OPEN-ENDED SKILL SPECIATION: EVOLVING NEW ABSTRACTIONS FOR AGENT SKILLS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model agents use reusable skills to encode procedural knowledge, tool interactions, and control logic. Existing skill-evolution methods optimize skill content within fixed representations or reward diversity without identifying behaviorally meaningful changes in abstraction. We formulate open-ended skill abstraction discovery as the evolution of an expanding repertoire that introduces temporal abstractions and intermediate representations while retaining verifiable utility. We propose Open-Ended Skill Speciation (OESS), a species-aware evolutionary framework that preserves and develops structural innovations when they address previously uncovered failure cases. OESS represents each skill using its input-output signature, temporal abstraction level, tool-dependency graph, failure-repair profile, and execution traces. A structured distance metric distinguishes changes in behavioral organization from superficial textual variation. To enable transitions between abstraction levels, OESS applies macro-mutations that convert repeated procedures into parameterized routines, encapsulate coordinated subskills, extract planners and verifiers, and transform linear workflows into state-dependent controllers. Candidates compete on quality within each species, while inter-species novelty and a lineage archive preserve intermediate innovations for subsequent composition. We evaluate OESS on task families spanning primitive, macro, monitor, and meta-level skills under matched execution budgets. We further test transfer across unseen tasks, models, agent harnesses, and environment perturbations. The evaluation measures verified failure coverage, species survival, abstraction depth, reuse, composition, and lineage-level transitions, yielding a systematic protocol for distinguishing functional abstraction discovery from inconsequential structural diversity.

1 INTRODUCTION

Large language model (LLM) agents increasingly rely on *skills*, reusable units that package procedural knowledge, tool interactions, and control logic. Skills allow agents to reuse solutions across tasks and expose their behaviors as inspectable and transferable artifacts. Recent systems acquire skills autonomously by synthesizing reusable APIs from web interactions (Zheng et al., 2025), combining task-specialized models through quality-diversity optimization (Kuroki et al., 2025), or modifying agent implementations within an open-ended archive (Zhang et al., 2025). These developments motivate a broader objective: discovering new *kinds* of skills whose internal abstractions support behaviors unavailable in the current repertoire.

This objective requires searching over both skill content and structure. Primitive skills encode local action rules, while macro-skills encapsulate reusable multi-step subroutines. Monitor skills inspect execution and trigger recovery, whereas meta-skills construct, select, or coordinate other skills. Transitions between these levels require structural changes beyond revising instructions within a fixed template. Such changes include introducing persistent state, extracting planners or verifiers, encapsulating multiple subskills behind a new interface, and converting linear workflows into feedback controllers. We define *open-ended skill abstraction discovery* as the evolution of an expanding repertoire in which candidates introduce new temporal abstractions and intermediate representations while retaining behavioral utility.

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.

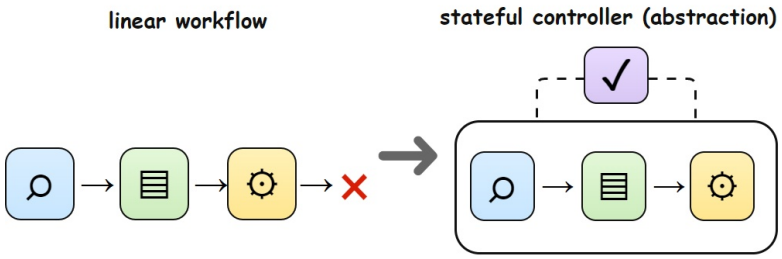

Figure 1: Overview of Open-Ended Skill Speciation.

Existing approaches provide several components of this process but do not directly support abstraction discovery. Skill-synthesis systems abstract successful trajectories into reusable procedures, but generally refine predefined skill representations within domain-specific workflows (Zheng et al., 2025). Quality-diversity methods maintain high-performing solutions across behavioral characteristics (Kuroki et al., 2025)(Cully, 2020), and open-ended agent archives preserve modification paths that can serve as stepping stones (Zhang et al., 2025). Diversity alone, however, does not identify abstraction-changing innovations. Textually dissimilar skills can reproduce known behavior, while structurally novel candidates can initially underperform despite enabling stronger descendants. Aggregate-quality selection can remove such candidates prematurely, and text-embedding novelty can reward superficial rewrites. Novelty-only search can also retain arbitrary complexity without addressing new failures. Open-ended abstraction discovery therefore requires a mechanism that identifies, protects, and develops *behaviorally verified structural innovations*.

We introduce *Open-Ended Skill Speciation* (OESS), a framework for evolving new abstractions in file-based agent skills. An overview of the framework is shown in Figure 1. OESS treats abstraction-changing candidates as emerging species whose survival depends on newly verified behavior. The framework characterizes each skill by its input-output signature, temporal abstraction level, tool-dependency graph, failure-repair profile, and execution traces. A structured distance over these characteristics groups related candidates and detects departures from existing behavioral organizations. Drawing on evolutionary methods that protect structural innovations through speciation (Stanley & Miikkulainen, 2002), OESS creates a species when a structurally distant candidate solves a previously uncovered and verifiable failure set. This criterion separates functional innovation from textual or architectural variation that does not change behavior.

To make abstraction-level transitions reachable, OESS combines local revisions with *abstraction-changing macro-mutations*. Procedural abstraction converts repeated action patterns into parameterized routines. Subskill encapsulation exposes a reusable interface over coordinated components. Planner and verifier extraction separates decision making, execution, and checking. Workflow-to-controller mutation introduces state-dependent control into linear procedures. Candidates compete on quality within species, giving new abstractions time to mature, while inter-species novelty promotes exploration across behavioral organizations. A lineage archive records ancestry, mutation operators, species transitions, and verified failure coverage across generations. The archive preserves useful stepping stones, supports subsequent composition, and enables analysis of abstraction-level capability transitions.

We evaluate OESS on task families requiring primitive, macro, monitor, and meta-level skills. Under matched execution budgets, we compare local mutation, random macro-mutation, novelty search, and species-aware verified search. The evaluation measures task performance, new-species survival, cross-task reuse, hierarchy depth, compositional benefit, and coverage of previously unsolved failures. We test transfer across unseen tasks, foundation models, agent harnesses, and environment perturbations to assess whether the evolved abstractions capture reusable behavior beyond benchmark-specific edits. Ablations remove species protection, macro-mutations, behavioral verification, and lineage retention, and replace structured species distance with text-embedding similarity. Lineage analyses identify the intermediate species and mutation events that precede abstraction-level transitions.

Our contributions are:

- We formulate open-ended skill abstraction discovery as the evolution of behaviorally useful skill structures across temporal and functional levels, extending optimization beyond the content of a fixed representation.
- We propose OESS, a species-aware evolutionary framework that combines structured skill distance, verified species formation, intra-species quality selection, inter-species novelty, and lineage preservation.
- We introduce abstraction-changing macro-mutations that evolve procedural abstractions, encapsulated subskills, planners, verifiers, and stateful controllers from existing skills.
- We develop an evaluation protocol that separates functional innovation from superficial diversity by measuring failure coverage, species survival, abstraction depth, reuse, composition, and transfer, supported by ablation and lineage analyses.

2 Related Work

2.1 Reusable Skills and Self-Improving Language Agents

Tool-augmented language models interleave generation with external computation. ReAct combines reasoning and actions, while Toolformer learns when and how to invoke predefined tools (Yao et al., 2022) (Schick et al., 2023). Gorilla retrieves API documentation, and embodied agents expose structured action interfaces for planning over external controllers (Patil et al., 2024) (Zhu et al., 2023). Other systems construct reusable artifacts from solved instances: LLMs as Tool Makers caches executable functions, CRAFT validates and abstracts code snippets, and SkillWeaver synthesizes and refines APIs from web interactions (Cai et al., 2023) (Yuan et al., 2023) (Zheng et al., 2025).

Open-world agents extend this process through continual acquisition. Voyager stores executable Minecraft programs, whereas DEPS revises long-horizon plans using execution feedback (Wang et al., 2024) (Wang et al., 2023). Related systems retain textual knowledge, executable skills, or feasibility-aware plans across episodes (Zhu et al., 2023) (Wang et al., 2023). Experience-based approaches instead preserve reflections or improve search-time deliberation (Shinn et al., 2023) (Zhou et al., 2024). Evolutionary frameworks broaden adaptation to populations of specialized or self-modifying agents (Kuroki et al., 2025) (Zhang et al., 2025).

2.2 Temporal Abstraction and Unsupervised Skill Discovery

Hierarchical reinforcement learning formalizes reusable behavior through temporal and task abstraction. Options define initiation, policy, and termination, while MAXQ decomposes an MDP and its value function using a subtask hierarchy (Sutton et al., 1999) (Dietterich, 1999). Option-Critic learns option policies and termination functions, while FeUdal Networks and hierarchical deep Q-networks implement manager-worker control with latent goals or intrinsic subgoal rewards (Bacon et al., 2016) (Vezhnevets et al., 2017) (Kulkarni et al., 2016). Goal-conditioned methods use intermediate targets: HIRO relabels high-level actions for off-policy learning, and Hierarchical Actor-Critic uses hindsight transitions under sparse rewards (Nachum et al., 2018) (Levy et al., 2017).

Unsupervised discovery encourages distinguishable or controllable behaviors. Variational Intrinsic Control and DIAYN maximize information between latent skills and states (Gregor et al., 2016) (Eysenbach et al., 2019), whereas DADS emphasizes predictable, distinguishable transitions (Sharma et al., 2019). APS combines successor features with state entropy, and CIC learns contrastive transition-skill representations (Liu & Abbeel, 2021) (Laskin et al., 2022).

2.3 Quality-Diversity and Open-Ended Evolution

Evolutionary methods preserve alternatives that objective-only optimization may discard. NEAT evolves topology using mutation and speciation; novelty search rewards behavioral difference; and local competition combines novelty with neighborhood-level quality (Stanley & Miikkulainen, 2002) (de Haro et al., 2008) (Szépkúti, 2011). MAP-Elites retains high-quality solutions across descriptor cells, with extensions using multiple emitters, covariance adaptation, differentiable objectives, or reinforcement-learning updates (Mouret & Clune, 2015) (Cully, 2020) (Fontaine et al., 2019) (Fontaine & Nikolaidis, 2021) (Pierrot & Flajolet, 2023).

Open-ended environment generation coevolves agents and challenges. POET and Enhanced POET evolve environments, transfer policies, and retain solvable challenges (Wang et al., 2019) (Wang et al., 2020). PAIRED generates curricula through regret, while Minimal Criterion Coevolution retains artifacts satisfying viability thresholds (Jiang et al., 2020) (Gorham & Laughlin, 2019). Recent language-agent systems similarly preserve diverse specialized or self-modifying implementations (Kuroki et al., 2025) (Zhang et al., 2025), while skill-synthesis systems accumulate verified artifacts organized through retrieval, task identity, or fixed tool representations (Zheng et al., 2025) (Yuan et al., 2023).

3 METHOD

We introduce Open-Ended Skill Speciation (OESS), an archive-based search procedure that discovers agent skills at increasingly expressive levels of abstraction. OESS represents each skill by its input-output signature, temporal abstraction, tool dependencies, failure-repair profile, and verified execution traces. These properties define a behavioral species space in which related skills compete locally, behaviorally distinct species coexist, and macro-mutations transform low-level procedures into reusable macro-skills, monitors, controllers, and meta-skills.

3.1 PROBLEM FORMULATION AND FRAMEWORK

Let $\mathcal{T} = \{\tau_i\}_{i=1}^{N}$ denote tasks sampled from $p(\tau)$. Each task specifies an initial state, an instruction, and an executable verifier V_τ. Given a skill s, the executor $\mathcal{E}$ produces the execution trace $\xi(s,\tau) = \mathcal{E}(s,\tau) = \big(o_0, a_0, o_1, a_1, \ldots, o_H\big)$, where o_t, a_t, and H denote the observations, actions, and realized horizon, respectively. The verifier returns $V_\tau(\xi) \in \{0,1\}$. We define the quality of skill s on $\mathcal{T}_s \subseteq \mathcal{T}$ as

$$Q(s) = \frac{1}{|\mathcal{T}_s|} \sum_{\tau \in \mathcal{T}_s} V_\tau(\xi(s,\tau)) \,. \tag{1}$$

A skill is an executable artifact $s = (c_s, \pi_s, \sigma_s)$ comprising an implementation c_s, an execution policy π_s, and a declared input-output signature σ_s. Starting from an initial archive $\mathcal{A}^0$, OESS constructs an archive $\mathcal{A}^G$ that balances skill quality, behavioral coverage, and abstraction diversity.

At generation g, OESS partitions the archive $\mathcal{A}^g$ into species $\mathcal{C}^g = \{C_k^g\}_{k=1}^{K_g}$ and connects archived skills through a directed lineage graph $\mathcal{L}^g$. Each iteration selects a species and a parent skill, applies a macro-mutation, evaluates the resulting candidate on diagnostic tasks, measures its verified novelty and structured distance, and updates the species partition and lineage graph. Figure 2 summarizes the complete OESS search framework, including its structured skill representation, species-aware selection, macro-mutation, verification-based admission, and archive maintenance. The complete optimization procedure is provided in the Appendix.

3.2 STRUCTURED REPRESENTATION AND SPECIATION

We map each skill to the structured descriptor

$$\Phi(s) = \big(\phi_{\text{io}}(s), \phi_{\text{temp}}(s), \phi_{\text{tool}}(s), \phi_{\text{repair}}(s), \phi_{\text{beh}}(s)\big), \tag{2}$$

where ϕ_{io} describes typed input-output semantics; ϕ_{temp} identifies the policy's abstraction level and control horizon; ϕ_{tool} represents tool or subskill invocations and data flow as a typed graph; ϕ_{repair} records failure-conditioned repairs and recovery behavior; and ϕ_{beh} summarizes verified action sequences, state transitions, termination decisions, and success patterns. The temporal abstraction levels are $\mathcal{H} = \{\text{Primitive}, \text{Macro}, \text{Monitor}, \text{Controller}, \text{Meta}\}$.

