret}(\pi) = \mathbb{E}_{d\sim\mathcal{D}_{\mathrm{tr}}}[R_d^{\mathrm{ref}} - R_{\pi,d}^{\mathrm{final}}]$. The outer fitness is $F(\pi) = -\mathrm{Regret}(\pi) + \lambda_{\mathrm{any}}\,\mathrm{Anytime}(\pi) - \lambda_{\mathrm{cost}}\,\mathrm{Cost}(\pi) - \lambda_{\mathrm{transfer}}\,\mathrm{Transfer}(\pi) - \lambda_{\mathrm{safe}}\,\mathrm{Violation}(\pi) - \lambda_{\mathrm{complex}}\,\mathrm{Complexity}(\pi)$, where all coefficients are nonnegative and fixed during final evaluation. Transfer is measured on held-out tasks or domains, and Complexity counts executable nodes and control-flow edges. The deployment program is selected by $\pi^\star \in \arg\max_{\pi\in\mathcal{H}_G} F(\pi)$ subject to $\mathrm{Valid}(\pi, \mathcal{G}_\pi) = 1$, $\mathrm{Cost}(\pi) \le B_{\max}$, $\mathrm{Violation}(\pi) \le \epsilon_{\mathrm{safe}}$. These constraints quantify resource and safety trade-offs within the evaluated environments. They provide no guarantee for behavior outside those environments.

3.5 TRAINING AND DEPLOYMENT

Training repeatedly samples and mutates a lifecycle, executes it across sampled domains, validates provenance and safety, computes F, and updates the archive. No gradients pass through agent executions, verifiers, or mutations. The appendix provides detailed pseudocode.

At meta-test time, $\pi^\star$ is frozen and executed on a new domain with a fresh state and a prescribed budget, yielding $(s_d^\star, \mathcal{G}_d^\star) = \mathrm{RunLifecycle}(\pi^\star, \mathrm{Agent}_\theta, \mathcal{T}_d, B_{\max})$, where $s_d^\star$ is the deployed skill and $\mathcal{G}_d^\star$ is its replayable provenance graph. The lifecycle revises domain skills using task feedback while keeping its control logic and the base-model parameters fixed during deployment. % Revised LaTeX code

Table 3: Efficiency under the common meta-training budget. Quality is held-out success; utility is quality divided by thousands of model calls.

Method	Quality	Calls	GPU-hours	Utility
FIXEDLOOP	49.6	10.8k	41	**4.59**
METAPROMPT	51.2	11.6k	44	4.41
EXPERTPOLICY	52.8	12.1k	47	4.36
PBT	54.1	15.9k	63	3.40
LIFETIME	**58.7**	13.8k	57	4.25

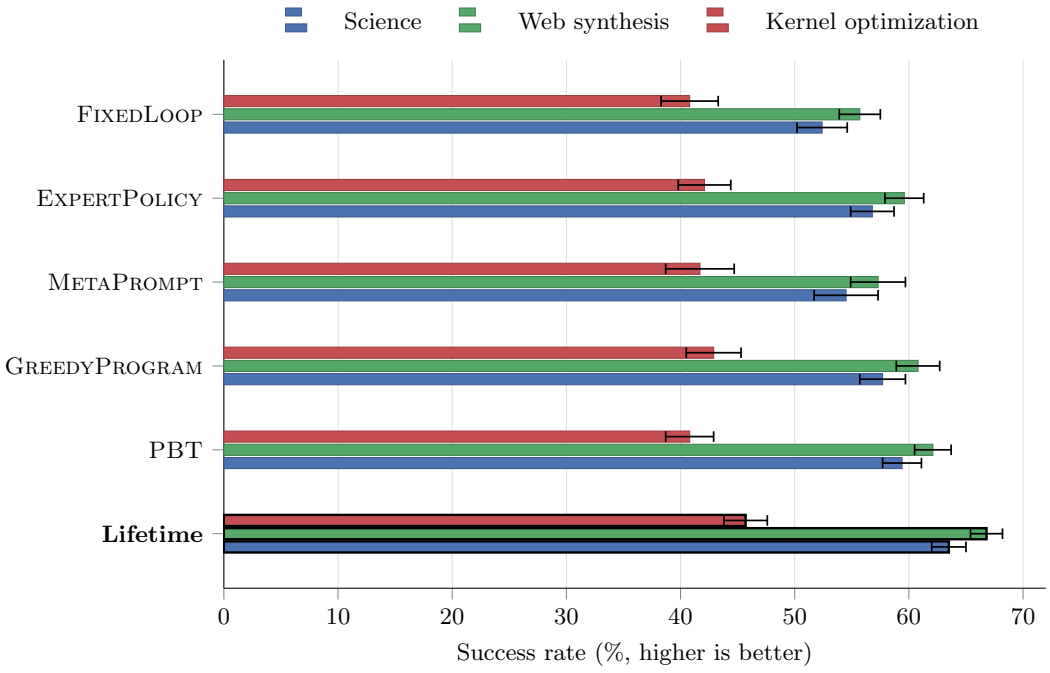


Figure 3: Meta-test success rates under matched inner-task execution budgets. Bars show means over five seeds and error bars show one standard deviation.

4 EXPERIMENTS

4.1 PROTOCOL

We evaluate whether evolving the skill-learning lifecycle improves learned skills under matched execution budgets and transfers to unseen domains. Meta-training covers four structurally distinct domains: spreadsheet manipulation, document transformation, repository-level software maintenance, and constrained planning. We reserve scientific data analysis, web-based information synthesis, and GPU-kernel optimization for meta-testing. Each domain contains 20 task families, with five development instances and ten disjoint evaluation instances per family. We use executable or deterministic verifiers when available and normalize continuous rewards to $[0, 1]$ using domain-specific reference bounds.

To separate lifecycle quality from domain exposure, development instances are used only for outer-loop selection and adaptation decisions, whereas evaluation instances remain inaccessible until scoring. Task-family splits prevent near-duplicate templates from appearing across these partitions. We preserve the native interaction horizon of each domain, so a lifecycle must allocate its budget across observation, revision, and verification without relying on artificially shortened episodes.

We compare LIFETIME with five alternatives. FIXEDLOOP follows a manually defined observe–revise–test procedure with fixed candidate allocation and termination. EXPERTPOLICY uses a domain-agnostic lifecycle designed by experienced agent developers. METAPROMPT uses a single language-model prompt to select each learning action without maintaining an executable controller. PBT tunes the hyperparameters of a fixed lifecycle through population-based training. GREEDYPROGRAM mutates only the current best controller. LIFETIME evolves an archive of executable meta-programs that can modify evidence selection, diagnosis, update operators, candidate allocation, testing, compression, deployment, stopping, and rollback. All methods use the same frozen base model, tool harness, initial skill format, task order, timeouts, verifier configuration, and total inner-task execution budget.

The outer search runs for 60 generations and maintains an archive of at most 40 programs. Each generation proposes eight candidates and applies successive halving over 8, 24, and 60 task instances. Fitness combines final reward, anytime performance, held-out regret, constraint violations, and program complexity. Unless specified otherwise, each method receives 1,200 meta-training executions and 40 executions to adapt a skill to each evaluation family. We report means and standard deviations over five seeds and use paired bootstrap intervals for aggregate comparisons.

Successive halving evaluates inexpensive evidence first and promotes candidates only when their intermediate performance justifies additional executions. Complexity penalties discourage controllers from accumulating branches that improve isolated development cases without producing consistent gains across task families.

4.2 PERFORMANCE AND TRANSFER

Table 1 reports success rates. Figure 3 shows the meta-test success rates under matched inner-task execution budgets. LIFETIME achieves the highest mean score in every domain, with averages of 61.1% on the meta-training domains and 58.7% on the unseen meta-test domains. The strongest baseline, PBT, achieves 57.2% and 54.1%. LIFETIME improves the test average by 4.6 points over PBT, and this difference is positive in four of the five paired seeds. On the held-out average, LIFETIME exceeds EXPERTPOLICY by 5.9 points and FIXEDLOOP by 9.1 points. Matched inner-task execution counts isolate structural lifecycle search from additional sampling.

The improvements span domains with task horizons, reward granularities, and tool interfaces. Compared with PBT, LIFETIME improves science, web synthesis, and kernel optimization by 4.1, 4.7, and 4.9 points. Its improvements on the four training domains range from 3.6 to 4.5 points. Repository maintenance is the hardest domain for all methods and shows the largest training-domain improvement, so domain difficulty does not explain the ranking. On a subset of short, deterministic spreadsheet tasks, the paired bootstrap intervals for LIFETIME and PBT overlap. These tasks have nearly binary failure modes and provide limited opportunities for adaptive diagnosis, evaluation allocation, or rollback.

The frozen lifecycle transfers without outer-loop optimization. Compared with FIXEDLOOP, it improves science and web synthesis by 11.1 points each and kernel optimization by 4.9 points. Retuning the numerical budget constants on development instances adds 1.2 points on average. This result attributes most of the transfer performance to the learned program structure, with a smaller contribution from domain-specific resource settings. Budget-normalized regret yields the same ordering: relative to a per-family oracle, LIFETIME obtains 0.118 ± 0.009 regret, compared with 0.153 ± 0.011 for PBT and 0.169 ± 0.014 for EXPERTPOLICY.

4.3 LIFECYCLE COMPONENTS

The held-out ablations in Table 2 identify three complementary sources of improvement. Fixing the evaluation allocation reduces the average from 58.7% to 56.2%, showing that adaptive budgeting contributes beyond operator selection. Restricting all updates to a single operator decreases the average by 3.8 points because localized procedural omissions and incorrect tool-use strategies require different revisions. Removing rollback decreases the average by 4.1 points and has the largest effect on kernel optimization, where performance falls from 45.7% to 39.8%. Preserving a validated skill is therefore important when low-level transformations can introduce destructive regressions.