For skills s_i and s_j, we combine normalized component distances as

$$\begin{aligned} D(s_i, s_j) = w_{\text{io}} d_{\text{io}} + w_{\text{temp}} d_{\text{temp}} + w_{\text{tool}} d_{\text{tool}} \\ + w_{\text{repair}} d_{\text{repair}} + w_{\text{beh}} d_{\text{beh}}, \end{aligned} \tag{3}$$

where the arguments (s_i, s_j) are omitted on the right-hand side, each component lies in $[0,1]$, $w_r \geq 0$, and $\sum_r w_r = 1$. Each species C_k is represented by the executable medoid $\mu_k =$

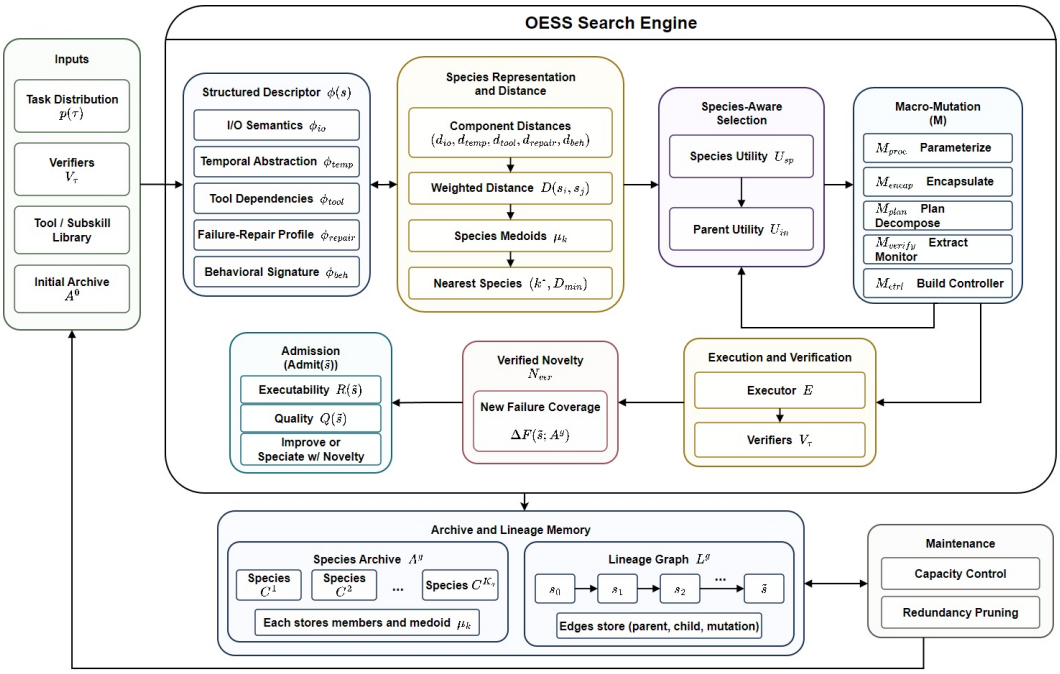

Figure 2: Overview of the OESS search framework.

$\arg\min_{s \in C_k} \sum_{s' \in C_k} D(s, s')$. For a candidate $\tilde{s}$, we identify the nearest species using

$$k^* = \arg\min_k D(\tilde{s}, \mu_k), \qquad D_{\min}(\tilde{s}) = D(\tilde{s}, \mu_{k^*}). \tag{4}$$

The candidate joins C_{k^*} if $D_{\min} \leq \delta_{\text{sp}}$. A candidate with $D_{\min} > \delta_{\text{sp}}$ is structurally eligible to form a new species and must additionally exhibit verified behavioral novelty.

3.3 Verified Novelty and Admission

Let $\mathcal{F}^g$ contain diagnostic tasks failed by at least one archived skill. We define the failures newly covered by $\tilde{s}$ as

$$\Delta\mathcal{F}(\tilde{s}; \mathcal{A}^g) = \left\{ \tau \in \mathcal{F}^g \;\middle|\; V_\tau(\xi(\tilde{s}, \tau)) = 1 \wedge \max_{s \in \mathcal{A}^g} V_\tau(\xi(s, \tau)) = 0 \right\}, \tag{5}$$

and compute normalized novelty as

$$N_{\text{ver}}(\tilde{s}; \mathcal{A}^g) = \frac{|\Delta\mathcal{F}(\tilde{s}; \mathcal{A}^g)|}{\max(1, |\mathcal{F}^g|)}. \tag{6}$$

Admission requires executability, a minimum quality level, and either within-species improvement or verified speciation:

$$\begin{aligned} \text{Admit}(\tilde{s}) = \mathbb{I}\big[& R(\tilde{s}) = 1 \wedge Q(\tilde{s}) \geq q_{\min} \\ & \wedge \big(\text{Improve}(\tilde{s}, C_{k^*}) \vee (D_{\min}(\tilde{s}) > \delta_{\text{sp}} \wedge N_{\text{ver}}(\tilde{s}; \mathcal{A}^g) \geq \nu_{\min})\big)\big]. \end{aligned} \tag{7}$$

Here, R indicates executability, and $q_{\min}$ and $\nu_{\min}$ are the quality and novelty thresholds. The operator Improve uses the within-species ranking defined in Section 3.5, which prevents textual rewrites and behaviorally redundant implementations from forming new species.

3.4 Abstraction-Changing Macro-Mutations

OESS samples a mutation from

$$\mathcal{M} = \{M_{\text{proc}}, M_{\text{encap}}, M_{\text{plan}}, M_{\text{verify}}, M_{\text{ctrl}}\}, \qquad \tilde{s} = M(s; \mathcal{X}), \quad M \sim p(M \mid \Phi(s), \mathcal{X}), \tag{8}$$

Method	Primitive	Macro	Monitor	Meta
LOCAL-MUTATION	72.1 ± 1.8	48.6 ± 2.5	39.8 ± 2.7	31.4 ± 2.2
BEST-OF-N	73.5 ± 1.5	51.2 ± 2.2	42.4 ± 2.4	33.1 ± 2.1
RANDOM-MACRO	73.0 ± 1.6	55.7 ± 2.0	45.3 ± 2.3	37.6 ± 2.4
NOVELTY-ONLY	71.8 ± 2.0	54.9 ± 2.4	47.1 ± 2.6	39.2 ± 2.5
TEXT-SPECIATION	73.4 ± 1.7	$\underline{58.0 \pm 2.1}$	$\underline{49.5 \pm 2.2}$	$\underline{41.0 \pm 2.3}$
OESS	$\mathbf{74.2 \pm 1.4}$	$\mathbf{62.8 \pm 1.8}^{\dagger}$	$\mathbf{55.9 \pm 2.0}^{\dagger}$	$\mathbf{48.7 \pm 2.1}^{\dagger}$

Method	Verified Species	Reuse (%)	Depth
LOCAL-MUTATION	11.6 ± 1.5	18.7 ± 2.1	2.2 ± 0.4
BEST-OF-N	13.8 ± 1.7	20.5 ± 2.4	2.4 ± 0.5
RANDOM-MACRO	18.9 ± 2.0	27.8 ± 2.7	3.1 ± 0.4
NOVELTY-ONLY	$\underline{25.7 \pm 2.4}$	31.4 ± 3.0	$\underline{3.8 \pm 0.4}$
TEXT-SPECIATION	22.4 ± 2.1	$\underline{35.6 \pm 2.6}$	3.5 ± 0.5
OESS	$\mathbf{31.8 \pm 2.2}^{\dagger}$	$\mathbf{44.1 \pm 2.5}^{\dagger}$	$\mathbf{4.6 \pm 0.5}^{\dagger}$

Table 1: Held-out success rates and abstraction statistics under matched search budgets. Bold and underline indicate the best and second-best results.

where $\mathcal{X}$ contains the parent's implementation, execution traces, verifier feedback, and accessible subskills. The mutation M_{proc} parameterizes recurring primitive sequences; M_{encap} extracts callable dependency subgraphs; M_{plan} separates goal decomposition from execution; and M_{verify} extracts repeated checks and repairs into monitors.

The mutation M_{ctrl} converts a linear workflow into a controller $\kappa = (\mathcal{Z}, z_0, \Gamma, \Omega)$, whose state and action updates are $z_{t+1} = \Gamma(z_t, o_t, e_t)$ and $a_t = \Omega(z_t, o_t)$, respectively, where e_t denotes verifier or monitor feedback. The resulting controller supports branching, retry, recovery, and stopping decisions. Each mutation must preserve the declared interface or introduce a checkable typed interface. OESS evaluates every offspring using the same admission rule.

3.5 SPECIES-AWARE SELECTION AND LINEAGE

Within species C_k, OESS ranks skills by

$$U_{\text{in}}(s) = Q(s) + \lambda_{\text{cov}} N_{\text{ver}}(s; \mathcal{A}^g) - \lambda_{\text{cost}} \widehat{C}(s), \tag{9}$$

where $\widehat{C}(s) \in [0, 1]$ is the normalized execution cost and both coefficients are non-negative. For search across species, OESS uses

$$U_{\text{sp}}(C_k^g) = \max_{s \in C_k^g} U_{\text{in}}(s) + \beta_{\text{nov}} \frac{1}{K_g - 1} \sum_{\ell \neq k} D(\mu_k, \mu_\ell) + \beta_{\text{exp}} \sqrt{\frac{\log(1+g)}{1+n_k^g}}, \tag{10}$$

where n_k^g is the number of times species C_k^g has been selected. The inter-species term is defined as zero when $K_g = 1$. OESS samples a species using a softmax over U_{sp} and then samples a parent using U_{in}. This procedure makes related skills compete locally and assigns every surviving species a non-zero probability of producing descendants.

For each admitted offspring, the lineage graph stores the tuple $(s, \tilde{s}, M)$. Species-level capacity constraints remove low-ranked redundant members while retaining representative and lineage-critical skills. The archive-level objective is

$$\max_{\mathcal{A}} \quad \sum_{C_k \in \mathcal{C}} \max_{s \in C_k} Q(s) + \eta \sum_{k < \ell} D(\mu_k, \mu_\ell) + \rho \left| \bigcup_{s \in \mathcal{A}} \Delta\mathcal{F}(s; \mathcal{A}^0) \right|, \tag{11}$$

subject to executability and archive-capacity constraints. The discrete search procedure optimizes the resulting trade-off among local quality, species diversity, and verified behavioral coverage.

Variant	Avg. Success	Species	Survival	Invalid Novelty
Full OESS	$\mathbf{60.4 \pm 1.5}$	$\mathbf{31.8 \pm 2.2}$	$\mathbf{68.7 \pm 3.1}$	$\mathbf{7.8 \pm 1.4}$
No species protection	55.8 ± 1.8	21.3 ± 2.1	42.6 ± 3.8	9.4 ± 1.6
No macro mutation	54.9 ± 1.9	19.5 ± 1.8	49.1 ± 3.5	8.7 ± 1.5
Text distance	55.5 ± 1.7	22.4 ± 2.1	51.8 ± 3.4	18.9 ± 2.0
No behav. verification	53.6 ± 2.4	$\underline{36.9 \pm 3.7}$	39.7 ± 4.6	43.5 ± 4.2
No lineage retention	57.1 ± 1.6	25.6 ± 2.0	55.2 ± 3.6	10.1 ± 1.7

Table 2: Ablations averaged across task families. Success, survival, and invalid novelty are percentages. Abbreviations: Avg. = Average; behav. = behavioral.

Method	Seen Setting	Unseen Tasks	New Model	New Harness	Perturbed Env.	Normalized Cost
LOCAL-MUTATION	48.0 ± 1.7	42.7 ± 2.0	44.1 ± 1.9	40.8 ± 2.2	35.9 ± 2.4	$\mathbf{1.00}$
BEST-OF-N	50.3 ± 1.8	44.0 ± 2.1	45.5 ± 2.0	42.4 ± 2.3	37.2 ± 2.5	1.02
NOVELTY-ONLY	53.3 ± 2.0	47.6 ± 2.3	49.8 ± 2.1	45.1 ± 2.4	39.6 ± 2.7	$\underline{1.07}$
OESS	$\mathbf{60.4 \pm 1.5}$	$\mathbf{55.2 \pm 1.9}$	$\mathbf{56.8 \pm 1.8}$	$\mathbf{51.7 \pm 2.1}$	$\mathbf{43.1 \pm 2.5}$	1.13

Table 3: Success after transferring frozen skill libraries. Cost includes search-time orchestration overhead and is normalized by LOCAL-MUTATION. Abbreviation: Env. = Environment.

4 EXPERIMENTS

We evaluate whether OESS discovers reusable skill abstractions that extend beyond variations of a fixed representation.

4.1 SETUP

The benchmark comprises four task families. Primitive tasks require short sequences of tool calls. Macro tasks require reusable procedures for dependency inspection, coordinated modification, and validation. Monitor tasks inject recoverable failures that require detection, diagnosis, and repair. Meta tasks require selecting, composing, or revising skills based on their signatures and execution histories. Each family contains 120 search, 60 validation, and 120 test tasks drawn from disjoint repositories, interfaces, and failure templates. We additionally hold out one generator per family to evaluate transfer to unseen task structures.

Each search run uses 100 generations, 24 candidate evaluations per generation, and five independent seeds. All methods receive identical budgets for environment executions, model calls, and verifier queries. OESS represents each candidate using its executable specification, input-output signature, tool-dependency graph, temporal abstraction level, failure-repair profile, and execution traces. A candidate forms a new species only if it is structurally distant from existing representatives and reproduces newly covered behavior on three independently instantiated tasks.