These results clarify the role of rollback in the controller. Rollback does not merely undo low-reward updates; it allows the program to explore targeted revisions while retaining a verified recovery point. This mechanism is especially valuable when validation signals are sparse and an apparently plausible modification can damage previously correct behavior.

Successful trajectories provide learning signals. Excluding them reduces the held-out average by 2.4 points, including a 3.0-point reduction on kernel optimization. Given both positive and negative traces, the lifecycle preserves operations shared by successful runs while modifying failure-specific behavior, which avoids unnecessary full-skill rewrites. Single-domain meta-training reaches a test average of 53.8%, 4.9 points below multi-domain evolution. Controllers evolved across diverse domains condition their actions on properties such as verifier disagreement, recurring tool errors, and declining validation reward. Single-domain controllers more closely fit their operator choices and stopping rules to the source environment.

4.4 Efficiency, Robustness, and Limitations

Lifecycle evolution adds control overhead while using fewer evaluations than population-based tuning. As shown in Table 3, LIFETIME uses 13.8k model calls and 57 GPU-hours, compared with 15.9k calls and 63 GPU-hours for PBT, and improves held-out quality by 4.6 points. Its utility is 4.25 quality points per thousand calls, compared with 3.40 for PBT. Simpler methods achieve higher utility because they incur lower costs, but their final quality ranges from 49.6% to 52.8%, compared with 58.7% for LIFETIME. Successive halving and conditional allocation reserve expensive verification for candidates that perform well in earlier stages.

EXPERTPOLICY has the highest initial anytime performance because its hand-designed schedule requires no population warm-up. LIFETIME overtakes all baselines at 35% of the budget and maintains the largest area under the anytime curve thereafter. Evaluations beyond 85% of the budget add less than one point to the average score. Under an alternative base model, a 25% context reduction, and a 10% transient tool-failure rate, LIFETIME retains most of its advantage or remains above PBT. Under these perturbations, compression, repeated low-cost probes, and rollback improve robustness, although the controller remains sensitive to changes in the harness.

The conclusions depend on two conditions. A misaligned verifier can favor lifecycles that exploit the proxy without improving true task quality. Expensive domains can also provide too few evaluations for reliable population selection, causing LIFETIME to behave like a conservative fixed loop. Lifecycle evolution therefore requires informative verification and a sufficient search budget, including when complexity regularization and held-out audits are used.

5 Conclusion

We presented Lifetime Skill-Learning (LSL), a two-level framework that evolves executable skill-learning lifecycles while keeping the foundation model and host agent fixed. By optimizing evidence selection, update operators, evaluation budgets, validation, deployment, and rollback, LSL makes persistent skill improvement adaptive, auditable, and replayable. Cross-domain evaluations, controlled comparisons, and component-level ablations establish that lifecycle evolution improves task performance, transfer, anytime behavior, budget efficiency, and safety while controlling lifecycle complexity. These findings confirm that optimizing how skills are constructed and maintained provides clear empirical advantages over fixed refinement procedures and conventional population-based optimization.

Limitations

Our evaluation covers a finite set of task families and tool environments, so additional domains may reveal different lifecycle preferences. The experiments use a fixed agent harness and a limited selection of frozen base models; broader model and interface coverage could further characterize robustness. Finally, verifier quality and evaluation budgets vary across practical deployments, and the current results do not fully capture settings with sparse feedback, highly stochastic outcomes, or substantially more expensive candidate evaluations.

AI Assistance Statement

We acknowledge the use of AI-based writing tools in the preparation of this manuscript. These tools were employed exclusively to assist with language editing, including improving sentence clarity, correcting grammatical mistakes, and enhancing the overall flow of the text. No AI assistance was sought or used for any aspect of the scientific research, including the design of the meta-evolution framework, implementation of algorithms, collection or analysis of experimental data, or formulation of conclusions. All AI-generated language suggestions were reviewed by the authors before being incorporated into the final text. The authors take full responsibility for all scientific claims and content herein.

References

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. *arXiv preprint arXiv:1606.04474*, 2016. URL https://arxiv.org/abs/1606.04474.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models. *arXiv preprint arXiv:2308.09687*, 2023. URL https://arxiv.org/abs/2308.09687.

Yan Duan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016. URL https://arxiv.org/abs/1611.02779.

Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*, 2018. URL https://arxiv.org/abs/1807.01774.

Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning*, 2024. URL https://proceedings.mlr.press/v235/fernando24a.html.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, 2017. URL https://proceedings.mlr.press/v70/finn17a.html.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers. *arXiv preprint arXiv:2309.08532*, 2023. URL https://arxiv.org/abs/2309.08532.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/36b7acf6f6010652b3f2a433774a66fe-Abstract-Conference.html.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017. URL https://arxiv.org/abs/1711.09846.

Gurusha Juneja, Gautam Jajoo, Hua Li, Jian Jiao, Nagarajan Natarajan, and Amit Sharma. Task facet learning: A structured approach to prompt optimization. In *Findings of the Association for Computational Linguistics: ACL 2025*, 2025. URL https://aclanthology.org/2025.findings-acl.1206/.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=sY5N0zY5Od.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *arXiv preprint arXiv:1612.00796*, 2017. URL https://arxiv.org/abs/1612.00796.

Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 2011. URL https://gpbib.pmacs.upenn.edu/gp-html/Lehman_2011_EC.html.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016. URL https://arxiv.org/abs/1603.06560.

Xiangyi Li et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026. URL https://arxiv.org/abs/2602.12670.

Yuxuan Liu, Zhaochen Su, Yuhao Zhang, Jiahe Guo, Zhongwei Xie, Huihao Jing, Lingyun Xie, Qing Zong, Yauwai Yim, Zhixiong Zhang, Haoran Li, and Yangqiu Song. Rethinking self-evolving agent skills: Feedback dynamics over multiple rounds. *arXiv preprint arXiv:2608.02636*, 2026. URL https://arxiv.org/abs/2608.02636.

David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. *arXiv preprint arXiv:1706.08840*, 2017. URL https://arxiv.org/abs/1706.08840.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023. URL https://arxiv.org/abs/2303.17651.

Luke Metz, James Harrison, C. Daniel Freeman, Amil Merchant, Lucas Beyer, James Bradbury, Naman Agrawal, Ben Poole, Igor Mordatch, Adam Roberts, and Jascha Sohl-Dickstein. Velo: Training versatile learned optimizers by scaling up. *arXiv preprint arXiv:2211.09760*, 2022. URL https://arxiv.org/abs/2211.09760.

Qirui Mi, Zhijian Ma, Mengyue Yang, Haoxuan Li, Yisen Wang, Haifeng Zhang, and Jun Wang. Skill-pro: Learning reusable skills from experience via non-parametric ppo for llm agents. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=9kJQjx2B80.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015. URL https://arxiv.org/abs/1504.04909.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. URL https://arxiv.org/abs/2310.08560.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023. URL https://arxiv.org/abs/2304.03442.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://aclanthology.org/2023.emnlp-main.494/.

David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy P. Lillicrap, and Greg Wayne. Experience replay for continual learning. *arXiv preprint arXiv:1811.11682*, 2018. URL https://arxiv.org/abs/1811.11682.

Juergen Schmidhuber. Goedel machines: Self-referential universal problem solvers making provably optimal self-improvements, 2003. URL https://arxiv.org/abs/cs/0309048.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023. URL https://arxiv.org/abs/2303.11366.

Jun Shu, Qi Xie, Lixuan Yi, Qian Zhao, Sanping Zhou, Zongben Xu, and Deyu Meng. Meta-weight-net: Learning an explicit mapping for sample weighting. *arXiv preprint arXiv:1902.07379*, 2019. URL https://arxiv.org/abs/1902.07379.

Claudio Spiess, Mandana Vaziri, Louis Mandel, and Martin Hirzel. Autopdl: Automatic prompt optimization for llm agents. *arXiv preprint arXiv:2504.04365*, 2025. URL https://arxiv.org/abs/2504.04365.

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. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=ehfRiF0R3a.

Jane X. Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z. Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016. URL https://arxiv.org/abs/1611.05763.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*, 2019. URL https://arxiv.org/abs/1901.01753.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023. URL https://arxiv.org/abs/2309.03409.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023a. URL https://arxiv.org/abs/2305.10601.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.

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

Genrui Zhang, Erle Zhu, Jinfeng Zhou, Caiyan Jia, and Hongning Wang. Skillevolver: Skill learning as a meta-skill. *arXiv preprint arXiv:2605.10500*, 2026. URL https://arxiv.org/abs/2605.10500.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025a. URL https://arxiv.org/abs/2505.22954.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=z5uVAKwmjf.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. *arXiv preprint arXiv:2308.10144*, 2023. URL https://arxiv.org/abs/2308.10144.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025. URL https://arxiv.org/abs/2504.07079.

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

Frédéric Zolla, André Nicolet, and Guillaume Demésy. Photonics in highly dispersive media: The exact modal expansion. *arXiv preprint arXiv:1807.02355*, 2018. URL https://arxiv.org/abs/1807.02355.

APPENDIX

A BUDGETED STATE TRANSITIONS AND ROLLBACK-SAFE SKILL DEPLOYMENT

This section defines the state transitions for candidate generation, isolated evaluation, acceptance, rollback, and final deployment in the inner lifecycle. The procedure enforces the execution budget, repository versioning, and provenance requirements while keeping the agent parameters θ frozen.

Let

$$h_t = (\mathcal{T}_t^{\text{seen}}, \mathcal{Z}_t, \mathcal{S}_t, b_t, q_t)$$

denote the lifecycle state before action $a_t = \pi(h_t)$. The transition operator returns an observation o_t, a nonnegative resource cost c_t, and the next state. An action is admissible if it is permitted by the program control flow and satisfies $c_t \le b_t$. For each executed action, the budget is updated as

$$b_{t+1} = b_t - c_t, \qquad 0 \le c_t \le b_t, \tag{5}$$

which keeps b_t nonincreasing and nonnegative throughout a valid execution. The audit and safety component q_{t+1} records the transition status, and the corresponding provenance record is appended to $\mathcal{G}_\pi$. Each state-changing operation is therefore linked to its input state, selected action, observation, cost, and resulting artifact or decision.

For a candidate-evaluation action at step t, let $s_t \in \mathcal{S}_t$ denote the incumbent and assign it the index $j = 0$. The diagnosis operator computes

$$\delta_t = g_t(\mathcal{Z}_t, s_t) = (e_t^{\text{miss}}, e_t^{\text{wrong}}, e_t^{\text{unused}}, e_t^{\text{risk}}),$$

and the selected edit operator u_t generates

$$\mathcal{C}_t = u_t(s_t, \delta_t; k_t) = \{\tilde{s}_{t,j}\}_{j=1}^{k_t}.$$

Each candidate is evaluated in an isolated agent session on its allocated validation set $\mathcal{V}_{t,j}$, subject to

$$\sum_{j=1}^{k_t} \beta_{t,j} \le b_t. \tag{6}$$

With $\tilde{s}_{t,0} = s_t$, the estimated reward of each evaluated incumbent or candidate is

$$\widehat{R}_{t,j} = \frac{1}{|\mathcal{V}_{t,j}|} \sum_{\tau_i \in \mathcal{V}_{t,j}} V_i(z_{t,j,i}), \qquad z_{t,j,i} \sim \text{Agent}_\theta(x_i, \mathcal{E}_i; \tilde{s}_{t,j}). \tag{7}$$

All sessions share the frozen parameter vector θ; only the supplied skill artifact varies.

Define the set of safe, evaluated candidates as

$$\mathcal{J}_t^{\text{safe}} = \left\{ j \in \{1, \dots, k_t\} : \text{Safe}(\tilde{s}_{t,j}) = 1 \text{ and } \widehat{R}_{t,j} \text{ has been obtained} \right\}. \tag{8}$$

If $\mathcal{J}_t^{\text{safe}}$ is nonempty, let $j_t^\star$ be any maximizer of $\widehat{R}_{t,j}$ over $j \in \mathcal{J}_t^{\text{safe}}$. The repository is updated as

$$s_{t+1} = \begin{cases} \tilde{s}_{t,j_t^\star}, & \mathcal{J}_t^{\text{safe}} \neq \varnothing \text{ and } \text{Accept}(\widehat{R}_{t,j_t^\star}, \widehat{R}_{t,0}) = 1, \\ s_t, & \text{otherwise.} \end{cases} \tag{9}$$

Algorithm 1 Budgeted execution of a frozen lifecycle program

Require: Frozen lifecycle program π; frozen agent Agent_θ; domain tasks $\mathcal{T}_d$; initial versioned repository $\mathcal{S}_0$; execution budget B
Ensure: Deployed skill $s_d^\star$ and replayable provenance graph $\mathcal{G}_d^\star$

Initialize $h_0 = (\varnothing, \varnothing, \mathcal{S}_0, B, q_0)$ and initialize an empty provenance graph $\mathcal{G}_\pi$
Set $t \leftarrow 0$
while $b_t > 0$, an admissible action remains, and the stopping condition of π is not satisfied **do**
    Select $a_t \leftarrow \pi(h_t)$
    if a_t is an observation action **then**
        Execute selected tasks from $\mathcal{T}_d$ with the current skill using Agent_θ
        Append the resulting scored trajectories to $\mathcal{Z}_t$ and the encountered tasks to $\mathcal{T}_t^{\mathrm{seen}}$
    else if a_t is a diagnosis action **then**
        Compute $\delta_t \leftarrow g_t(\mathcal{Z}_t, s_t)$
    else if a_t is an editing or compression action **then**
        Select the designated operator u_t and generate $\mathcal{C}_t \leftarrow u_t(s_t, \delta_t; k_t)$, subject to the remaining budget
    else if a_t is an evaluation action **then**
        Allocate $\{\beta_{t,j}\}$ such that $\sum_j \beta_{t,j} \le b_t$
        for each candidate selected by the allocation **do**
            Evaluate it in an isolated session using Equation (7)
        end for
    else if a_t is a selection action **then**
        Construct $\mathcal{J}_t^{\mathrm{safe}}$ using Equation (8)
        Update the active skill using Equation (9)
        if the candidate is accepted **then**
            Insert the candidate as a child version of s_t
        else
            Retain s_t
        end if
    else if a_t is a rollback action **then**
        Restore the designated validated repository version permitted by the lifecycle program
    else if a_t is a termination action **then**
        Terminate the inner-loop execution
    end if
    Apply $(h_{t+1}, o_t, c_t) \leftarrow \mathrm{Transition}(h_t, a_t; \mathrm{Agent}_\theta)$ with $0 \le c_t \le b_t$
    Set $b_{t+1} \leftarrow b_t - c_t$
    Append $(h_t, a_t, o_t, c_t, h_{t+1})$ and all generated artifact records to $\mathcal{G}_\pi$
    Update the audit and safety status in q_{t+1}
    Set $t \leftarrow t + 1$
end while
Let $T \leftarrow t$
if $\mathrm{Valid}(\pi, \mathcal{G}_\pi) = 1$ **then**
    Select $s_d^\star \in \arg\max_{s \in \mathcal{S}_T} \widehat{R}(s)$ among validated repository versions
    Deploy $s_d^\star \leftarrow \mathrm{Deploy}(s_d^\star)$
else
    Do not deploy any newly generated skill version
end if
Set $\mathcal{G}_d^\star \leftarrow \mathcal{G}_\pi$
return $(s_d^\star, \mathcal{G}_d^\star)$

An accepted candidate is inserted as a new version with s_t as its parent, together with its diagnosis, edit, evaluation, and acceptance records. In all other cases, including an empty safe set, rejection by the acceptance predicate, or the absence of an admissible completed evaluation, the incumbent remains active. This update implements rollback without modifying the incumbent artifact.

During meta-training, each candidate lifecycle program applies this procedure to sampled domains, and the gradient-free outer evolution uses the resulting domain outcomes. At meta-test time, both the

selected program $\pi^\star$ and the agent parameters θ remain frozen. Only the domain-specific repository state, trace buffer, budget, and provenance graph evolve. Training and deployment therefore share the same inner-loop state transitions, acceptance rule, rollback semantics, and termination conditions. They differ in the subsequent treatment of the lifecycle program: during meta-training, the outer loop applies mutation and archive selection; at meta-test time, the selected program remains fixed.

B Environment Details

B.1 Datasets and Task Settings

We organize the evaluation as a cross-domain meta-learning benchmark containing seven domains. Spreadsheet manipulation, document transformation, repository-level software maintenance, and constrained planning constitute the meta-training domains. Scientific data analysis, web-based information synthesis, and GPU-kernel optimization are held out for meta-test evaluation. Each domain contains 20 task families, with five development instances and ten disjoint evaluation instances per family. Development instances are used for lifecycle evolution, hyperparameter selection, and acceptance-threshold calibration. Evaluation instances are accessed only for final reporting. All methods receive the same task order, initial skill representation, execution environment, verifier configuration, timeout limits, and interaction budget.

Spreadsheet tasks require editing structured workbooks while preserving formulas, formatting, references, and unaffected cells. Document-transformation tasks cover structured rewriting, layout-sensitive editing, format conversion, and content-preserving modifications. Repository-maintenance tasks require navigating multi-file codebases, localizing defects, modifying implementations, and passing repository tests. Constrained-planning tasks require producing executable action sequences under ordering, capacity, dependency, and resource constraints. Scientific-analysis tasks combine data preprocessing, statistical computation, visualization, and evidence-grounded conclusions. Web-synthesis tasks require retrieving information from multiple pages, reconciling potentially conflicting evidence, and producing source-grounded responses. GPU-kernel tasks require implementing or optimizing kernels under numerical-correctness and runtime constraints.

Each task is represented by an instruction, an isolated execution environment, and an executable or deterministic verifier. Agent outputs consist of a final response together with the generated artifacts, tool calls, and environment state. Skills are versioned inference-time packages containing procedural instructions and optional scripts, examples, or auxiliary resources. The initial skill package follows a common schema across methods and contains only task-interface documentation and general execution guidelines. Agents are prompted to solve the task step by step, use the available tools when necessary, verify the resulting artifacts, and return a concise completion record.

Task families, templates, and underlying assets are partitioned before instance generation. Meta-test domains are excluded from outer-loop evolution, and evaluation instances are not included in prompts, mutation contexts, trajectory buffers, or skill examples. Near-duplicate instructions are removed using normalized token overlap and embedding-based similarity filtering. Generated artifacts and provenance records are additionally checked to ensure that evaluation-instance identifiers and reference outputs do not appear in evolved skills.

B.2 Preprocessing and Evaluation

Inputs are converted to a unified task schema containing the instruction, available tools, environment metadata, resource limits, and verifier entry point. Text fields are normalized to UTF-8, line endings are canonicalized, and environment-specific absolute paths are replaced with sandbox-relative paths. Tool observations are truncated only when they exceed the model context window; in that case, the most recent observation, error messages, and verifier-relevant sections are retained. Each candidate skill is evaluated in a fresh process with an independent workspace, cleared conversation state, and a fixed task timeout of 20 minutes.