We compare OESS with LOCAL-MUTATION, which applies local edits to a single incumbent; BEST-OF-N, which retains the highest-scoring candidate in each generation; RANDOM-MACRO, which uniformly samples the same macro-mutations without preserving species; and NOVELTY-ONLY, which archives behaviorally novel candidates without requiring verified failure coverage. TEXT-SPECIATION replaces the structured skill distance with embedding similarity between skill documents.

The primary metric is held-out success rate. We also report the number of verified species that cover previously unsolved failures, ten-generation species survival, cross-task reuse, maximum verified call-graph depth, and composition gain over the strongest individual skill. Invalid novelty measures the proportion of archived candidates that appear novel but cannot reproduce the claimed behavior. We perform a paired bootstrap over tasks and seeds with 10,000 resamples; † denotes $p < 0.05$.

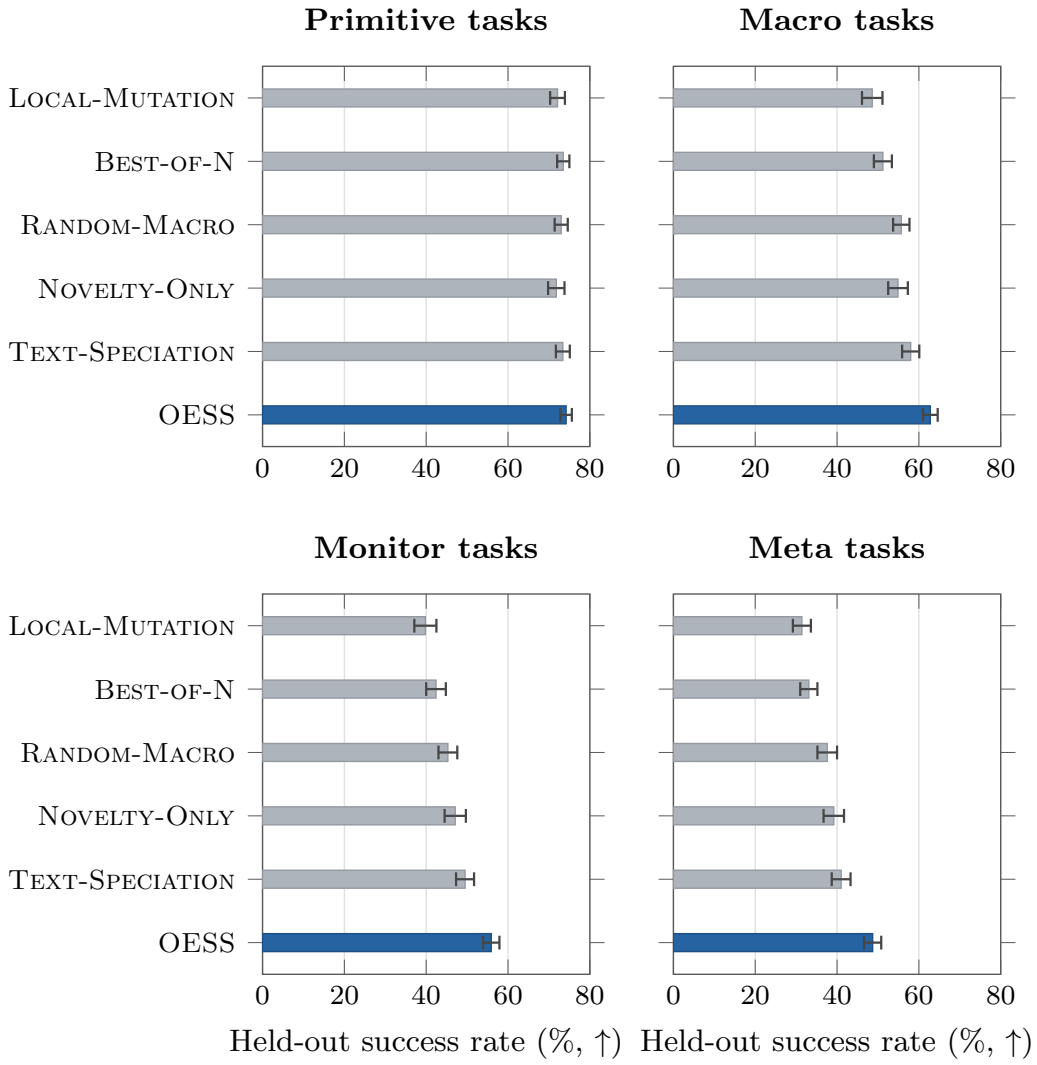

Figure 3: Held-out success rates under matched search budgets. Bars and error bars report the mean and standard deviation across five independent seeds. OESS is highlighted in blue.

4.2 PERFORMANCE AND ABSTRACTION DISCOVERY

Table 1 and Figure 3 show that OESS achieves its largest gains on tasks requiring abstractions beyond local action sequences. Compared with LOCAL-MUTATION, OESS improves success by 2.1 points on primitive tasks and by 14.2, 16.1, and 17.3 points on macro, monitor, and meta tasks, respectively. The performance gap increases when tasks require reusable procedures, feedback-driven recovery, or skill composition. BEST-OF-N evaluates the same number of candidates but trails OESS by 11.6 points on monitor tasks and 15.6 points on meta tasks.

Macro-mutations improve higher-level performance when combined with species-aware selection. RANDOM-MACRO increases call-graph depth from 2.2 to 3.1 and outperforms LOCAL-MUTATION on all three higher-level task families, but remains 7.1 to 11.1 points below OESS. NOVELTY-ONLY discovers 25.7 verified species and reaches a depth of 3.8, but trails OESS by 8.8 points on monitor tasks and 9.5 points on meta tasks.

Archive statistics associate the performance gains with reusable hierarchical behavior. OESS yields 31.8 verified species, 44.1% cross-task reuse, and a maximum verified depth of 4.6. Composing its

evolved skills improves success by 9.3 ± 1.4 points over the best individual skill, compared with 4.1 ± 1.0 points for RANDOM-MACRO and 5.6 ± 1.2 points for NOVELTY-ONLY.

4.3 COMPONENT ANALYSIS

The ablations in Table 2 isolate the effects of structural exploration, species protection, and behavioral verification. Removing species protection reduces average success by 4.6 points and survival by 26.1 points. Removing macro-mutations reduces average success by 5.5 points and hierarchical depth from 4.6 to 2.9, showing that local revisions rarely create new temporal interfaces.

Behavioral verification separates functional innovation from superficial diversity. Removing verification increases the number of nominal species from 31.8 to 36.9, while invalid novelty rises from 7.8% to 43.5%, survival falls to 39.7%, and average success decreases by 6.8 points. Replacing structured distance with text distance increases invalid novelty to 18.9% and reduces success by 4.9 points because textual descriptions can conceal differences in tool dependencies and repair behavior. Removing lineage retention reduces survival from 68.7% to 55.2% and reuse from 44.1% to 37.0%, indicating that preserved ancestry helps the search revisit productive intermediate abstractions.

4.4 TRANSFER AND STEPPING STONES

Table 3 evaluates frozen skill libraries without further evolution. Compared with LOCAL-MUTATION, OESS improves success by 12.5 points on unseen tasks, 12.7 points with a substituted model, and 10.9 points under a new execution harness. Under randomized latency, renamed interface elements, and delayed verifier feedback, OESS retains a 7.2-point advantage. The smaller gain under environment perturbations indicates residual dependence on timing, error vocabulary, and harness conventions.

Lineage analysis characterizes the role of species protection. Across five runs, 74 verified transitions produce descendants that solve failures unsolved by every ancestor. Of these transitions, 28% follow an immediately improving parent, while 61% pass through at least one neutral or temporarily worse ancestor. Removing an intermediate node and replaying subsequent mutations recovers the final behavior in 34% of cases. Many successful lineages first encapsulate a repeated procedure, then expose its termination state for monitoring, and subsequently extract a planner that composes subskills. The main results, ablations, transfer tests, and lineage analysis jointly show that macro-mutations make abstraction changes reachable, speciation allows intermediate abstractions to mature, and verification ties persistent novelty to reproducible behavior.

5 CONCLUSION

We presented OESS, a species-aware evolutionary framework for discovering behaviorally useful skill abstractions through structured distance, verified species formation, abstraction-changing macro-mutations, and lineage preservation. Evaluations across skill levels established that OESS produces durable structural innovations, expands coverage of previously unsolved failures, and improves reuse, composition, hierarchy formation, and transfer. Comparisons under matched budgets confirmed the advantages of species-aware verified search, while ablation and lineage analyses identified the mechanisms and intermediate transitions responsible for these gains. These results validate speciation as an effective foundation for open-ended skill abstraction discovery.

LIMITATIONS

The structured skill representation relies on manually defined abstraction categories and distance components, which may not capture every behaviorally meaningful distinction among evolving skills. Moreover, the framework introduces several interacting thresholds and weighting coefficients whose optimal values may vary across domains and stages of evolution. Finally, although lineage preservation supports compositional development, maintaining an expanding archive can increase computational and storage overhead over long runs. Future work could explore adaptive representations, automatic hyperparameter adjustment, and more efficient archive management.

AI Assistance Transparency Statement

To ensure transparency, the authors declare that AI language tools were used in a narrow and well-defined capacity during the preparation of this work. Specifically, such tools assisted with editorial refinement—including restructuring sentences for improved readability, eliminating grammatical inconsistencies, and clarifying technical phrasing. The intellectual substance of this paper, comprising the open-ended skill speciation mechanism, the abstraction evolution methodology, the agent skill library design, and all experimental analyses, was developed entirely by the human research team. AI assistance was not sought for any aspect of the scientific process, including problem formulation, theoretical derivation, experimental execution, or result interpretation. All AI-proposed changes were individually reviewed and approved by the authors. The authors hold full responsibility for the correctness and originality of the research presented herein.

References

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. *arXiv preprint arXiv:1609.05140*, 2016. URL https://arxiv.org/abs/1609.05140.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023. URL https://arxiv.org/abs/2305.17126.

Antoine Cully. Multi-emitter map-elites: Improving quality, diversity and convergence speed with heterogeneous sets of emitters. *arXiv preprint arXiv:2007.05352*, 2020. URL https://arxiv.org/abs/2007.05352.

M. López de Haro, A. Santos, and S. B. Yuste. Simple equation of state for hard disks on the hyperbolic plane. *arXiv preprint arXiv:0809.3090*, 2008. URL https://arxiv.org/abs/0809.3090.

Thomas G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition, 1999. URL https://arxiv.org/abs/cs/9905014.

Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=SJx63jRqFm.

Matthew C. Fontaine, Julian Togelius, Stefanos Nikolaidis, and Amy K. Hoover. Covariance matrix adaptation for the rapid illumination of behavior space. *arXiv preprint arXiv:1912.02400*, 2019. URL https://arxiv.org/abs/1912.02400.

Matthew Christopher Fontaine and Stefanos Nikolaidis. Differentiable quality diversity. In *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=uJGObgFU0lU.

Caroline S. Gorham and David E. Laughlin. On the formation of solid states beyond perfect crystals: Quasicrystals, geometrically-frustrated crystals and glasses. *arXiv preprint arXiv:1907.08839*, 2019. URL https://arxiv.org/abs/1907.08839.

Karol Gregor, Danilo Jimenez Rezende, and Daan Wierstra. Variational intrinsic control. *arXiv preprint arXiv:1611.07507*, 2016. URL https://arxiv.org/abs/1611.07507.

Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. *arXiv preprint arXiv:2010.03934*, 2020. URL https://arxiv.org/abs/2010.03934.

Tejas D. Kulkarni, Karthik R. Narasimhan, Ardavan Saeedi, and Joshua B. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *arXiv preprint arXiv:1604.06057*, 2016. URL https://arxiv.org/abs/1604.06057.

So Kuroki, Taishi Nakamura, Takuya Akiba, and Yujin Tang. Agent skill acquisition for large language models via cycleqd. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=Kvdh12wGC0.

Michael Laskin, Hao Liu, Xue Bin Peng, Denis Yarats, Aravind Rajeswaran, and Pieter Abbeel. Cic: Contrastive intrinsic control for unsupervised skill discovery. *arXiv preprint arXiv:2202.00161*, 2022. URL https://arxiv.org/abs/2202.00161.

Andrew Levy, George Konidaris, Robert Platt, and Kate Saenko. Learning multi-level hierarchies with hindsight. *arXiv preprint arXiv:1712.00948*, 2017. URL https://arxiv.org/abs/1712.00948.

Hao Liu and Pieter Abbeel. Aps: Active pretraining with successor features. *arXiv preprint arXiv:2108.13956*, 2021. URL https://arxiv.org/abs/2108.13956.

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.

Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *arXiv preprint arXiv:1805.08296*, 2018. URL https://arxiv.org/abs/1805.08296.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. In *Advances in Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=tBRNC6YemY.

Thomas Pierrot and Arthur Flajolet. Evolving populations of diverse rl agents with map-elites. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=CBfYffLqWqb.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023. URL https://arxiv.org/abs/2302.04761.

Archit Sharma, Shixiang Gu, Sergey Levine, Vikash Kumar, and Karol Hausman. Dynamics-aware unsupervised discovery of skills. *arXiv preprint arXiv:1907.01657*, 2019. URL https://arxiv.org/abs/1907.01657.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R. Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=vAElhFcKW6.

Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies, 2002. URL https://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning, 1999. URL https://doi.org/10.1016/S0004-3702(99)00052-1.

István Szépkúti. Multidimensional or relational? how to organize an on-line analytical processing database. *arXiv preprint arXiv:1103.3863*, 2011. URL https://arxiv.org/abs/1103.3863.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1703.01161*, 2017. URL https://arxiv.org/abs/1703.01161.

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

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.