Binary task success is determined by the domain verifier. Spreadsheet and document tasks compare required cell values, formulas, document structure, formatting constraints, and output-file validity. Repository tasks execute the provided unit and integration tests in an isolated container. Planning tasks validate action legality, constraint satisfaction, and goal completion. Scientific-analysis tasks

check required numerical outputs and generated artifacts against reference tolerances. Web-synthesis tasks use deterministic citation, coverage, and answer-field checks followed by a frozen rubric-based evaluator for claims that cannot be verified structurally. GPU-kernel tasks require numerical agreement with the reference implementation within an absolute tolerance of 10^{-4} and a relative tolerance of 10^{-3}; performance rewards are computed only for correct kernels after warm-up runs.

String-valued answers are Unicode-normalized, stripped of leading and trailing whitespace, and compared after collapsing repeated whitespace and canonicalizing line endings. Numeric answers are parsed after removing thousands separators and non-semantic formatting, and are considered correct under the task-specific absolute and relative tolerances. LaTeX-valued answers are normalized by removing presentation-only delimiters, including `\boxed{}`, canonicalizing equivalent fractions and signs, and ignoring non-semantic spacing commands. When multiple reference answers are provided, a prediction is counted as correct if it matches any admissible reference. Invalid files, execution errors, verifier timeouts, missing required outputs, and unsafe tool actions receive zero reward.

The primary metric is task success rate, averaged first within each task family and then across families so that every family has equal weight. Continuous verifier scores are clipped and linearly normalized to $[0, 1]$ using domain-specific reference bounds. We additionally report final reward, area under the anytime-performance curve, regret relative to a per-family oracle, model-call count, GPU-hours, reward per thousand model calls, lifecycle-program complexity, and safety-constraint violation rate. Program complexity is measured as the number of executable nodes plus control-flow edges after dead-code elimination.

All stochastic experiments use five seeds, $\{13, 21, 42, 87, 100\}$, controlling task order, model sampling, population initialization, parent selection, and program mutation. Tables report the mean and standard deviation across seeds. Aggregate method comparisons use paired bootstrap resampling over task families with 10,000 replicates and 95% confidence intervals. The compared methods are FIXEDLOOP, EXPERTPOLICY, METAPROMPT, GREEDYPROGRAM, population-based training (PBT), and LIFETIME. Ablations fix the candidate-evaluation budget, restrict updates to one operator, disable rollback, remove successful trajectories, or restrict meta-training to a single domain.

B.3 IMPLEMENTATION AND HARDWARE

All experiments use `Qwen2.5-7B-Instruct` as the frozen agent backbone, served locally with `vLLM`. The same model checkpoint is used for task execution, trajectory diagnosis, skill revision, candidate generation, lifecycle-program mutation, compression, and rubric-based evaluation. Model weights remain frozen throughout meta-training and meta-testing. No gradients are propagated through task executions, verifiers, skill updates, or lifecycle mutations, and no separate value or reward model is trained. Executable task verifiers provide candidate rewards whenever available; the frozen backbone is used as a judge only for web-synthesis criteria that cannot be evaluated deterministically.

Task-solving calls use temperature 0.2, nucleus sampling with $p = 0.95$, and a maximum of 4,096 generated tokens. Diagnosis, skill-editing, compression, and program-mutation calls use temperature 0.7, $p = 0.95$, and a maximum of 3,072 generated tokens to encourage candidate diversity. Verifier-facing rubric judgments use greedy decoding. The maximum context length is 32,768 tokens. Tool outputs longer than 12,000 tokens are reduced using deterministic head–tail retention and error-focused extraction before being added to the model context.

The inner lifecycle has a maximum depth of six skill-revision rounds. At each round, a program may inspect up to eight trajectories and generate between one and four candidate skills. Candidate generation therefore has a maximum branching factor of four. Each candidate first receives two low-fidelity evaluation rollouts; surviving candidates receive up to eight additional rollouts, subject to the remaining per-family budget. Candidate and incumbent skills are evaluated on identical instances using paired execution seeds. A candidate is accepted only if it is verifier-valid, introduces no new safety violation, and improves mean development reward by at least 0.02. Ties within 0.01 are resolved in favor of the shorter skill. If an accepted candidate underperforms its parent on the subsequent validation batch, the repository is rolled back to the most recent validated version. Inner-loop execution stops after six rounds, after two consecutive rounds without an accepted update, or when the execution budget is exhausted. Each evaluation family receives 40 task executions.

Table 4: Robustness on unseen meta-test domains under deployment perturbations. Results are success rates (%, mean±std over five seeds). Δ denotes the average improvement of LIFETIME over PBT.

	PBT			LIFETIME			
Condition	Science	Web	Kernel	Science	Web	Kernel	Δ
Standard	59.4±1.7	62.1±1.6	40.8±2.1	63.5±1.5	66.8±1.4	45.7±1.9	+4.6
Alternative model	56.7±2.0	59.0±1.8	38.2±2.5	59.8±1.8	62.4±1.7	41.6±2.3	+3.3
Context reduction (25%)	57.9±1.9	59.8±2.1	38.9±2.4	61.7±1.7	64.1±1.8	43.2±2.1	+4.1
Tool failure (10%)	55.8±2.3	58.6±2.0	36.7±2.8	60.2±2.0	63.0±1.9	41.0±2.5	+4.4
Combined	51.9±2.6	54.7±2.4	33.5±3.1	55.4±2.4	58.3±2.2	37.1±2.9	+3.6

The outer search runs for 60 generations and maintains a population archive of at most 40 lifecycle programs. Each generation proposes eight mutated descendants. Mutations can replace update operators, insert or delete typed branches, alter evidence-selection rules, change candidate allocations, modify stopping conditions, or modify rollback conditions. Programs are rejected before execution if they fail static type checking, contain unbounded loops, bypass resource accounting, or omit mandatory provenance and safety checks. Successive halving evaluates surviving programs on 8, 24, and 60 task instances, retaining the top half at each stage. Every method receives at most 1,200 meta-training task executions per seed. Fitness coefficients are fixed using development instances and weight final reward, normalized anytime area, held-out regret, model-call cost, safety violations, and lifecycle complexity as 1.0, 0.2, 0.5, 0.02, 1.0, and 0.01, respectively.

Experiments are executed on a Linux cluster with eight NVIDIA A100 GPUs with 80 GB of memory per node and dual 64-core AMD EPYC processors. Model inference uses `PyTorch 2.5`, CUDA 12.4, BF16 arithmetic, tensor parallelism of two for each model replica, and deterministic verifier execution. Four replicas run concurrently on each node. Repository, document, spreadsheet, web, and kernel environments are isolated using Docker containers with 16 CPU cores and 64 GB of host memory per worker. GPU-kernel timing uses a dedicated A100 device, 20 warm-up iterations, and 100 timed iterations, with median latency used for scoring. The complete LIFETIME search consumes approximately 13.8k model calls and 57 A100 GPU-hours per reported run.

C Additional Experimental Results

C.1 Robustness to Model and Environment Perturbations

We evaluate the learned lifecycle under deployment conditions that differ from meta-training. The alternative-model condition replaces the frozen base model without modifying the lifecycle program. Context reduction truncates the available context window by 25%, prioritizing the current skill and recent observations. Transient tool failures independently replace 10% of tool responses with retryable errors. As shown in Table 4, LIFETIME remains above PBT under all perturbations. The difference between the two methods decreases from 4.6 points under the standard condition to 3.3 points with the alternative base model.

C.2 Anytime Performance

Table 5 reports held-out success as a function of the consumed adaptation budget. EXPERTPOLICY achieves the highest success rates at 10% and 25% of the budget. LIFETIME achieves the highest success rates from 40% through 100% of the budget and the highest normalized area under the anytime curve. Its success rate increases from 57.9 at 85% of the budget to 58.7 at 100%.

C.3 Sensitivity to the Inner-Loop Budget

We vary the number of executions available for adapting each evaluation family while keeping the outer-loop program fixed. As shown in Table 6, the success rates of LIFETIME increase as the number of available executions increases. LIFETIME exceeds PBT by 2.9, 3.3, 4.6, 5.0, and 5.1 points at budgets of 10, 20, 40, 60, and 80 executions, respectively.

Table 5: Anytime success rate on unseen meta-test domains at different fractions of the 40-execution adaptation budget. Values are mean±std over five seeds. AUC is the normalized area under the success–budget curve.

Method	10%	25%	40%	60%	85%	100%	AUC
FIXEDLOOP	41.8±2.1	44.6±2.0	46.8±1.8	48.2±1.7	49.2±1.6	49.6±1.5	46.8±1.4
EXPERTPOLICY	**45.7±1.8**	**49.8±1.7**	51.3±1.6	52.1±1.5	52.6±1.4	52.8±1.4	50.7±1.3
METAPROMPT	42.9±2.5	46.1±2.3	48.5±2.1	50.0±1.9	50.9±1.8	51.2±1.7	48.3±1.7
GREEDYPROGRAM	40.6±2.4	47.7±2.1	51.9±1.9	53.1±1.8	53.6±1.7	53.8±1.6	50.4±1.6
PBT	39.8±2.2	47.9±2.0	52.0±1.8	53.4±1.7	53.9±1.6	54.1±1.5	50.6±1.5
LIFETIME	40.9±2.0	49.1±1.8	**53.7±1.6**	**56.4±1.5**	**57.9±1.4**	**58.7±1.4**	**53.4±1.3**

Table 6: Meta-test average success rate (%, mean±std over five seeds) under different per-family adaptation budgets. The same lifecycle programs are used for all budgets.

Method	10 executions	20 executions	40 executions	60 executions	80 executions
FIXEDLOOP	43.7±1.9	46.8±1.7	49.6±1.5	50.4±1.5	50.8±1.4
EXPERTPOLICY	46.9±1.7	50.1±1.5	52.8±1.4	53.6±1.3	53.9±1.3
PBT	46.2±1.8	50.7±1.6	54.1±1.5	55.0±1.4	55.5±1.4
LIFETIME	**49.1±1.7**	**54.0±1.5**	**58.7±1.4**	**60.0±1.3**	**60.6±1.3**

C.4 DEPLOYMENT SAFETY AND REGRESSION CONTROL

We measure the unsafe candidate rate before filtering, the deployment violation rate after lifecycle validation, and the frequency with which an accepted update decreases held-out reward by more than two percentage points. Table 7 reports these measurements for each method. LIFETIME has deployment violation and harmful regression rates of 0.3% and 3.2%, respectively. Without rollback, its unsafe candidate and deployment violation rates are 3.7% and 0.8%, while its harmful regression rate is 10.8%.

C.5 LIFECYCLE COMPLEXITY AND SEARCH DIVERSITY

Table 8 compares the structural complexity of the selected controllers. LIFETIME contains 28.6 program nodes on average, of which 19.3 are active during a meta-test execution. Its archive contains 7.4 distinct operators on average and has a mean normalized tree-edit distance of 0.37. The corresponding values are 6.2 and 0.18 for GREEDYPROGRAM.

C.6 LEARNED OPERATOR USAGE

Table 9 reports the fraction of accepted updates attributable to each operator. Patching accounts for 34.7% and 31.5% of accepted updates in science and web synthesis, respectively. In kernel optimization, restructuring, check insertion, and rollback account for 27.9%, 22.6%, and 16.4% of accepted updates, respectively. Compression accounts for 8.7%, 7.8%, and 6.1% of accepted updates in science, web synthesis, and kernel optimization.

C.7 PAIRED BOOTSTRAP TESTS

Paired bootstrap tests are conducted over evaluation families using 10,000 resamples. Table 10 reports the mean paired differences and 95% confidence intervals. The meta-test average difference between LIFETIME and each baseline is positive, and all corresponding confidence intervals exclude zero. For individual domains, the confidence interval for the comparison with GREEDYPROGRAM on kernel optimization includes zero.

Table 7: Safety and regression statistics on meta-test domains (%, mean±std over five seeds). Lower is better. Unsafe candidates are measured before validation; deployment violations and regressions are measured after the method's acceptance decision. Abbreviations: Unsafe = unsafe candidates; Violations = deployment violations; Regressions = harmful regressions.

Method	Unsafe	Violations	Regressions	Rollbacks per family
FIXEDLOOP	4.8±0.7	1.3±0.4	8.7±1.2	0.6±0.2
EXPERTPOLICY	4.2±0.6	1.0±0.3	6.9±1.0	0.9±0.3
METAPROMPT	5.5±0.9	1.6±0.5	9.4±1.5	0.7±0.3
PBT	4.6±0.7	0.9±0.3	6.1±0.9	1.2±0.3
LIFETIME w/o rollback	3.7±0.6	0.8±0.3	10.8±1.7	0.0±0.0
LIFETIME	**3.5±0.5**	**0.3±0.2**	**3.2±0.7**	1.8±0.4

Table 8: Complexity of selected lifecycle programs and diversity of the final search archive. Counts are mean±std over five seeds. Active nodes count controller nodes visited during a meta-test execution. Structural diversity is the mean normalized tree-edit distance between archive programs.

Method	Program nodes	Control-flow edges	Active nodes	Distinct operators	Structural diversity
FIXEDLOOP	11.0±0.0	12.0±0.0	11.0±0.0	3.0±0.0	–
EXPERTPOLICY	19.0±0.0	23.0±0.0	15.6±0.8	5.0±0.0	–
METAPROMPT	8.4±1.1	8.1±1.3	8.4±1.1	4.2±0.4	–
GREEDYPROGRAM	25.2±2.7	31.4±3.2	17.9±1.5	6.2±0.7	0.18±0.03
PBT	16.0±0.0	19.0±0.0	14.1±0.6	4.0±0.0	0.09±0.02
LIFETIME	28.6±3.1	36.8±4.0	19.3±1.6	7.4±0.5	**0.37±0.04**

D PROMPT TEMPLATES

D.1 OVERVIEW

This appendix records the prompt templates used by the frozen language model during task execution and skill-lifecycle optimization. The templates cover agent inference, lifecycle action selection, trajectory diagnosis, skill revision and compression, lifecycle-program mutation, and the corresponding prompting-based baselines and ablations. Environment interaction, reward computation, acceptance decisions, and final evaluation are performed by the tool harness and task verifiers rather than by an LLM judge.

D.2 NOTATION

We use the following placeholders throughout the templates:

- {TASK}: the task instruction x.
- {ENVIRONMENT}: the available environment interface, tools, and execution constraints E.
- {SKILL}: the current versioned skill s_t.
- {TRACE}: an execution trajectory containing actions, tool calls, observations, and the final output.
- {TRACE_BUFFER}: the scored trajectory set $Z_t = \{(z_{t,j}, r_{t,j})\}$.
- {SUCCESS_TRACES} and {FAILURE_TRACES}: trajectories partitioned according to verifier outcomes.
- {VERIFIER_FEEDBACK}: deterministic or executable verifier outputs associated with a trajectory.
- {DIAGNOSIS}: the structured deficiency record δ_t.
- {OPERATOR}: the selected skill-update operator.
- {CANDIDATE_COUNT}: the requested number of candidate skills k_t.

Table 9: Distribution of accepted lifecycle actions on meta-test domains (%, mean±std over five seeds). Rows sum to approximately 100%, with deviations caused by rounding.

Domain	Patch	Restructure	Check insertion	Resource revision	Compression	Rollback
Science	34.7±3.2	17.9±2.4	18.8±2.6	10.6±1.9	8.7±1.6	9.4±1.8
Web synthesis	31.5±3.0	19.6±2.5	16.4±2.3	15.8±2.1	7.8±1.5	8.9±1.7
Kernel optimization	18.7±2.6	27.9±3.0	22.6±2.7	8.4±1.6	6.1±1.3	16.4±2.2

Table 10: Paired bootstrap comparisons on unseen meta-test domains. Differences are success-rate points for LIFETIME minus the baseline. Confidence intervals use 10,000 family-level bootstrap resamples.

Baseline	Science	Web	Kernel	Meta-test average
FIXEDLOOP	+11.1 [8.4, 13.8]	+11.1 [8.7, 13.6]	+4.9 [1.2, 8.5]	+9.1 [7.2, 11.0]
EXPERTPOLICY	+6.7 [4.2, 9.1]	+7.2 [4.9, 9.5]	+3.6 [0.4, 6.9]	+5.9 [4.2, 7.6]
METAPROMPT	+9.0 [5.9, 12.0]	+9.5 [6.5, 12.3]	+4.0 [0.2, 7.8]	+7.5 [5.3, 9.7]
GREEDYPROGRAM	+5.8 [3.2, 8.4]	+6.0 [3.5, 8.5]	+2.8 [−0.5, 6.2]	+4.9 [3.1, 6.8]
PBT	+4.1 [1.8, 6.4]	+4.7 [2.5, 6.9]	+4.9 [1.6, 8.1]	+4.6 [2.9, 6.3]

- {CANDIDATE_SKILLS}: generated candidate skill versions $\widetilde{s}_{t,1:k_t}$.
- {VALIDATION_RECORDS}: candidate-level rewards, costs, safety results, and verifier outputs.
- {LIFECYCLE_STATE}: the current state $h_t = (T_t^{\text{seen}}, Z_t, S_t, b_t, q_t)$.
- {ACTION_SCHEMA}: the set of admissible lifecycle actions and their typed arguments.
- {PROGRAM}: an executable lifecycle meta-program π.
- {ARCHIVE}: evaluated lifecycle programs, metrics, lineages, and provenance records.
- {BUDGET} and {REMAINING_BUDGET}: the total and remaining execution budgets.
- {SAFETY_CONSTRAINTS}: the deployment and execution constraints enforced by the harness.
- {PROVENANCE_REQUIREMENTS}: required parent, edit, evaluation, resource, and deployment records.

All structured responses are parsed as JSON. The model is instructed not to emit Markdown fences or text outside the specified object.

D.3 MAIN INFERENCE PROMPT

Purpose. Execute a task using the current domain skill while preserving a complete interaction trace.

Inputs. {TASK}, {ENVIRONMENT}, and {SKILL}.

Expected Output. Tool calls conforming to the environment interface, followed by a task-specific final answer. The host agent records all actions, observations, tool calls, and the final answer in the trajectory.

Prompt.

```
You are an agent operating in the environment described below.

Task:
{TASK}

Environment and available tools:
{ENVIRONMENT}
```

```
Current skill:
{SKILL}

Complete the task using the available tools and the current skill. Follow all
environment constraints and skill preconditions. Check intermediate results
when the environment permits verification. Do not claim completion until the
task requirements have been satisfied.

If a tool call is required, issue it using the environment's tool-call
interface. After completing the task, return only the final task answer in the
format required by the task.
```

D.4 Baseline Prompts

D.4.1 FixedLoop Skill Revision

Purpose. Implement the manually defined observe–revise–test baseline using a fixed candidate allocation and fixed termination procedure.

Inputs. {SKILL}, {TRACE_BUFFER}, {VERIFIER_FEEDBACK}, and a fixed {CANDIDATE_COUNT}.