Rui Wang, Joel Lehman, Aditya Rawal, Jiale Zhi, Yulun Li, Jeff Clune, and Kenneth O. Stanley. Enhanced poet: Open-ended reinforcement learning through unbounded invention of learning challenges and their solutions. *arXiv preprint arXiv:2003.08536*, 2020. URL https://arxiv.org/abs/2003.08536.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv:2302.01560*, 2023. URL https://arxiv.org/abs/2302.01560.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022. URL https://arxiv.org/abs/2210.03629.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R. Fung, Hao Peng, and Heng Ji. Craft: Customizing llms by creating and retrieving from specialized toolsets. *arXiv preprint arXiv:2309.17428*, 2023. URL https://arxiv.org/abs/2309.17428.

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

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.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=njwv9BsGHF.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, Yu Qiao, Zhaoxiang Zhang, and Jifeng Dai. Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *arXiv preprint arXiv:2305.17144*, 2023. URL https://arxiv.org/abs/2305.17144.

APPENDIX

A GENERATION-WISE ARCHIVE, SPECIES, AND LINEAGE UPDATES

This section defines the generation-wise state transition of OESS by integrating species-aware selection, macro-mutation, execution-based evaluation, verified admission, species assignment, medoid recomputation, and lineage recording into a discrete search procedure.

The search state at generation g is

$$\mathcal{S}^g = \big(\mathcal{A}^g, \mathcal{C}^g, \mathcal{L}^g, \{n_k^g\}_{k=1}^{K_g}\big),$$

where $\mathcal{A}^g$ is the skill archive, $\mathcal{C}^g = \{C_k^g\}_{k=1}^{K_g}$ is its species partition, $\mathcal{L}^g$ is the directed lineage graph, and n_k^g is the number of times species C_k^g has been selected. Each archived skill belongs to exactly one species:

$$\mathcal{A}^g = \bigcup_{k=1}^{K_g} C_k^g, \qquad C_k^g \cap C_\ell^g = \varnothing \quad \text{for } k \neq \ell.$$

The representative of each nonempty species is its executable medoid:

$$\mu_k^g = \arg\min_{s \in C_k^g} \sum_{s' \in C_k^g} D(s, s').$$

A.1 SELECTION AND CANDIDATE CONSTRUCTION

Given the current state, OESS selects species C_k^g according to the softmax of its species utility:

$$P_{\mathrm{sp}}^g(k) = \frac{\exp(U_{\mathrm{sp}}(C_k^g))}{\sum_{\ell=1}^{K_g} \exp(U_{\mathrm{sp}}(C_\ell^g))}. \tag{12}$$

Conditioned on species C_k^g, OESS samples a parent according to the within-species utility:

$$P_{\text{par}}^g(s \mid k) = \frac{\exp(U_{\text{in}}(s))}{\sum_{s' \in C_k^g} \exp(U_{\text{in}}(s'))}, \qquad s \in C_k^g. \tag{13}$$

The selected parent s defines the mutation context $\mathcal{X}(s)$, which contains its implementation, execution traces, verifier feedback, and accessible subskills. OESS then samples a mutation $M \in \mathcal{M}$ from $p(M \mid \Phi(s), \mathcal{X}(s))$ and constructs the offspring as $\tilde{s} = M(s; \mathcal{X}(s))$.

OESS executes the offspring on its diagnostic task set $\mathcal{T}_{\tilde{s}}$. The resulting executions determine its quality $Q(\tilde{s})$, executability indicator $R(\tilde{s})$, behavioral descriptor component $\phi_{\text{beh}}(\tilde{s})$, failure-repair component $\phi_{\text{repair}}(\tilde{s})$, and verified novelty $N_{\text{ver}}(\tilde{s}; \mathcal{A}^g)$. These quantities, together with the statically available interface, temporal, and tool-dependency components, complete $\Phi(\tilde{s})$ before nearest-species matching and admission.

A.2 Admission and Species Assignment

Let

$$k^* = \arg\min_{1 \le k \le K_g} D(\tilde{s}, \mu_k^g), \qquad D_{\min}(\tilde{s}) = D(\tilde{s}, \mu_{k^*}^g).$$

The admission predicate uses the pre-update archive $\mathcal{A}^g$, which also serves as the reference set for computing the candidate's novelty. If the candidate is rejected, the archive, species partition, and lineage graph remain unchanged; only the selection count of the sampled species is updated.

If the candidate is admitted and satisfies

$$D_{\min}(\tilde{s}) > \delta_{\text{sp}} \quad \wedge \quad N_{\text{ver}}(\tilde{s}; \mathcal{A}^g) \ge \nu_{\min}, \tag{14}$$

it initializes a singleton species $C_{K_g+1} = \{\tilde{s}\}$ with medoid $\tilde{s}$. Otherwise, the admitted candidate is assigned to its nearest species $C_{k^*}^g$. Admission is therefore determined before an admitted offspring either creates a verified species or competes within an existing species.

After insertion, OESS applies species-level capacity constraints using the within-species ranking. Archive maintenance can remove redundant low-ranked members while retaining medoids and lineage-critical skills as required. Empty species are deleted, the remaining species are reindexed, and their medoids are recomputed from the surviving executable members. The lineage edge $(s, \tilde{s}, M)$ is recorded exactly when $\tilde{s}$ remains in the maintained archive.

The procedure terminates after G generations. At generation g, candidate evaluation, novelty computation, and admission use the same pre-update archive $\mathcal{A}^g$. Archive maintenance begins only after admission and species assignment have been determined.

B Environment Details

B.1 Datasets and Task Settings

We construct a verifier-backed benchmark with four task families, organized by the level of abstraction required for successful execution. PRIMITIVE tasks require sequences of one to three tool calls for operations such as file inspection, structured lookup, and localized editing. MACRO tasks require reusable multi-step procedures for dependency analysis, coordinated modification, and post-execution validation. MONITOR tasks inject recoverable failures, including malformed tool outputs, transient execution errors, and incomplete modifications, and require agents to detect, diagnose, and repair them. META tasks require agents to select, compose, or revise multiple skills based on their typed signatures, dependency relations, and prior execution traces.

Each family contains 120 search tasks, 60 validation tasks, and 120 held-out test tasks. Search tasks provide behavioral feedback during evolution. The validation split determines global hyperparameters and archive checkpoints, and the test split is inaccessible until final evaluation. We partition repositories, interface schemas, initial states, and failure templates before task instantiation, ensuring that each repository and exact failure template occurs in only one split. We also reserve one task generator from each family for unseen-task transfer evaluation. Its templates and generated instances are excluded from search and validation.

Algorithm 1 Generation-wise OESS search and archive transition

Require: Initial archive $\mathcal{A}^0$; initial species partition $\mathcal{C}^0$; task distribution $p(\tau)$; executable verifiers $\{V_\tau\}$; executor $\mathcal{E}$; mutation family $\mathcal{M}$; mutation distributions $p(M \mid \Phi(s), \mathcal{X})$; number of generations G; thresholds $q_{\min}$, $\nu_{\min}$, and δ_{sp}; species-level and archive-level capacity constraints

Ensure: Final archive $\mathcal{A}^G$, species partition $\mathcal{C}^G$, and lineage graph $\mathcal{L}^G$

Initialize $\mathcal{L}^0 \leftarrow \varnothing$
Set $n_k^0 \leftarrow 0$ for every $C_k^0 \in \mathcal{C}^0$
Compute the medoid of every species in $\mathcal{C}^0$
for $g = 0, \ldots, G-1$ **do**
  Construct the current diagnostic failure set $\mathcal{F}^g$ from verifier outcomes of archived skills
  Compute $U_{\mathrm{in}}(s)$ for every $s \in \mathcal{A}^g$
  Compute $U_{\mathrm{sp}}(C_k^g)$ for $k = 1, \ldots, K_g$
  Sample $k \sim P_{\mathrm{sp}}^g$ using Equation (12)
  Sample $s \sim P_{\mathrm{par}}^g(\cdot \mid k)$ using Equation (13)
  Set $n_k^g \leftarrow n_k^g + 1$
  Form $\mathcal{X}(s)$ from the parent implementation, traces, verifier feedback, and accessible subskills
  Sample $M \sim p(M \mid \Phi(s), \mathcal{X}(s))$
  Construct $\tilde{s} \leftarrow M(s; \mathcal{X}(s))$
  Execute $\xi(\tilde{s}, \tau) \leftarrow \mathcal{E}(\tilde{s}, \tau)$ for every $\tau \in \mathcal{T}_{\tilde{s}}$
  Evaluate $V_\tau(\xi(\tilde{s}, \tau))$ and compute $R(\tilde{s})$, $Q(\tilde{s})$, and $\Phi(\tilde{s})$
  Compute $\Delta\mathcal{F}(\tilde{s}; \mathcal{A}^g)$ and $N_{\mathrm{ver}}(\tilde{s}; \mathcal{A}^g)$
  Find $k^* \leftarrow \arg\min_j D(\tilde{s}, \mu_j^g)$ and set $D_{\min}(\tilde{s}) \leftarrow D(\tilde{s}, \mu_{k^*}^g)$
  Evaluate

$$b \leftarrow \mathbb{I}[R(\tilde{s}) = 1 \wedge Q(\tilde{s}) \geq q_{\min} \wedge (\mathrm{Improve}(\tilde{s}, C_{k^*}^g) \vee (D_{\min}(\tilde{s}) > \delta_{\mathrm{sp}} \wedge N_{\mathrm{ver}}(\tilde{s}; \mathcal{A}^g) \geq \nu_{\min}))]$$

  Initialize $\mathcal{A}^{g+1} \leftarrow \mathcal{A}^g$, $\mathcal{C}^{g+1} \leftarrow \mathcal{C}^g$, and $\mathcal{L}^{g+1} \leftarrow \mathcal{L}^g$
  if $b = 1$ **then**
    Insert $\tilde{s}$ into $\mathcal{A}^{g+1}$
    if $D_{\min}(\tilde{s}) > \delta_{\mathrm{sp}}$ **and** $N_{\mathrm{ver}}(\tilde{s}; \mathcal{A}^g) \geq \nu_{\min}$ **then**
      Create $C_{K_g+1}^{g+1} \leftarrow \{\tilde{s}\}$ with medoid $\mu_{K_g+1}^{g+1} \leftarrow \tilde{s}$
      Initialize $n_{K_g+1}^{g+1} \leftarrow 0$
    else
      Insert $\tilde{s}$ into $C_{k^*}^{g+1}$
    end if
    Apply the stated species-level and archive-level capacity constraints
    Remove from $\mathcal{C}^{g+1}$ any skills removed from $\mathcal{A}^{g+1}$, and delete empty species
    if $\tilde{s} \in \mathcal{A}^{g+1}$ **then**
      Add lineage edge $(s, \tilde{s}, M)$ to $\mathcal{L}^{g+1}$
    end if
    Recompute every affected species medoid
  end if
  Carry all updated selection counts into generation $g + 1$
end for
return $\mathcal{A}^G$, $\mathcal{C}^G$, and $\mathcal{L}^G$

Each task specifies an initial sandbox state, a natural-language instruction, an allowed tool set, a maximum execution budget, and a deterministic executable verifier. The agent returns an executable skill artifact and its declared input–output signature. The verifier determines task success from the resulting environment state and execution trace, without evaluating free-form textual answers. Skills invoke shell, file-system, structured-data, and repository-analysis tools through typed interfaces. Primitive, macro, monitor, and meta tasks allow at most 8, 20, 30, and 40 tool calls, respectively, with a wall-clock timeout of 300 seconds per execution.

All methods use the same system prompt, which specifies the available tools, interface contracts, sandbox restrictions, and requirement to produce an executable artifact. Candidate-generation prompts contain the parent implementation, its typed signature, relevant execution traces, verifier feedback,

and the selected mutation operator. They exclude test examples, test-derived execution traces, and test verifier outputs. To reduce contamination and memorization risks, we generate benchmark instances from private templates and newly instantiated repositories, randomize identifiers across splits, and remove exact or near-duplicate task and repository matches using normalized instruction hashes and token-level MinHash similarity with a threshold of 0.85.

B.2 Preprocessing and Evaluation

Before execution, we normalize task instructions to UTF-8, canonicalize line endings, and replace environment-specific absolute paths with sandbox-relative paths. We parse each skill document into an executable implementation, a typed input–output declaration, and optional metadata describing dependencies, termination conditions, and recovery behavior. Candidates receive zero task reward if they fail to parse, reference unavailable tools, violate the declared interface, exceed the execution budget, or terminate without producing a checkable output.

Verifier decisions are binary. A task succeeds only if the candidate terminates within the resource limits and satisfies all task-specific assertions. These assertions cover output correctness, required state changes, preservation of protected files, interface conformance, and the absence of unresolved execution errors. For tasks with multiple valid solutions, the verifier checks semantic equivalence through canonicalized structured outputs and environment-state predicates, without comparing implementation text. Numeric fields are parsed after the removal of insignificant whitespace and formatting tokens. Integers are compared exactly, while floating-point values use absolute and relative tolerances of 10^{-6}. Lists and mappings are canonicalized when the task specification declares their ordering irrelevant.

The primary metric is the held-out task success rate, averaged within each task family. We also report: (i) the number of verified species that solve at least one failure unsolved by the preceding archive; (ii) species survival, defined as the fraction of newly admitted species retaining an archived descendant after ten generations; (iii) cross-task reuse, defined as the percentage of successful test executions that invoke a skill evolved on a different search task; (iv) the maximum verified call-graph depth; (v) the composition gain over the strongest individual skill; and (vi) invalid novelty, defined as the fraction of archived candidates whose claimed novel behavior cannot be reproduced on three independently instantiated diagnostic tasks. For transfer evaluation, we freeze the final skill archive and disable further mutation and archive updates.