Expected Output. Exactly {CANDIDATE_COUNT} complete candidate skills.

Prompt.

```
You revise a reusable agent skill from scored execution evidence.

Current skill:
{SKILL}

Execution trajectories and verifier feedback:
{TRACE_BUFFER}
{VERIFIER_FEEDBACK}

Generate exactly {CANDIDATE_COUNT} revised versions of the skill. Identify
recurring failures, preserve instructions supported by successful executions,
and revise instructions that caused or failed to prevent errors. Each
candidate must be self-contained and executable by the same frozen agent.
Do not change the task, tools, verifier, or execution budget.

Return only:
{
  "candidates": [
    {
      "candidate_id": "candidate_1",
      "skill": "<complete revised skill>"
    }
  ]
}
```

D.4.2 ExpertPolicy Diagnosis and Revision

Purpose. Support the domain-agnostic lifecycle manually designed by experienced agent developers. The policy fixes the ordering of diagnosis, operator choice, testing, acceptance, and rollback; the model supplies the diagnosis and revised skill.

Inputs. {SKILL}, {SUCCESS_TRACES}, {FAILURE_TRACES}, {VERIFIER_FEEDBACK}, and the operator prescribed by the expert policy.

Expected Output. A structured diagnosis and one complete revised skill.

Prompt.

```
You are revising an agent skill under a fixed expert-designed lifecycle.

Current skill:
{SKILL}

Successful trajectories:
{SUCCESS_TRACES}

Failed trajectories:
{FAILURE_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Required update operator:
{OPERATOR}

Contrast successful and failed executions. Preserve procedures shared by
successful executions and apply only the required update operator to address
failure-specific deficiencies. Do not alter the environment interface or
introduce unsupported tools.

Return only:
{
  "diagnosis": {
    "missing_procedures": ["<item>"],
    "misleading_instructions": ["<item>"],
    "unused_components": ["<item>"],
    "safety_risks": ["<item>"]
  },
  "revised_skill": "<complete revised skill>"
}
```

D.4.3 METAPROMPT ACTION SELECTION

Purpose. Select the next skill-learning action for METAPROMPT, which uses a single language-model prompt instead of an executable lifecycle controller.

Inputs. {LIFECYCLE_STATE}, {ACTION_SCHEMA}, {REMAINING_BUDGET}, {SAFETY_CONSTRAINTS}, and {PROVENANCE_REQUIREMENTS}.

Expected Output. One admissible lifecycle action with typed arguments and a brief decision rationale.

Prompt.

```
You control one step of a skill-learning lifecycle.

Current lifecycle state:
{LIFECYCLE_STATE}

Admissible actions and argument schema:
{ACTION_SCHEMA}

Remaining execution budget:
{REMAINING_BUDGET}
```

```
Safety constraints:
{SAFETY_CONSTRAINTS}

Provenance requirements:
{PROVENANCE_REQUIREMENTS}

Select exactly one next action. The action must be admissible, remain within
the available budget, preserve versioning and provenance, and never bypass
validation or safety checks. Prefer the action expected to improve validated
task performance rather than training performance alone.

Return only:
{
  "action":
  ↪  "<observe|diagnose|edit|evaluate|select|compress|deploy|rollback|terminate>",
  "arguments": {},
  "rationale": "<brief evidence-based reason>",
  "estimated_execution_cost": <non-negative integer>
}
```

D.5 Few-Shot and Demonstration Prompts

All inference and lifecycle prompts are zero-shot. No task-specific few-shot examples, manually labeled demonstrations, or in-context input–output exemplars are included. Successful trajectories may be selected as execution evidence by a lifecycle program, but they are supplied through the structured trajectory fields defined above rather than as fixed demonstrations embedded in the prompt.

D.6 Reasoning and Rationale Prompts

D.6.1 Trajectory-Contrast Diagnosis

Purpose. Contrast successful and failed trajectories and produce the structured diagnosis used to constrain skill updates.

Inputs. {SKILL}, {SUCCESS_TRACES}, {FAILURE_TRACES}, and {VERIFIER_FEEDBACK}.

Expected Output. A JSON diagnosis containing missing procedures, misleading instructions, unused components, and safety risks.

Prompt.

```
You diagnose deficiencies in a reusable agent skill.

Current skill:
{SKILL}

Successful trajectories:
{SUCCESS_TRACES}

Failed trajectories:
{FAILURE_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Compare successful and failed executions. Attribute each deficiency to
specific evidence. Distinguish failures caused by missing procedures,
incorrect or misleading instructions, unnecessary or unused skill content,
and potentially unsafe actions. Do not propose a complete revised skill.
```

```
Return only:
{
  "missing_procedures": [
    {
      "description": "<missing procedure>",
      "evidence": ["<trajectory or verifier reference>"]
    }
  ],
  "misleading_instructions": [
    {
      "description": "<incorrect or misleading instruction>",
      "evidence": ["<trajectory or verifier reference>"]
    }
  ],
  "unused_components": [
    {
      "description": "<unused component>",
      "evidence": ["<trajectory reference>"]
    }
  ],
  "safety_risks": [
    {
      "description": "<risk>",
      "evidence": ["<trajectory, verifier, or constraint reference>"]
    }
  ]
}
```

D.6.2 LIFECYCLE-ACTION RATIONALE

Purpose. Produce an auditable rationale when a language-model decision is used inside an executable lifecycle action.

Inputs. {LIFECYCLE_STATE}, the selected action, observed evidence, and applicable constraints.

Expected Output. A concise JSON rationale linked to trace, validation, budget, and safety evidence.

Prompt.

```
Explain the supplied lifecycle decision using only the recorded state and
evidence.

Lifecycle state:
{LIFECYCLE_STATE}

Selected action:
{SELECTED_ACTION}

Relevant evidence:
{TRACE_BUFFER}
{VALIDATION_RECORDS}

Constraints:
{SAFETY_CONSTRAINTS}
{PROVENANCE_REQUIREMENTS}

Return only:
{
  "decision": "<selected action>",
  "supporting_evidence": ["<record identifier>"],
```

```
  "expected_effect": "<brief expected effect>",
  "budget_justification": "<brief justification>",
  "safety_and_provenance_check": "<brief check>"
}
```

The rationale is stored for auditing and provenance. Acceptance and deployment remain determined by the executable lifecycle, verifier outcomes, and validity checks.

D.7 Data Generation and Pseudo-Labeling Prompts

No synthetic task generation or pseudo-labeling prompt is used. Tasks are drawn from the predefined domain families, and rewards are supplied by executable or deterministic verifiers. Candidate skills and lifecycle programs are generated by the templates below.

D.7.1 Operator-Conditioned Skill Generation

Purpose. Generate candidate skills using the update operator selected by the lifecycle program.

Inputs. {SKILL}, {DIAGNOSIS}, {OPERATOR}, {CANDIDATE_COUNT}, and {SAFETY_CONSTRAINTS}.

Expected Output. Exactly k_t complete, independently executable candidate skills and their edit records.

Prompt.

```
You generate candidate revisions of a reusable agent skill.

Current skill:
{SKILL}

Structured diagnosis:
{DIAGNOSIS}

Selected update operator:
{OPERATOR}

Number of candidates:
{CANDIDATE_COUNT}

Safety constraints:
{SAFETY_CONSTRAINTS}

Apply only the selected operator:

- patch: make localized changes for identified omissions or incorrect
  constraints;
- restructure: reorganize or replace a flawed procedure while preserving
  validated behavior;
- insert_checks: add precondition, intermediate, or completion checks;
- revise_resources: correct instructions for using existing executable
  resources or tools;
- compress: remove redundancy while preserving validated behavior.

Generate exactly {CANDIDATE_COUNT} distinct candidates. Each candidate must
be a complete skill rather than a patch description. Preserve instructions
supported by successful trajectories unless the diagnosis provides direct
counterevidence. Do not introduce new tools, modify the verifier, or weaken
safety constraints.
```

```
Return only:
{
  "operator": "{OPERATOR}",
  "candidates": [
    {
      "candidate_id": "candidate_1",
      "parent_version": "<current version identifier>",
      "edit_summary": ["<atomic edit>"],
      "addressed_diagnosis_items": ["<diagnosis item identifier>"],
      "skill": "<complete candidate skill>"
    }
  ]
}
```

D.7.2 SKILL COMPRESSION

Purpose. Produce a shorter skill version without intentionally changing its validated behavior.

Inputs. {SKILL}, {SUCCESS_TRACES}, {VALIDATION_RECORDS}, and {SAFETY_CONSTRAINTS}.

Expected Output. One complete compressed skill and an explicit removal record.

Prompt.

```
Compress the reusable agent skill while preserving its validated behavior.

Current skill:
{SKILL}

Successful execution evidence:
{SUCCESS_TRACES}

Validation records:
{VALIDATION_RECORDS}

Safety constraints:
{SAFETY_CONSTRAINTS}

Remove redundancy, unused explanations, and duplicated procedures. Preserve
tool syntax, preconditions, ordering constraints, validation checks, recovery
procedures, and safety requirements supported by the evidence. Do not add new
capabilities or change the environment interface.

Return only:
{
  "compressed_skill": "<complete compressed skill>",
  "removed_or_merged_content": ["<item>"],
  "preserved_critical_constraints": ["<item>"]
}
```

D.7.3 LIFECYCLE-PROGRAM MUTATION

Purpose. Generate a typed descendant lifecycle program by modifying control logic or operator choices while retaining bounded execution and mandatory validation.

Inputs. {PROGRAM}, {ARCHIVE}, the lifecycle action schema, {BUDGET}, {SAFETY_CONSTRAINTS}, and {PROVENANCE_REQUIREMENTS}.