We run each method with five independent seeds, $\{0, 1, 2, 3, 4\}$, and report the mean and sample standard deviation across seeds. We assess statistical significance using a paired bootstrap over tasks and seeds with 10,000 resamples. Differences with two-sided $p < 0.05$ are marked with $\dagger$. All methods receive equal budgets for environment executions, language-model calls, verifier queries, and maximum tool calls. We evaluate the following baselines: Local-Mutation, Best-of-N, Random-Macro, Novelty-Only, and Text-Speciation. The ablations remove species protection, macro-mutations, behavioral verification, or lineage retention, or replace the structured species distance with the cosine distance between skill-document embeddings.

B.3 Implementation and Hardware

The main experiments use `Qwen2.5-Coder-32B-Instruct` as the frozen agent backbone. We deploy the model locally for parent-conditioned candidate generation, implementation revision, macro-mutation realization, and tool-augmented task execution. OESS and all baselines use the same model and decoding configuration. Deterministic task verifiers provide quality and admission signals. The evaluation uses no learned value model, reward model, or LLM-based judge. Text-Speciation embeds normalized skill documents with `BAAI/bge-large-en-v1.5` and computes cosine distance. For new-model transfer, we replace the frozen executor with `Llama-3.1-70B-Instruct` and retain the evolved skill library without adaptation.

Candidate generation uses a temperature of 0.7, nucleus sampling with $p = 0.95$, a maximum of 4,096 generated tokens, and a repetition penalty of 1.05. Verifier-facing replay is deterministic, with temperature 0, and each claimed novel behavior is reproduced on three independently instantiated tasks. Model responses are truncated only at complete tool-call or code-block boundaries. Failed generations are not resampled beyond the candidate budget.

Method	25 Gen.	50 Gen.	75 Gen.	100 Gen.	Ver. Species	Reuse (%)
Local-Mutation	43.7 ± 1.9	46.1 ± 1.8	47.3 ± 1.7	48.0 ± 1.7	11.6 ± 1.5	18.7 ± 2.1
Random-Macro	45.8 ± 2.1	49.4 ± 1.9	51.1 ± 1.8	52.9 ± 1.7	18.9 ± 2.0	27.8 ± 2.7
Novelty-Only	46.9 ± 2.3	50.2 ± 2.2	52.1 ± 2.1	53.3 ± 2.0	25.7 ± 2.4	31.4 ± 3.0
Text-Speciation	47.2 ± 2.0	51.6 ± 1.9	54.2 ± 1.8	55.5 ± 1.7	22.4 ± 2.1	35.6 ± 2.6
OESS	$\mathbf{49.1 \pm 1.8}$	$\mathbf{54.8 \pm 1.7}$	$\mathbf{58.1 \pm 1.6}$	$\mathbf{60.4 \pm 1.5}$	$\mathbf{31.8 \pm 2.2}$	$\mathbf{44.1 \pm 2.5}$

Table 4: Average held-out success under different search budgets. Results report the mean and standard deviation across five seeds. Generation 100 corresponds to the budget used in the main experiments. Abbreviations: Gen. = Generation; Ver. = Verified.

Each search run performs 100 generations with 24 candidate evaluations per generation, yielding 2,400 evaluated offspring per seed and method. OESS first samples a species using a softmax temperature of 0.5, then samples parents within that species using a softmax temperature of 0.3. The five macro-mutations—procedural parameterization, subskill encapsulation, planner extraction, verifier or monitor extraction, and controller construction—are sampled uniformly from the operators compatible with the parent's abstraction level. Each offspring can introduce at most four new subskills, and the verified call graph has a maximum depth of six. Controller executions allow at most three retries per recoverable failure and 40 total tool calls.

The structured distance uses weights

$$(w_{\mathrm{io}}, w_{\mathrm{temp}}, w_{\mathrm{tool}}, w_{\mathrm{repair}}, w_{\mathrm{beh}}) = (0.15, 0.20, 0.25, 0.15, 0.25).$$

The species threshold is $\delta_{\mathrm{sp}} = 0.35$, the minimum admission quality is $q_{\min} = 0.20$, and the verified-novelty threshold is $\nu_{\min} = 0.02$. Species formation additionally requires at least one newly covered diagnostic failure. Within-species selection uses $\lambda_{\mathrm{cov}} = 0.5$ and $\lambda_{\mathrm{cost}} = 0.1$, while species-level selection uses $\beta_{\mathrm{nov}} = 0.25$ and $\beta_{\mathrm{exp}} = 0.10$. We select these values once on the validation split and fix them across all task families, baselines, ablations, and transfer experiments.

The archive stores at most 256 executable skills and 16 members per species. When either limit is exceeded, candidates are removed in ascending order of within-species utility, while species medoids, current species elites, and lineage-critical nodes with surviving descendants are preserved. Behavioral descriptors are computed from up to 32 stratified execution traces per skill. Tool-dependency distance uses normalized graph-edit distance. Failure-repair distance uses Jaccard distance over canonicalized failure and repair labels. Behavioral distance combines action-sequence edit distance with disagreement over verifier outcomes and termination decisions.

Experiments run in isolated Docker containers with 16 virtual CPU cores, 64 GB RAM, and a 20 GB writable workspace per task. Language-model inference is served with `vLLM` using bfloat16 precision and tensor parallelism across four NVIDIA A100 80 GB GPUs. Search workers execute asynchronously, and archive updates and candidate admission are serialized to ensure deterministic results for a fixed seed. We use Python 3.11, PyTorch 2.4, CUDA 12.1, and vLLM 0.6.2. During task execution, network access is disabled and process creation is restricted to the sandbox. Each run records prompts, model outputs, tool calls, verifier results, archive updates, mutation operators, random seeds, and lineage edges for reproducibility.

C Additional Experimental Results

C.1 Search-Budget Scaling

As shown in Table 4, the average held-out success of OESS increases from 49.1 at 25 generations to 60.4 at 100 generations. At 100 generations, OESS achieves an average held-out success of 60.4, compared with 55.5 for Text-Speciation, 53.3 for Novelty-Only, 52.9 for Random-Macro, and 48.0 for Local-Mutation. OESS also yields 31.8 verified species and a cross-task reuse rate of 44.1%.

C.2 Sensitivity to Speciation and Novelty Thresholds

Table 5 reports the results for different species-distance and verified-novelty thresholds. With $\delta_{\mathrm{sp}} = 0.20$ and $\nu_{\min} = 0.02$, the method produces 37.4 verified species and an invalid-novelty rate

δ_{sp}	$\nu_{\min}$	Avg. Success	Verified Species	Species Survival (%)	Invalid Novelty (%)	Depth
0.20	0.02	57.6 ± 1.8	37.4 ± 2.8	59.8 ± 3.6	14.9 ± 2.1	4.5 ± 0.5
0.20	0.04	58.7 ± 1.7	34.2 ± 2.5	64.1 ± 3.3	10.8 ± 1.8	4.5 ± 0.4
0.30	0.02	59.3 ± 1.6	34.9 ± 2.4	65.2 ± 3.2	10.1 ± 1.7	4.6 ± 0.5
0.30	**0.04**	$\mathbf{60.4 \pm 1.5}$	31.8 ± 2.2	$\mathbf{68.7 \pm 3.1}$	7.8 ± 1.4	$\mathbf{4.6 \pm 0.5}$
0.30	0.08	58.9 ± 1.7	25.6 ± 2.0	70.3 ± 3.0	$\mathbf{5.9 \pm 1.2}$	4.2 ± 0.4
0.40	0.04	58.5 ± 1.8	26.7 ± 2.1	69.5 ± 3.2	6.7 ± 1.3	4.1 ± 0.5
0.40	0.08	56.8 ± 1.9	20.9 ± 1.8	71.0 ± 2.9	5.4 ± 1.1	3.8 ± 0.4

Table 5: Sensitivity to the species-distance threshold δ_{sp} and verified-novelty threshold $\nu_{\min}$. Results are averaged across task families and five seeds.

Operator	Prop. (%)	Adm. (%)	New Sp. (%)	Fail. Cov. (%)	Desc. Gain	Med. Surv.
Procedural parameterization	23.4 ± 2.1	31.7 ± 2.8	12.9 ± 1.7	9.8 ± 1.4	2.1 ± 0.6	18.6 ± 2.7
Subskill encapsulation	21.1 ± 1.9	28.9 ± 2.5	18.7 ± 2.0	13.6 ± 1.6	3.4 ± 0.7	24.8 ± 3.1
Planner extraction	18.5 ± 1.8	22.6 ± 2.2	24.1 ± 2.3	17.9 ± 1.9	4.8 ± 0.9	29.7 ± 3.4
Monitor extraction	19.3 ± 2.0	26.8 ± 2.4	22.5 ± 2.1	21.7 ± 2.0	5.2 ± 0.8	31.5 ± 3.2
Controller construction	17.7 ± 1.7	19.4 ± 2.1	$\mathbf{31.6 \pm 2.7}$	$\mathbf{24.8 \pm 2.2}$	$\mathbf{6.1 \pm 1.0}$	$\mathbf{35.9 \pm 3.7}$

Table 6: Outcomes of abstraction-changing macro-mutations. Proposal share is the fraction of generated candidates using each operator. Failure coverage is the percentage of admitted candidates that solve at least one previously uncovered diagnostic failure. Median survival is measured in generations. Abbreviations: Prop. = Proposal share; Adm. = Admission rate; Sp. = Species; Fail. Cov. = Failure coverage; Desc. Gain = Descendant success gain; Med. Surv. = Median survival.

of 14.9%. With $\delta_{\mathrm{sp}} = 0.40$ and $\nu_{\min} = 0.08$, these values are 20.9 and 5.4%, respectively, while the verified depth is 3.8. The configuration $\delta_{\mathrm{sp}} = 0.30$ and $\nu_{\min} = 0.04$ achieves an average success of 60.4, a species-survival rate of 68.7%, and a verified depth of 4.6.

C.3 Macro-Mutation Operator Analysis

Table 6 shows that procedural parameterization has an admission rate of 31.7% and produces new species in 12.9% of cases. Planner extraction, monitor extraction, and controller construction have admission rates of 22.6%, 26.8%, and 19.4%, respectively. Their failure-coverage rates are 17.9%, 21.7%, and 24.8%, and their median survival times are 29.7, 31.5, and 35.9 generations, respectively.

C.4 Performance by Failure Type

Table 7 shows that OESS achieves higher held-out success than the evaluated baselines across all failure categories. Relative to Text-Speciation, OESS increases held-out success by 6.8, 6.4, 7.8, 6.8, 5.6, and 7.2 percentage points for tool errors, partial completion, invalid states, delayed feedback, interface shifts, and dependency failures, respectively. Invalid-state recovery and delayed-feedback cases require persistent state and feedback-conditioned control beyond isolated local corrections.

C.5 Per-Seed Stability

Table 8 reports the results across five seeds. Average success ranges from 58.9 to 61.8, cross-task reuse ranges from 41.0% to 46.8%, and composition gain ranges from 7.8 to 10.7 percentage points. The invalid-novelty rate remains below 10% for every seed.

C.6 Archive Capacity and Search Efficiency

Table 9 reports the results for archive capacities ranging from 32 to 256. Increasing the capacity from 32 to 128 changes average success from 56.9 to 60.4, the number of verified species from 20.7 to 31.8, cross-task reuse from 35.2% to 44.1%, and normalized cost from 1.07 to 1.13. Increasing the capacity from 128 to 256 changes average success from 60.4 to 60.7, the number of verified

Method	Tool Err.	Partial Comp.	Invalid State	Delayed Fdbk.	Interface Shift	Dependency Fail.
Local-Mutation	45.2 ± 2.4	43.8 ± 2.5	38.6 ± 2.7	34.9 ± 2.8	41.3 ± 2.5	36.7 ± 2.6
Best-of-N	47.1 ± 2.3	45.9 ± 2.4	40.8 ± 2.6	36.5 ± 2.7	43.0 ± 2.4	38.9 ± 2.5
Random-Macro	52.4 ± 2.2	50.1 ± 2.3	45.7 ± 2.5	41.8 ± 2.6	46.7 ± 2.4	44.3 ± 2.5
Novelty-Only	54.1 ± 2.5	51.7 ± 2.6	48.8 ± 2.7	44.6 ± 2.9	48.5 ± 2.6	46.1 ± 2.7
Text-Speciation	56.0 ± 2.3	54.2 ± 2.4	50.5 ± 2.5	46.9 ± 2.7	51.8 ± 2.5	48.7 ± 2.6
OESS	$\mathbf{62.8 \pm 2.0}$	$\mathbf{60.6 \pm 2.1}$	$\mathbf{58.3 \pm 2.2}$	$\mathbf{53.7 \pm 2.5}$	$\mathbf{57.4 \pm 2.3}$	$\mathbf{55.9 \pm 2.4}$

Table 7: Held-out success rates by recoverable failure type. Each cell reports the mean and standard deviation across five seeds. Abbreviations: Err. = Error; Comp. = Completion; Fdbk. = Feedback; Fail. = Failure.

Metric	Seed 1	Seed 2	Seed 3	Seed 4	Seed 5	Mean $\pm$ Std.
Average success	59.1	61.8	60.6	58.9	61.6	60.4 ± 1.5
Verified species	29.0	34.0	32.0	30.0	34.0	31.8 ± 2.2
Species survival (%)	65.2	71.8	69.4	64.9	72.2	68.7 ± 3.1
Cross-task reuse (%)	41.0	46.8	44.2	42.1	46.4	44.1 ± 2.5
Verified depth	4.0	5.0	5.0	4.0	5.0	4.6 ± 0.5
Composition gain	7.8	10.4	9.5	8.1	10.7	9.3 ± 1.3
Invalid novelty (%)	6.2	8.7	7.4	7.1	9.6	7.8 ± 1.4

Table 8: Per-seed results for OESS. Success, survival, reuse, composition gain, and invalid novelty are percentages or percentage-point changes.

species from 31.8 to 34.6, cross-task reuse from 44.1% to 44.8%, the invalid-novelty rate from 7.8% to 9.1%, and normalized cost from 1.13 to 1.21.