Expected Output. One executable descendant program and a machine-readable mutation record.

Prompt.

```
You mutate an executable program that controls a skill-learning lifecycle.

Parent program:
{PROGRAM}

Evaluated archive:
{ARCHIVE}

Permitted lifecycle actions and types:
{ACTION_SCHEMA}

Execution budget:
{BUDGET}

Safety constraints:
{SAFETY_CONSTRAINTS}

Provenance requirements:
{PROVENANCE_REQUIREMENTS}

Produce one descendant program. You may modify operator calls, typed
branches, bounded loops, evidence-selection rules, candidate-allocation
rules, stopping conditions, or rollback conditions. The descendant must:

1. use only permitted lifecycle actions;
2. have statically typed action arguments;
3. contain only bounded loops;
4. account for every execution cost;
5. preserve mandatory validation, versioning, provenance, and safety checks;
6. never update the frozen agent parameters; and
7. never deploy an unvalidated skill.

Return only:
{
  "parent_program_id": "<identifier>",
  "mutation_summary": ["<structural change>"],
  "program": "<complete executable lifecycle program>",
  "expected_behavioral_difference": "<brief description>"
}
```

D.8 Evaluation Prompts

D.8.1 Verifier-Based Candidate Evaluation

Purpose. Specify the model-facing instruction used when executing a candidate skill on a validation task. Candidate scoring itself is performed by the task verifier.

Inputs. {TASK}, {ENVIRONMENT}, and one candidate skill.

Expected Output. A task-specific final answer and a complete execution trace. The verifier returns the reward and any deterministic feedback separately.

Prompt.

```
You are evaluating a candidate skill in an isolated agent session.

Task:
{TASK}
```

```
Environment and available tools:
{ENVIRONMENT}

Candidate skill:
{CANDIDATE_SKILL}

Complete the task using only the supplied environment and candidate skill.
Follow all task, tool, resource, and safety constraints. Do not use information
from other candidate evaluations. Return only the final answer in the format
required by the task.
```

D.8.2 SAFETY AND PROVENANCE AUDIT RECORD

Purpose. Structure an evidence-grounded audit record for a candidate prior to executable validity and deployment checks.

Inputs. The parent and candidate skills, edit record, validation records, resource usage, {SAFETY_CONSTRAINTS}, and {PROVENANCE_REQUIREMENTS}.

Expected Output. A JSON audit record. The host system independently enforces validity, budget, and safety constraints.

Prompt.

```
Create an audit record for the proposed skill candidate.

Parent skill:
{PARENT_SKILL}

Candidate skill:
{CANDIDATE_SKILL}

Edit record:
{EDIT_RECORD}

Validation records:
{VALIDATION_RECORDS}

Resource usage:
{RESOURCE_USAGE}

Safety constraints:
{SAFETY_CONSTRAINTS}

Provenance requirements:
{PROVENANCE_REQUIREMENTS}

Use only the supplied records. Do not infer an unobserved evaluation result
and do not authorize deployment.

Return only:
{
  "parent_identified": <true|false>,
  "edit_record_complete": <true|false>,
  "validation_record_complete": <true|false>,
  "resource_record_complete": <true|false>,
  "potential_safety_violations": ["<constraint and evidence reference>"],
  "missing_provenance": ["<required record>"],
```

```
  "evidence_references": ["<record identifier>"]
}
```

No LLM-as-a-judge prompt is used for task scoring, candidate acceptance, or reported evaluation. Candidate rewards are computed by the domain verifier, and a candidate replaces the incumbent only through the lifecycle's executable acceptance rule. Safety, budget, and provenance requirements are additionally enforced by the host system.

D.9 Ablation Prompt Variants

D.9.1 Fixed-Budget Variant

Purpose. Remove adaptive evaluation allocation while preserving all other lifecycle decisions.

Inputs. The standard lifecycle state and a fixed evaluation schedule.

Expected Output. The same output as the standard lifecycle-action prompt, with evaluation allocations constrained to the supplied schedule.

Prompt.

```
{METAPROMPT_ACTION_SELECTION_PROMPT}

Additional ablation constraint:
Evaluation allocation is fixed to the following schedule:
{FIXED_EVALUATION_SCHEDULE}

Do not modify candidate counts, fidelity levels, stage sizes, or per-candidate
evaluation allocations. Select all other lifecycle actions normally.
```

D.9.2 Fixed-Operator Variant

Purpose. Remove adaptive update-operator selection.

Inputs. The standard diagnosis and a single fixed update operator.

Expected Output. Candidate skills generated exclusively with the fixed operator.

Prompt.

```
{OPERATOR_CONDITIONED_SKILL_GENERATION_PROMPT}

Additional ablation constraint:
The update operator is fixed to:
{FIXED_OPERATOR}

Do not select, substitute, or combine another update operator.
```

D.9.3 No-Rollback Variant

Purpose. Disable rollback to a previously validated skill version.

Inputs. The standard lifecycle state and action schema with rollback removed.

Expected Output. One admissible non-rollback lifecycle action.

Prompt.

```
{METAPROMPT_ACTION_SELECTION_PROMPT}

Additional ablation constraint:
The rollback action is unavailable. Do not select or emulate rollback.
Continue from the current skill version using another admissible action, or
terminate when no admissible action remains.
```

D.9.4 NO-SUCCESS-TRACES VARIANT

Purpose. Remove successful trajectories from diagnosis and revision evidence.

Inputs. {SKILL}, {FAILURE_TRACES}, and corresponding verifier feedback.

Expected Output. The standard diagnosis schema without references to successful trajectories.

Prompt.

```
You diagnose deficiencies in a reusable agent skill using failed executions
only.

Current skill:
{SKILL}

Failed trajectories:
{FAILURE_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Identify missing procedures, misleading instructions, unused components that
are directly evidenced by the failed executions, and potential safety risks.
No successful trajectories are available. Do not infer which existing
instructions are validated by successful behavior.

Return only:
{
  "missing_procedures": [
    {
      "description": "<missing procedure>",
      "evidence": ["<failure trajectory or verifier reference>"]
    }
  ],
  "misleading_instructions": [
    {
      "description": "<incorrect or misleading instruction>",
      "evidence": ["<failure trajectory or verifier reference>"]
    }
  ],
  "unused_components": [
    {
      "description": "<unused component>",
      "evidence": ["<failure trajectory reference>"]
    }
  ],
  "safety_risks": [
    {
      "description": "<risk>",
      "evidence": ["<failure trajectory, verifier, or constraint reference>"]
    }
```

```
  ]
}
```

D.9.5 SINGLE-DOMAIN META-TRAINING VARIANT

Purpose. Restrict lifecycle-program evolution to evidence and archive entries from one meta-training domain.

Inputs. The standard program-mutation inputs filtered to a single domain.

Expected Output. One executable descendant lifecycle program in the standard mutation format.

Prompt.

```
{LIFECYCLE_PROGRAM_MUTATION_PROMPT}

Additional ablation constraint:
All archive entries, metrics, and execution evidence are drawn from:
{META_TRAIN_DOMAIN}

Use no evidence, metrics, or program evaluations from other domains.
```

The single-domain variant changes only the meta-training data available to lifecycle evolution; it does not introduce a distinct task-inference prompt.

D.10 OUTPUT FORMATTING

Lifecycle-level model calls use strict JSON outputs. Arrays may be empty when no evidence supports an item, but all required keys must be present. Identifiers must refer to versioned skills, trajectories, verifier records, or archive entries supplied in the input. Free-form text outside the JSON object is rejected and retried within the same model-call accounting policy.

Skill fields contain complete inference-time artifacts rather than diffs. Program fields contain complete executable lifecycle programs rather than natural-language plans. The host system parses each response, checks its schema and types, and rejects malformed outputs before execution. Lifecycle programs additionally undergo static type checking and bounded-execution checking. Candidate deployment requires a reproducible parent identifier, edit record, validation record, resource record, and satisfaction of the executable budget, provenance, and safety checks.

E CASE STUDY

We qualitatively inspect representative lifecycle executions to understand how the learned controller adapts its refinement strategy to different failure modes. The cases below are drawn from held-out task families and illustrate three recurring behaviors: selecting an update operator based on the diagnosed cause rather than the surface error, preserving useful procedures from successful trajectories, and rejecting locally plausible revisions that introduce regressions. All examples use a frozen base agent; only the skill artifact is modified.

E.1 CASE I: REPAIRING EVIDENCE ALIGNMENT IN WEB SYNTHESIS

The first case concerns a web-synthesis task in which the agent must compare several software services and produce a recommendation supported by retrieved evidence. The initial skill already contains generally useful browsing instructions, including query decomposition, preference for primary sources, and citation insertion. However, it treats citation as a final formatting operation. Consequently, the agent collects relevant pages but loses the correspondence between individual claims and their sources during synthesis.

A representative failed output produced by the initial skill is shown below.

```
Recommendation: Service A is the better choice for small teams because it
offers a lower entry price, stronger access controls, and broader integration
support. Service B is preferable when advanced analytics are required.

Sources:
[1] Service A pricing page
[2] Service B documentation
[3] Independent product comparison
```

The answer is fluent and its source list appears plausible, but the verifier cannot determine which source supports "lower entry price," "stronger access controls," or "broader integration support." Inspection of the trajectory reveals that the agent opened separate pricing, security, and integration pages, then compressed their contents into an unstructured note. During answer generation, it reconstructed the comparison from memory and attached the collected URLs only after drafting the recommendation. The failure is therefore not missing retrieval. It is a loss of evidence alignment between browsing and synthesis.