D Prompt Templates

D.1 Overview

This appendix specifies the prompt templates for constructing, mutating, executing, and evaluating agent skills. All methods share the same executor prompt, tool interface, decoding configuration, and sandbox unless specified otherwise. Methods differ only in the candidate-generation prompt, mutation-operator selection, archive context, and admission rule. The templates contain no task-specific demonstrations.

D.2 Notation

We use the following placeholders:

- {`TASK`}: the natural-language task instruction.
- {`INITIAL_STATE`}: the initial environment state and observable context.
- {`TOOL_SPECIFICATIONS`}: the available tools, their typed arguments, return values, and usage constraints.
- {`SUBSKILL_LIBRARY`}: accessible subskills and their declared input–output signatures.
- {`PARENT_SKILL`}: the complete executable specification of the selected parent skill.
- {`PARENT_SIGNATURE`}: the declared typed input–output signature of the parent.
- {`PARENT_DESCRIPTOR`}: the structured descriptor $\Phi(s)$ of the parent.
- {`EXECUTION_TRACES`}: observed execution traces for the parent or candidate.
- {`VERIFIER_FEEDBACK`}: task-verifier outcomes and associated diagnostic feedback.
- {`FAILURE_CASES`}: diagnostic tasks or traces not solved by the current archive.
- {`ARCHIVE_SUMMARY`}: species representatives, structured descriptors, and verified failure coverage in the current archive.

Archive Capacity	Avg. Success	Verified Species	Reuse (%)	Invalid Novelty (%)	Normalized Cost
32	56.9 ± 1.8	20.7 ± 1.9	35.2 ± 2.7	6.9 ± 1.3	1.07
64	59.1 ± 1.6	27.4 ± 2.1	40.8 ± 2.6	7.3 ± 1.4	1.10
128	$\mathbf{60.4 \pm 1.5}$	$\mathbf{31.8 \pm 2.2}$	$\mathbf{44.1 \pm 2.5}$	7.8 ± 1.4	1.13
256	60.7 ± 1.6	34.6 ± 2.5	44.8 ± 2.7	9.1 ± 1.6	1.21

Table 9: Sensitivity to archive capacity. Cost is normalized by Local-Mutation under the matched 100-generation budget.

- {MUTATION_OPERATOR}: one of parameterize, encapsulate, plan_decompose, extract_monitor, or build_controller.
- {CANDIDATE_SKILL}: the complete specification of a generated offspring.
- {EXECUTION_BUDGET}: the maximum number of environment actions or tool invocations allowed for an episode.
- {CANDIDATE_A} and {CANDIDATE_B}: two skills compared during descriptor validation or redundancy analysis.

A skill specification contains an implementation, an execution policy, and a declared typed input–output signature. Its temporal abstraction level belongs to $\mathcal{H} = \{\text{PRIMITIVE}, \text{MACRO}, \text{MONITOR}, \text{CONTROLLER}, \text{META}\}$.

D.3 Main Inference Prompt

Purpose. Execute a selected skill on a task while satisfying its declared interface, tool constraints, and execution budget.

Inputs. The task instruction, initial state, selected skill, tool specifications, accessible subskills, and execution budget.

Expected Output. A sequence of valid tool or subskill calls followed by a terminal structured record. The terminal record reports the final status and output without modifying the skill during evaluation.

Prompt.

```
You are an agent executing a reusable skill in an external environment.

Task:
{TASK}

Initial state:
{INITIAL_STATE}

Selected skill:
{CANDIDATE_SKILL}

Available tools:
{TOOL_SPECIFICATIONS}

Accessible subskills:
{SUBSKILL_LIBRARY}

Execution budget:
{EXECUTION_BUDGET}

Follow the selected skill's declared input-output signature and execution
policy. Use only the listed tools and subskills. Base each action on the
```

```
current observation and any verifier or monitor feedback available at that
step. Satisfy all preconditions, argument types, and termination conditions.

If the skill defines a planner, use it to select or decompose goals before
execution. If it defines a monitor or controller, update its state after each
observation and apply its retry, recovery, and stopping rules. Report success
only when the skill's completion conditions are satisfied.

During execution, emit only valid tool or subskill calls. At termination,
return exactly:

{
  "status": "success" | "failure",
  "output": <task-dependent output or null>,
  "termination_reason": "<brief factual reason>",
  "invoked_subskills": ["<subskill name>", "..."],
  "tool_call_count": <non-negative integer>
}
```

D.4 Baseline Prompts

D.4.1 Local Mutation

Purpose. Generate a candidate by locally revising one incumbent while preserving its temporal interface, callable subskill hierarchy, planner, monitor, and stateful controller structure.

Inputs. The parent skill, its signature, execution traces, verifier feedback, tool specifications, and accessible subskills.

Expected Output. One executable candidate with the same input–output signature and abstraction level as the parent.

Prompt.

```
Revise an executable agent skill using local edits only.

Parent skill:
{PARENT_SKILL}

Declared signature:
{PARENT_SIGNATURE}

Execution traces:
{EXECUTION_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Available tools:
{TOOL_SPECIFICATIONS}

Accessible subskills:
{SUBSKILL_LIBRARY}

Make the smallest changes required to address the observed failures. Revise
existing instructions, conditions, argument construction, error handling, or
termination checks as needed.

Preserve the existing callable abstractions, planner, verifier, monitor,
persistent control state, subskill hierarchy, and multi-step macro
interface. Preserve the parent's declared signature and temporal abstraction
```

```
level. Return a complete replacement.

Return exactly:

{
  "name": "<skill name>",
  "abstraction_level": "<unchanged parent level>",
  "input_schema": <JSON object>,
  "output_schema": <JSON object>,
  "dependencies": ["<tool or subskill>", "..."],
  "execution_policy": "<complete policy>",
  "implementation": "<complete executable implementation>",
  "termination_conditions": ["<condition>", "..."],
  "failure_handling": [
    {
      "condition": "<failure condition>",
      "response": "<local repair>"
    }
  ],
  "change_summary": ["<local change>", "..."]
}
```

D.4.2 BEST-OF-N

Purpose. Generate independent local revisions from the same incumbent. The executor evaluates all candidates and retains the highest-scoring candidate.

Inputs. The local-mutation inputs and a candidate index.

Expected Output. One independently generated local candidate per invocation.

Prompt.

```
Generate candidate {CANDIDATE_INDEX} of {NUM_CANDIDATES} as an independent
local revision of the incumbent.

Parent skill:
{PARENT_SKILL}

Declared signature:
{PARENT_SIGNATURE}

Execution traces:
{EXECUTION_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Available tools:
{TOOL_SPECIFICATIONS}

Accessible subskills:
{SUBSKILL_LIBRARY}

Produce a complete executable replacement that preserves the parent's
signature and abstraction level. Address the observed failures through local
changes to existing instructions, conditions, arguments, repairs, or
termination checks. Preserve the existing macro, planner, monitor,
controller, callable subskill hierarchy, and persistent state. Generate the
candidate independently of all other candidates.
```

```
Return exactly the skill JSON schema specified in
Appendix~\ref{app:prompt:output}.
```

D.4.3 RANDOM MACRO-MUTATION

Purpose. Apply one abstraction-changing macro-mutation sampled uniformly from the operator set used by OESS, without species-aware selection or species preservation.

Inputs. The uniformly sampled operator, parent skill, traces, verifier feedback, tools, and accessible subskills.

Expected Output. One executable candidate satisfying the sampled macro-mutation.

Prompt.

```
Apply the following macro-mutation to the parent skill:

Mutation operator:
{MUTATION_OPERATOR}

Parent skill:
{PARENT_SKILL}

Declared signature:
{PARENT_SIGNATURE}

Execution traces:
{EXECUTION_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Available tools:
{TOOL_SPECIFICATIONS}

Accessible subskills:
{SUBSKILL_LIBRARY}

Implement the sampled mutation as a complete executable skill. Preserve the
declared interface unless the mutation introduces a new typed interface that
can be checked automatically. Apply only the specified mutation operator.
Ignore species membership and archive novelty during generation.

Return exactly the skill JSON schema specified in
Appendix~\ref{app:prompt:output}.
```

D.4.4 NOVELTY-ONLY SEARCH

Purpose. Generate behaviorally distinct candidates without conditioning admission on newly verified failure coverage.

Inputs. The parent skill, archive behavior summary, execution traces, verifier feedback, tools, and accessible subskills.

Expected Output. One executable candidate that differs behaviorally from archived candidates.

Prompt.

```
Generate an executable variation of the parent skill with a behaviorally
distinct solution strategy.

Parent skill:
{PARENT_SKILL}

Archived behavioral summaries:
{ARCHIVE_SUMMARY}

Execution traces:
{EXECUTION_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Available tools:
{TOOL_SPECIFICATIONS}

Accessible subskills:
{SUBSKILL_LIBRARY}

Differentiate the candidate through its action organization, control flow,
tool or subskill usage, recovery behavior, or termination behavior.
Superficial rewriting is insufficient. Newly covered failure cases need not
be established during generation.

Apply one available local or macro-level transformation. Return a complete
executable skill using the standard skill JSON schema.
```

D.4.5 TEXT SPECIATION

Purpose. Generate candidates with the OESS macro-mutation templates when species assignment uses embedding similarity between skill documents in place of structured distance.

Inputs. The selected parent and mutation context used by the corresponding OESS macro-mutation.

Expected Output. One executable candidate in the standard skill schema.

Prompt.

```
Use the macro-mutation template associated with {MUTATION_OPERATOR}.
Generate the candidate from the supplied parent, traces, verifier feedback,
tools, and subskills. Exclude structured species descriptors and
structured-distance information from the prompt.

Return exactly the standard skill JSON object.
```

D.5 FEW-SHOT AND DEMONSTRATION PROMPTS

We use no few-shot demonstrations. Candidate generation, mutation, execution, and descriptor extraction operate zero-shot from the skill specification, execution traces, verifier feedback, tool documentation, and archive state.

D.6 Reasoning and Rationale Prompts

D.6.1 Structured Skill Descriptor Extraction

Purpose. Map an executable skill and its verified traces to the structured descriptor $\Phi(s) = (\phi_{\text{io}}, \phi_{\text{temp}}, \phi_{\text{tool}}, \phi_{\text{repair}}, \phi_{\text{beh}})$ used for speciation.

Inputs. A skill specification, tool and subskill schemas, verified execution traces, and verifier feedback.

Expected Output. A structured JSON descriptor containing input–output semantics, temporal abstraction, tool dependencies, failure-repair behavior, and behavioral signatures.

Prompt.

```
Extract a structured behavioral descriptor from the executable skill and its
observed executions.

Skill:
{CANDIDATE_SKILL}

Tool specifications:
{TOOL_SPECIFICATIONS}

Accessible subskills:
{SUBSKILL_LIBRARY}

Verified execution traces:
{EXECUTION_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Use only information supported by the skill or traces. Exclude hidden or
unsupported capabilities. Assign exactly one temporal abstraction level:

- Primitive: a short local action rule or tool-call sequence.
- Macro: a reusable parameterized multi-step procedure.
- Monitor: a component that inspects execution and triggers repair.
- Controller: a state-dependent policy supporting branching, retry,
  recovery, or stopping.
- Meta: a skill that constructs, selects, composes, or revises other skills.

Represent tool and subskill dependencies as a typed directed graph. Record
failure-repair behavior when a failure condition and its response are
explicit or observed. Summarize behavior using action sequences, state
transitions, termination decisions, and verified success patterns.

Return exactly:

{
  "io_semantics": {
    "inputs": [
      {
        "name": "<name>",
        "type": "<type>",
        "semantics": "<meaning>"
      }
    ],
    "outputs": [
      {
        "name": "<name>",
        "type": "<type>",
```

```
        "semantics": "<meaning>"
      }
    ],
    "preconditions": ["<condition>", "..."],
    "postconditions": ["<condition>", "..."]
  },
  "temporal_abstraction": {
    "level": "Primitive" | "Macro" | "Monitor" | "Controller" | "Meta",
    "control_horizon": "single-step" | "short" | "multi-step" | "episodic",
    "termination_basis": ["<basis>", "..."]
  },
  "tool_dependency_graph": {
    "nodes": [
      {
        "id": "<identifier>",
        "kind": "tool" | "subskill" | "internal_component",
        "type": "<declared type>"
      }
    ],
    "edges": [
      {
        "source": "<identifier>",
        "target": "<identifier>",
        "relation": "calls" | "passes_data" | "checks" | "controls"
      }
    ]
  },
  "failure_repair_profile": [
    {
      "failure_condition": "<observed or declared condition>",
      "detection": "<detection mechanism>",
      "repair": "<repair action>",
      "verified": true | false
    }
  ],
  "behavioral_signature": {
    "action_patterns": ["<pattern>", "..."],
    "state_transitions": ["<transition>", "..."],
    "branching_conditions": ["<condition>", "..."],
    "termination_decisions": ["<decision>", "..."],
    "verified_success_patterns": ["<pattern>", "..."]
  }
}
```

D.6.2 MACRO-MUTATION SELECTION

Purpose. Select an applicable abstraction-changing mutation from the parent descriptor, observed traces, failures, and accessible subskills.

Inputs. The parent descriptor, execution traces, verifier feedback, failure cases, and subskill library.

Expected Output. One mutation operator and a concise, evidence-grounded target specification.