A fixed rewrite loop responds to the verifier feedback by appending a broad instruction such as "include citations for all factual claims." This changes the wording of the final answer but not the intermediate representation. In subsequent executions, the agent places several citations at the end of a paragraph without establishing which citation entails which claim. In some traces, it also cites a pricing page for an access-control statement because both facts concern the same service.

The LSL controller contrasts this failure with a successful trajectory from a different comparison task. In the successful trace, the agent had maintained a compact claim ledger while browsing. Each entry paired a normalized claim with a source location and a short supporting passage. The diagnosis therefore records the existing retrieval procedure as useful, identifies the missing claim-to-source binding as the failure-specific deficiency, and selects a check-insertion operator rather than restructuring the entire skill.

The resulting skill patch is localized:

```
Before drafting the answer, construct an evidence ledger.

For every externally verifiable claim, record:
- claim: the exact proposition that may appear in the answer;
- entity: the product, organization, or system to which it applies;
- source: the page that directly supports the proposition;
- support: the smallest retrieved passage that entails the claim;
- status: supported, conflicting, or unresolved.

Do not merge claims merely because they concern the same entity.
Draft only from entries marked supported. Place the corresponding citation
immediately after the supported claim. If sources conflict, state the scope
or date of the disagreement instead of selecting one silently.
```

With this skill, the agent's intermediate artifact becomes explicit:

```
claim: Service A provides SAML-based single sign-on on its enterprise plan.
entity: Service A
source: official security documentation
support: "SAML SSO is available to Enterprise workspaces."
status: supported

claim: Service A has broader integration support than Service B.
entity: Service A / Service B
source: integration directories
```

```
support: no comparable definition of "integration" across the two directories
status: unresolved
```

The revised final answer retains the supported access-control statement and removes the unsupported claim about broader integrations. It also qualifies the pricing comparison by plan and billing period instead of presenting "lower price" as an unconditional product property. The important behavioral change is not simply that the answer contains more citations. The skill moves evidence alignment upstream, so unsupported generalizations are intercepted before generation.

The provenance record links the deployed patch to the failed trace, the successful claim-ledger trace, the diagnosis, and the validation executions. This makes it possible to audit why the new instruction was introduced and which prior behavior it was designed to preserve.

E.2 Case II: Structural Replacement for a Kernel-Optimization Failure

The second case illustrates a failure that cannot be repaired by adding a local reminder. The task asks the agent to optimize a GPU kernel while preserving its functional contract across input shapes. The incumbent skill recommends reading the reference implementation, identifying expensive operations, applying low-level transformations, and running the provided benchmark. It does not clearly separate semantic validation from performance testing.

In one execution, the agent observes a redundant boundary check and rewrites the kernel so that each program instance processes a larger contiguous block. The generated code is shorter and executes normally on the initially inspected shape:

```
pid = program_id(0)
offsets = pid * BLOCK + arange(0, BLOCK)
x = load(x_ptr + offsets)
y = load(y_ptr + offsets)
store(out_ptr + offsets, maximum(x, y))
```

The transformation silently removes masked loads and stores. It remains correct when the input length is aligned with the block size but accesses out-of-range locations for ragged lengths. Because the initial benchmark uses an aligned example, the agent interprets successful execution as evidence that the optimization is valid and proceeds to make further tuning changes.

The lifecycle diagnosis distinguishes this case from a syntax or API-use failure. The generated kernel compiles, the launch configuration is accepted, and the intended elementwise operation is recognizable. The failure arises because the skill's workflow allows optimization hypotheses to be validated on the same narrow example that motivated them. A localized instruction such as "remember edge cases" is judged insufficient: the skill needs a different ordering of actions and an explicit semantic gate.

LSL therefore chooses a restructuring operator and replaces the single-pass optimization procedure with a staged workflow:

```
Stage A: Recover the contract.
1. State the output shape, indexing domain, dtype behavior, broadcasting
   assumptions, and legal memory region.
2. Identify dimensions or lengths that need not align with a launch block.
3. Preserve the reference implementation as an immutable oracle.

Stage B: Establish semantic probes.
1. Test a minimal valid input.
2. Test a ragged shape that crosses a block boundary.
3. Test a non-contiguous or broadcasted input when permitted by the contract.
4. Compare outputs with the oracle before collecting timing evidence.

Stage C: Apply one optimization hypothesis.
```

```
Record the transformation and the assumption that makes it valid. If the
assumption is not guaranteed by the task contract, implement the required
masking or reject the transformation.

Stage D: Validate before benchmarking.
Run the semantic probes in an isolated session. Benchmark only candidates
that pass every applicable probe. Never interpret successful compilation or
a single aligned execution as semantic validation.
```

The revised skill causes the agent to retain masking in the boundary-sensitive operations:

```
pid = program_id(0)
offsets = pid * BLOCK + arange(0, BLOCK)
mask = offsets < n
x = load(x_ptr + offsets, mask=mask)
y = load(y_ptr + offsets, mask=mask)
store(out_ptr + offsets, maximum(x, y), mask=mask)
```

A later candidate attempts a more aggressive vectorized transformation. It passes the common contiguous examples but fails a probe involving a permitted broadcasted operand. The candidate's local execution trace contains no tool error, and its code superficially resembles the successful version. Nevertheless, its validation record conflicts with the recovered contract. The lifecycle rejects the candidate and restores the last validated skill rather than incorporating the new optimization advice.

This case exposes the role of rollback as a semantic safeguard. Without rollback, the most recently edited skill would remain active and could teach future executions to repeat an invalid transformation. Under LSL, the failed candidate is retained in the provenance graph as negative evidence, while deployment points back to the validated parent. The repository therefore records both the unsuccessful branch and the reason it was not promoted.

E.3 Case III: Preserving Successful Behavior During Document Transformation

The third case involves a document-transformation skill. The task requires the agent to reorganize a technical report while preserving cross-references, code blocks, and literal command strings. The initial skill correctly instructs the agent to extract headings, rewrite prose for consistency, and save the result to a new artifact. However, it does not distinguish editable natural language from protected spans.

A failed trajectory begins with a structurally valid source fragment:

```
As discussed in Section~\ref{sec:setup}, execute
\texttt{python run.py --config configs/base.yaml}.
Do not replace the token \texttt{AUTO}.
```

The agent improves the surrounding prose but also "normalizes" the command and identifier:

```
As described in the setup section, run
\texttt{python run.py -config config/base.yml}.
Do not replace the token \texttt{Auto}.
```

The output remains readable, and a prose-oriented review may overlook the corruption. Yet the section reference is no longer machine-resolvable, the command-line flag has changed, the path refers to a different file, and a case-sensitive token has been altered. The observed problem is not poor writing quality. It is an inappropriate transformation boundary.

The trace buffer also contains a successful document task in which the agent copied a code sample verbatim while revising only its surrounding explanation. LSL uses this successful trace to identify

behavior that should be generalized: protected content must be isolated before rewriting and restored afterward. The diagnosis marks the prose-revision instructions as useful, while identifying the absence of a preservation protocol as the specific deficiency. It selects a resource-revision operator and adds a lightweight span manifest to the skill package.

The evolved procedure is:

```
Before rewriting, extract protected spans and assign stable placeholders.

Protected spans include:
- code blocks and inline code;
- command lines, file paths, URLs, and configuration keys;
- labels, references, citations, and other document-control sequences;
- user-declared literal strings;
- content whose capitalization or punctuation is semantically significant.

Store, for each placeholder:
- the original byte sequence;
- its span type;
- the surrounding anchor text;
- whether relocation is permitted.

Rewrite only the unprotected text. Restore protected spans from the manifest,
not from model memory. After restoration, verify that every placeholder is
resolved exactly once and that each protected byte sequence matches its
source unless the task explicitly requests its modification.
```

The candidate produced under this procedure reads:

```
As described in Section~\ref{sec:setup}, run
\texttt{python run.py --config configs/base.yaml}.
The literal token \texttt{AUTO} must remain unchanged.
```

Here, the prose is edited while the control sequence, command, path, and literal token remain intact. The lifecycle additionally inserts a post-restoration check because the first patched candidate left one placeholder unresolved inside a footnote. Instead of rewriting the skill again from scratch, it adds a narrow invariant: no deployment may occur while placeholder markers remain in the output.

This case also demonstrates skill compression. During candidate generation, one revision accumulates separate warnings for commands, paths, labels, citations, inline code, and identifiers. Although individually correct, the instructions are repetitive and encourage the agent to perform several overlapping scans. The compression operator replaces them with the unified protected-span abstraction above. The compressed candidate is accepted only after it reproduces the preservation behavior of its parent on isolated validation tasks. Thus, compression removes redundant wording without discarding the operational invariant learned from earlier failures.

E.4 Cross-Case Interpretation

The three cases show that a useful lifecycle policy cannot be reduced to repeated skill rewriting. In web synthesis, the relevant repair is a localized evidence-alignment check that preserves an already effective retrieval strategy. In kernel optimization, the failure requires structural replacement of the validation order, followed by rejection and rollback when a candidate violates the recovered contract. In document transformation, positive and negative trajectories jointly reveal a reusable protected-span abstraction, which is subsequently compressed without weakening its preservation guarantees.

Across cases, the controller responds to the inferred cause of failure rather than its final manifestation. Missing citations are treated as a broken intermediate representation rather than a formatting defect. A kernel that executes successfully is not treated as correct when the test coverage does not reflect the task contract. A fluent document rewrite is rejected when it mutates protected literals. Each

deployed skill is consequently accompanied by a replayable chain connecting observed traces, diagnosis records, selected update operators, candidate artifacts, validation evidence, and deployment or rollback decisions. These records make the learned procedures inspectable and clarify not only what changed, but why a particular type of change was selected.