Prompt.

```
Select one abstraction-changing mutation for the parent skill.

Parent descriptor:
{PARENT_DESCRIPTOR}

Execution traces:
```

```
{EXECUTION_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Unsolved diagnostic failures:
{FAILURE_CASES}

Accessible subskills:
{SUBSKILL_LIBRARY}

Choose exactly one operator:

- parameterize: convert a recurring primitive sequence into a reusable,
  parameterized routine;
- encapsulate: extract a coordinated dependency subgraph behind a callable
  typed interface;
- plan_decompose: separate goal decomposition and subskill selection from
  execution;
- extract_monitor: extract repeated checks, diagnosis, and repair into a
  monitor;
- build_controller: convert a linear workflow into state-dependent control
  with branching, retry, recovery, and stopping.

Select an operator whose prerequisite pattern is supported by the skill or
traces. Identify the recurring sequence, dependency subgraph, planning
decision, repair pattern, or control deficiency targeted by the mutation.

Return exactly:

{
  "operator": "parameterize" | "encapsulate" | "plan_decompose" |
              "extract_monitor" | "build_controller",
  "target": "<specific component or behavior>",
  "evidence": ["<trace- or skill-supported observation>", "..."],
  "required_interface": {
    "inputs": <JSON object>,
    "outputs": <JSON object>
  }
}
```

D.7 Data Generation and Pseudo-Labeling Prompts

Task success does not use LLM-generated pseudo-labels. Executable task verifiers determine success labels and newly covered failures. The following templates generate candidate skill artifacts.

D.7.1 Procedural Abstraction Mutation

Purpose. Convert recurring primitive action sequences into a parameterized macro-skill.

Inputs. The parent skill, repeated execution patterns, verifier feedback, tool definitions, and accessible subskills.

Expected Output. A complete executable skill containing a reusable parameterized routine and a checkable interface.

Prompt.

```
Transform the parent skill by applying procedural abstraction.

Parent skill:
```

```
{PARENT_SKILL}

Declared signature:
{PARENT_SIGNATURE}

Execution traces:
{EXECUTION_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Available tools:
{TOOL_SPECIFICATIONS}

Accessible subskills:
{SUBSKILL_LIBRARY}

Identify a recurring multi-step primitive sequence supported by the traces.
Extract it into a named, parameterized routine with explicit typed inputs,
typed outputs, preconditions, postconditions, and termination conditions.
Replace duplicated occurrences with calls to the routine.

Preserve observable behavior outside the targeted sequence. Add no unrelated
capabilities. Preserve the parent's external signature unless the mutation
requires a new external interface that can be checked automatically.

Set "mutation_operator" to "parameterize" and return the complete skill using
the standard JSON schema.
```

D.7.2 SUBSKILL ENCAPSULATION MUTATION

Purpose. Encapsulate a coordinated dependency subgraph behind a reusable callable interface.

Inputs. The parent skill, dependency information, traces, verifier feedback, tools, and subskills.

Expected Output. A complete skill with an explicit encapsulated subskill and updated caller logic.

Prompt.

```
Transform the parent skill by encapsulating a coordinated dependency
subgraph.

Parent skill:
{PARENT_SKILL}

Parent descriptor:
{PARENT_DESCRIPTOR}

Execution traces:
{EXECUTION_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Available tools:
{TOOL_SPECIFICATIONS}

Accessible subskills:
{SUBSKILL_LIBRARY}

Identify a connected group of tool or subskill invocations that jointly
```

```
implements a reusable function. Encapsulate this group behind one callable,
typed interface. Specify its inputs, outputs, internal dependencies,
preconditions, completion conditions, and propagated failure states. Update
the parent to call the encapsulated component and remove duplicated internal
steps.

Preserve data flow and externally visible behavior. Encapsulate only
operations that jointly implement the identified function.

Set "mutation_operator" to "encapsulate" and return the complete skill using
the standard JSON schema.
```

D.7.3 PLANNER EXTRACTION MUTATION

Purpose. Separate goal decomposition and subskill selection from task execution.

Inputs. The parent skill, task traces, verifier feedback, tools, and candidate subskills.

Expected Output. A complete skill with distinct planner and executor components.

Prompt.

```
Transform the parent skill by extracting a planner.

Parent skill:
{PARENT_SKILL}

Declared signature:
{PARENT_SIGNATURE}

Execution traces:
{EXECUTION_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Available tools:
{TOOL_SPECIFICATIONS}

Accessible subskills:
{SUBSKILL_LIBRARY}

Separate goal decomposition and subskill selection from execution. The
planner must map the task state and declared interfaces to an ordered or
conditional plan. Each plan step must name a tool or subskill, provide typed
arguments, define expected outputs, and specify a completion condition. The
executor must carry out the plan and return observations or failures to the
planner when replanning is permitted.

Use an accessible tool or subskill for each corresponding environment
operation. Preserve the parent skill's external signature and checkable
termination behavior.

Set "mutation_operator" to "plan_decompose" and return the complete skill
using the standard JSON schema, including explicit "planner" and "executor"
fields.
```

D.7.4 MONITOR EXTRACTION MUTATION

Purpose. Extract repeated execution checks, diagnosis, and repair into a reusable monitor.

Inputs. The parent skill, failed and successful traces, verifier feedback, tools, and subskills.

Expected Output. A complete skill with a monitor that emits structured execution decisions.

Prompt.

```
Transform the parent skill by extracting a failure monitor.

Parent skill:
{PARENT_SKILL}

Execution traces:
{EXECUTION_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Available tools:
{TOOL_SPECIFICATIONS}

Accessible subskills:
{SUBSKILL_LIBRARY}

Identify repeated checks that distinguish normal progress, recoverable
failure, unrecoverable failure, and successful completion. Extract these
checks and their associated repairs into a monitor invoked at the relevant
execution points.

The monitor must consume the current observation, recent action history, and
available verifier feedback. It must return exactly one decision:
"continue", "retry", "repair", "stop_success", or "stop_failure". For "retry"
and "repair", it must identify the affected step and provide a bounded
response. It may return "stop_success" only after receiving an explicit
completion signal.

Set "mutation_operator" to "extract_monitor" and return the complete skill
using the standard JSON schema, including:

"monitor": {
  "inputs": <JSON object>,
  "decision_schema": {
    "decision": "continue" | "retry" | "repair" |
                "stop_success" | "stop_failure",
    "target": "<step identifier or null>",
    "response": "<repair specification or null>"
  },
  "checks": ["<check>", "..."]
}
```

D.7.5 Stateful Controller Mutation

Purpose. Convert a linear workflow into a state-dependent controller supporting branching, retry, recovery, and stopping.

Inputs. The parent skill, traces, verifier or monitor feedback, tool definitions, and accessible subskills.

Expected Output. A complete skill containing a controller $\kappa = (\mathcal{Z}, z_0, \Gamma, \Omega)$ with explicit states, transitions, and action rules.

Prompt.

```
Transform the parent skill from a linear workflow into a state-dependent
controller.

Parent skill:
{PARENT_SKILL}

Execution traces:
{EXECUTION_TRACES}

Verifier or monitor feedback:
{VERIFIER_FEEDBACK}

Available tools:
{TOOL_SPECIFICATIONS}

Accessible subskills:
{SUBSKILL_LIBRARY}

Define a finite set of control states Z, an initial state z0, a state-update
function Gamma, and an action function Omega. The update must depend only on
the current state, current observation, and available verifier or monitor
feedback. The action function must depend only on the current state and
observation.

Introduce branches supported by distinct observed conditions. Specify
bounded retry behavior, recovery transitions, successful stopping, and
failed stopping. Every reachable nonterminal state must define at least one
action and transition. All loops must have finite bounds.

Set "mutation_operator" to "build_controller" and include:

"controller": {
  "states": ["<state>", "..."],
  "initial_state": "<state>",
  "terminal_states": {
    "success": ["<state>", "..."],
    "failure": ["<state>", "..."]
  },
  "transitions": [
    {
      "state": "<current state>",
      "condition": "<observation or feedback condition>",
      "next_state": "<next state>"
    }
  ],
  "actions": [
    {
      "state": "<state>",
      "condition": "<observation condition>",
      "action": "<tool or subskill invocation>"
    }
  ],
  "retry_limits": [
    {
      "state": "<state>",
      "maximum_retries": <non-negative integer>,
      "on_exhaustion": "<next state>"
    }
  ]
}

Return the complete skill using the standard JSON schema.
```

D.8 Evaluation Prompts

D.8.1 Executability and Interface Validation

Purpose. Validate that a generated candidate is syntactically complete, callable through the declared interface, and restricted to available dependencies before environment evaluation.

Inputs. The candidate skill, tool specifications, and accessible subskill library.

Expected Output. A machine-readable validity decision and a list of concrete violations.

Prompt.

```
Validate the candidate skill without repairing it.

Candidate skill:
{CANDIDATE_SKILL}

Available tools:
{TOOL_SPECIFICATIONS}

Accessible subskills:
{SUBSKILL_LIBRARY}

Check all of the following:

1. The skill declares typed inputs and outputs.
2. The implementation is complete and syntactically executable.
3. Every referenced tool and subskill is available.
4. Every invocation satisfies the declared argument schema.
5. All required outputs are defined on every terminal path.
6. Termination conditions are explicit.
7. Retry and recovery loops are bounded.
8. Any newly introduced interface is internally consistent and checkable.
9. The declared abstraction level is consistent with the implementation.

Return exactly:

{
  "executable": true | false,
  "interface_valid": true | false,
  "dependency_valid": true | false,
  "control_flow_valid": true | false,
  "violations": [
    {
      "category": "syntax" | "interface" | "dependency" |
                  "control_flow" | "termination" | "abstraction_level",
      "location": "<component>",
      "description": "<specific violation>"
    }
  ]
}
```

D.8.2 Behavioral Reproduction Check

Purpose. Determine whether a candidate reproduces its claimed newly covered behavior across independently instantiated diagnostic tasks. Executable verifier outputs determine the final success decision.

Inputs. The candidate, claimed failure coverage, execution traces from independent task instances, and verifier outputs.

Expected Output. A structured reproduction record grounded in verifier outcomes.

Prompt.

```
Summarize the reproducibility of the candidate's claimed behavioral novelty.

Candidate skill:
{CANDIDATE_SKILL}

Claimed newly covered failures:
{FAILURE_CASES}

Independent execution traces:
{EXECUTION_TRACES}

Executable verifier outputs:
{VERIFIER_FEEDBACK}

Use the verifier outcomes without modification or reinterpretation. For each
claimed failure pattern, report whether the candidate succeeds on every
required independent instantiation and whether all archived comparison
skills fail on the same instances. A claim is reproduced only when both
conditions hold.

Return exactly:

{
  "claims": [
    {
      "failure_id": "<identifier>",
      "candidate_outcomes": [0 | 1, "..."],
      "archive_max_outcomes": [0 | 1, "..."],
      "reproduced": true | false
    }
  ],
  "all_required_claims_reproduced": true | false
}
```

D.8.3 Structured Redundancy Analysis

Purpose. Identify behaviorally redundant candidates during species-level capacity control while preserving representative and lineage-critical skills.

Inputs. Two candidate skills, their structured descriptors, verified traces, and lineage metadata.

Expected Output. A structured comparison of behavioral equivalence and distinguishing evidence.

Prompt.

```
Compare two archived skills for behavioral redundancy.

Candidate A:
{CANDIDATE_A}

Candidate B:
{CANDIDATE_B}

Structured descriptors:
{CANDIDATE_DESCRIPTORS}
```

```
Verified execution traces:
{EXECUTION_TRACES}

Lineage metadata:
{LINEAGE_METADATA}

Compare typed input-output semantics, temporal abstraction, tool and subskill
dependencies, failure-repair behavior, action organization, state
transitions, and termination decisions. Establish redundancy from behavioral
evidence, excluding textual similarity alone. Preserve representative and
lineage-critical skills.

Return exactly:

{
  "behaviorally_redundant": true | false,
  "equivalent_components": [
    "io" | "temporal" | "tool" | "repair" | "behavior"
  ],
  "distinguishing_components": [
    "io" | "temporal" | "tool" | "repair" | "behavior"
  ],
  "evidence": ["<trace- or specification-grounded observation>", "..."],
  "lineage_critical": {
    "candidate_a": true | false,
    "candidate_b": true | false
  }
}
```

D.9 Ablation Prompt Variants

D.9.1 No Species Protection

Purpose. Generate candidates with the full OESS mutation templates while removing species-conditioned protection and local competition.

Inputs. The selected parent and standard mutation context.

Expected Output. A candidate in the standard skill schema.

Prompt.

```
Apply the selected OESS mutation template to the parent skill using the
provided traces, verifier feedback, tools, and accessible subskills.

Exclude species identity, species medoids, species age, species utility, and
species-specific archive context. The generated candidate competes directly
with all archived candidates.

Return exactly the standard skill JSON object.
```

D.9.2 No Macro-Mutation

Purpose. Restrict OESS candidate generation to local revisions that preserve the parent's abstraction structure.

Inputs. The parent skill, traces, verifier feedback, tools, and subskills.

Expected Output. A locally revised candidate with no new abstraction-changing component.

Prompt.

```
Revise the parent skill using local edits only.

Parent skill:
{PARENT_SKILL}

Execution traces:
{EXECUTION_TRACES}

Verifier feedback:
{VERIFIER_FEEDBACK}

Available tools:
{TOOL_SPECIFICATIONS}

Accessible subskills:
{SUBSKILL_LIBRARY}

Preserve the parent's input-output signature, temporal abstraction level,
dependency structure, and control organization. Preserve all existing
multi-step routines, dependency subgraphs, planners, monitors, and stateful
controllers. Revise only existing instructions, arguments, checks, repairs,
or stopping conditions.

Return exactly the standard skill JSON object.
```

D.9.3 TEXT-DISTANCE SPECIATION

Purpose. Construct a textual representation for species assignment when embedding similarity between skill documents replaces structured distance.

Inputs. The complete candidate skill document.

Expected Output. A canonical text rendering used as the embedding input.

Prompt.

```
Render the following skill as a canonical text document for embedding-based
species assignment:

{CANDIDATE_SKILL}

Include the skill name, declared inputs, declared outputs, dependencies,
execution policy, implementation, failure handling, and termination
conditions. Preserve the skill's substantive content. Exclude behavioral
interpretations, inferred capabilities, execution traces, verifier outcomes,
and structured descriptor labels.

Return plain text only.
```

D.9.4 NO BEHAVIORAL VERIFICATION

Purpose. Generate and describe candidate novelty without conditioning species formation on reproducible coverage of previously unsolved failures.

Inputs. The candidate, archive summary, and structured descriptor.

Expected Output. A structural novelty record that excludes verifier-based failure coverage.

Prompt.

```
Assess structural novelty relative to the current archive.

Candidate skill:
{CANDIDATE_SKILL}

Candidate descriptor:
{PARENT_DESCRIPTOR}

Archive summary:
{ARCHIVE_SUMMARY}

Compare the candidate with archived species representatives using
input-output semantics, temporal abstraction, tool dependencies,
failure-repair structure, and behavior description. Exclude executable
verifier outcomes, independently reproduced failures, and newly covered
failure sets.

Return exactly:

{
  "nearest_species": "<species identifier or null>",
  "structurally_distinct": true | false,
  "distinct_components": [
    "io" | "temporal" | "tool" | "repair" | "behavior"
  ],
  "evidence": ["<descriptor-grounded difference>", "..."]
}
```

D.9.5 No Lineage Retention

Purpose. Use the standard candidate-generation templates without exposing ancestry or preserving lineage-critical intermediates.

Inputs. The selected parent and current evaluation context.

Expected Output. A candidate in the standard skill schema with no lineage-dependent fields.

Prompt.

```
Generate a candidate using the selected mutation template.

Use only the selected parent, its execution traces, verifier feedback,
available tools, and accessible subskills. Exclude ancestor skills,
descendant histories, prior mutation paths, lineage-critical annotations,
replay information, parent identifiers, and lineage metadata.

Return exactly the standard skill JSON object.
```

D.10 Output Formatting

All candidate-generation prompts return one JSON object with the following schema. Fields that do not apply to a candidate are represented by `null` or an empty list.

```
{
  "name": "<unique skill name>",
  "description": "<concise functional description>",
```

```
  "mutation_operator": "local" | "parameterize" | "encapsulate" |
                       "plan_decompose" | "extract_monitor" |
                       "build_controller",
  "abstraction_level": "Primitive" | "Macro" | "Monitor" |
                       "Controller" | "Meta",
  "input_schema": {
    "<input name>": {
      "type": "<type>",
      "description": "<semantic description>",
      "required": true | false
    }
  },
  "output_schema": {
    "<output name>": {
      "type": "<type>",
      "description": "<semantic description>"
    }
  },
  "preconditions": ["<condition>", "..."],
  "postconditions": ["<condition>", "..."],
  "dependencies": [
    {
      "name": "<tool or subskill name>",
      "kind": "tool" | "subskill",
      "purpose": "<usage>"
    }
  ],
  "execution_policy": "<complete decision and execution policy>",
  "implementation": "<complete executable implementation>",
  "planner": <planner object or null>,
  "monitor": <monitor object or null>,
  "controller": <controller object or null>,
  "termination_conditions": {
    "success": ["<condition>", "..."],
    "failure": ["<condition>", "..."]
  },
  "failure_handling": [
    {
      "condition": "<failure condition>",
      "detection": "<detection rule>",
      "response": "<bounded repair or stopping action>"
    }
  ],
  "interface_changes": {
    "changed": true | false,
    "justification": "<reason or null>",
    "compatibility": "<compatibility rule or null>"
  },
  "change_summary": ["<substantive change>", "..."]
}
```

The response must contain valid JSON with no Markdown fences or surrounding text. Tool and subskill names must exactly match the supplied specifications. Every control-flow path must terminate or specify an explicit finite retry bound. Generator-produced rationales are restricted to the evidence, change-summary, and interface-justification fields; executable verification determines validity.

E Case Study

We qualitatively examine an archived lineage produced by a repository-maintenance task. The task required an agent to replace a deprecated configuration interface, update dependent call sites, and preserve the repository's existing behavior. The environment provided tools for file search, file editing, dependency inspection, test execution, and verifier feedback. The verifier returned only the failing

command and a short diagnostic message, without identifying the edit responsible for the failure. The repository also contained generated files, compatibility wrappers, and a test whose first execution could terminate before producing a complete report.

The initial skill was a linear workflow synthesized from successful primitive trajectories:

```
skill migrate_config_key(old_key, new_key):
    search repository for old_key
    replace every matching occurrence with new_key
    run the repository test command
    if the test command succeeds:
        return success
    return failure
```

Although the instruction-level workflow was plausible, its execution trace revealed three failure modes. First, unrestricted replacement modified a generated snapshot that had to remain unchanged. Second, the skill updated a compatibility wrapper before inspecting its callers, removing the fallback required by an older entry point. Third, every unsuccessful test execution caused immediate termination, including executions that produced incomplete reports without evidence of a semantic regression. Local mutations changed search terms, added textual cautions, or appended another test invocation while preserving the same behavioral sequence: search, edit, test, and stop. The skill therefore continued to conflate edit scope, dependency order, and failure recovery.

Procedural abstraction. A macro-mutation extracted the recurring inspect–edit–validate pattern into a parameterized routine. The descendant classified textual matches by artifact role and required an explicit validation command for each edited region.

```
routine patch_region(target, rewrite_rule, local_check):
    context = inspect(target)
    if context.is_generated or context.is_vendored:
        return SKIPPED_PROTECTED_ARTIFACT

    patch = apply_rewrite(context, rewrite_rule)
    result = run(local_check)

    if result.passed:
        return PATCH_ACCEPTED(target, patch)
    revert(patch)
    return PATCH_REJECTED(target, result.diagnostic)
```

This mutation introduced an intermediate representation beyond prompt compression. Each candidate edit produced a typed outcome, such as `PATCH_ACCEPTED`, `PATCH_REJECTED`, or `SKIPPED_PROTECTED_ARTIFACT`. These outcomes exposed partial progress to subsequent skills and prevented a failed local check from invalidating unrelated edits. The routine also handled a previously unresolved case in which the deprecated key appeared in both source code and generated fixtures: it changed the source occurrence and recorded the fixture as protected.

The macro-skill still depended on the order in which targets were supplied. Patching the compatibility wrapper before its downstream callers passed the local check, but the repository-level verifier subsequently reported an import failure. The failure arose from the coordination of individually valid patches, not from an incorrect rewrite within a target.

Subskill encapsulation and planning. A subsequent mutation encapsulated dependency discovery, patch application, and validation behind a new interface. An extracted planner constructed an explicit migration plan before modifying any files.

```
skill plan_migration(symbol):
    definitions = find_definitions(symbol)
```

```
    callers = trace_callers(definitions)
    protected = detect_generated_and_vendored_paths()
    checks = infer_checks_for(callers, definitions)

    return MigrationPlan(
        preserve = definitions.compatibility_boundaries,
        edit_order = callers.leaves_before_wrappers(),
        protected_paths = protected,
        validation_map = checks,
        completion_condition =
            "no editable caller uses the deprecated interface"
    )
```

For the examined repository, the planner emitted the following artifact:

```
MigrationPlan:
  preserve:
    - src/compat/config_adapter.py::read_legacy_key
  edit_order:
    - src/loaders/yaml_loader.py
    - src/commands/run_job.py
    - src/runtime/bootstrap.py
    - src/compat/config_adapter.py
  protected_paths:
    - tests/snapshots/config_schema.generated.json
    - vendor/legacy_plugin/
  validation_map:
    src/loaders/yaml_loader.py:
      - tests/loaders/test_yaml_loader.py
    src/commands/run_job.py:
      - tests/commands/test_run_job.py
    src/runtime/bootstrap.py:
      - tests/runtime/test_bootstrap.py
    src/compat/config_adapter.py:
      - tests/compat/test_config_adapter.py
  completion_condition:
    editable callers use the new key and the compatibility boundary
    still accepts legacy input
```

This plan illustrates the abstraction change preserved by speciation. The descendant introduced a reusable object representing edit order, protected artifacts, local validation obligations, and a semantic completion condition. Different executors could consume the plan because its output left the exact tool calls unspecified.

The planner also changed how the compatibility wrapper was handled. The linear ancestor treated the remaining legacy key as evidence of an incomplete migration. The planned skill distinguished an intentional compatibility boundary from an unmodified caller. It preserved the legacy read path while migrating internal consumers to the new interface, resolving a failure that global replacement could not address.

Monitor extraction. The next lineage transition followed an execution in which the repository test command ended with the diagnostic `worker exited before report finalization`. The existing executor treated every unsuccessful command as evidence of an invalid patch and reverted all accepted edits. A verifier-extraction mutation separated semantic failures from infrastructure and observation failures:

```
monitor classify_validation(result, expected_scope):
    if result.report_complete and result.assertions_failed:
        return SEMANTIC_FAILURE(
            failing_tests = result.failed_tests,
```

```
            implicated_scope = expected_scope
        )

    if result.contains("worker exited before report finalization"):
        return INCOMPLETE_OBSERVATION(
            recovery = "rerun affected checks without parallel workers"
        )

    if result.contains("unknown option"):
        return HARNESS_MISMATCH(
            recovery = "inspect available test-runner interface"
        )

    if result.passed and result.covered(expected_scope):
        return VERIFIED

    return UNRESOLVED(
        evidence = result.stdout_tail,
        recovery = "request narrower diagnostic execution"
    )
```

The monitor explicitly represented uncertainty. The prior skill made a binary decision from the process exit status. The extracted monitor determined whether the available report supported a semantic conclusion. For an incomplete run, it retained the accepted patches and requested a narrower, non-parallel validation command. For a harness mismatch, it revised the invocation without modifying repository files. Patch-level repair or reversion occurred only when a report contained an attributable assertion failure.

This monitor formed a distinct behavioral species because it changed the failure-repair profile and the tool-dependency graph. Validation output became input to a reusable diagnostic component that could reroute execution without directly editing source files. Superficially rephrased ancestors remained behaviorally equivalent to the original linear workflow and were therefore excluded as separate species.

Stateful controller. The final descendant converted the planned workflow into a feedback-driven controller. Its persistent state recorded the current plan, accepted patches, pending checks, observed failure class, and recovery history.

```
state MigrationState =
    DISCOVER
  | PLAN
  | PATCH(target)
  | LOCAL_VALIDATE(target)
  | REPAIR(target, diagnosis)
  | GLOBAL_VALIDATE
  | COMPLETE
  | SAFE_ABORT

controller migrate_deprecated_interface(request):
    state = DISCOVER
    plan = none
    accepted = []
    recovery_log = []

    while state not in {COMPLETE, SAFE_ABORT}:
        observation = observe_environment()

        match state:
            DISCOVER:
                inventory = inspect_repository(request.symbol)
                state = PLAN
```

```
PLAN:
    plan = plan_migration(request.symbol)
    if plan.has_ambiguous_definition:
        state = SAFE_ABORT
    else:
        state = PATCH(plan.next_target())

PATCH(target):
    outcome = patch_region(
        target,
        request.rewrite_rule,
        plan.validation_map[target]
    )
    if outcome is PATCH_ACCEPTED:
        accepted.append(outcome.patch)
        state = LOCAL_VALIDATE(target)
    elif outcome is SKIPPED_PROTECTED_ARTIFACT:
        plan.mark_protected(target)
        state = PATCH(plan.next_target())
    else:
        state = REPAIR(target, outcome.diagnostic)

LOCAL_VALIDATE(target):
    diagnosis = classify_validation(
        observation.validation_result,
        expected_scope = target
    )
    if diagnosis is VERIFIED:
        if plan.has_pending_target:
            state = PATCH(plan.next_target())
        else:
            state = GLOBAL_VALIDATE
    elif diagnosis is INCOMPLETE_OBSERVATION:
        recovery_log.append(diagnosis)
        rerun_narrow_check(target, diagnosis.recovery)
        state = LOCAL_VALIDATE(target)
    elif diagnosis is HARNESS_MISMATCH:
        recovery_log.append(diagnosis)
        adapt_test_invocation()
        state = LOCAL_VALIDATE(target)
    else:
        state = REPAIR(target, diagnosis)

REPAIR(target, diagnosis):
    repair = propose_scope_limited_repair(
        target, diagnosis, accepted
    )
    if repair.preserves_plan_invariants:
        apply(repair)
        state = LOCAL_VALIDATE(target)
    else:
        revert_changes_for(target)
        state = SAFE_ABORT

GLOBAL_VALIDATE:
    result = run_repository_verifier()
    diagnosis = classify_validation(
        result,
        expected_scope = plan.completion_condition
    )
    if diagnosis is VERIFIED:
        state = COMPLETE
    elif diagnosis is INCOMPLETE_OBSERVATION:
        recovery_log.append(diagnosis)
        run_serial_repository_verifier()
```

```
                    state = GLOBAL_VALIDATE
                elif diagnosis is SEMANTIC_FAILURE:
                    target = localize_failure(
                        diagnosis,
                        plan,
                        accepted
                    )
                    state = REPAIR(target, diagnosis)
                else:
                    state = SAFE_ABORT

    return MigrationReport(
        status = state,
        accepted_patches = accepted,
        preserved_boundaries = plan.preserve,
        skipped_artifacts = plan.protected_paths,
        recoveries = recovery_log
    )
```

A representative execution identified the compatibility adapter as an intentional boundary and the generated snapshot as a protected artifact. The controller then migrated leaf callers before modifying the runtime bootstrap. During local validation of the command module, the test runner produced an incomplete report. The monitor classified the event as INCOMPLETE_OBSERVATION, and the controller reran only the affected command tests without parallel workers. The resulting report exposed an assertion failure caused by a stale keyword argument. The repair state localized the issue to the command module, updated the call site, and repeated local validation while retaining the previously accepted loader changes.

During global validation, the harness rejected an inherited command-line option. The controller classified this event as HARNESS_MISMATCH, inspected the available test-runner interface, and issued an equivalent supported invocation. A tooling incompatibility therefore caused no repository-code changes. The final migration report separated changed source files, preserved compatibility behavior, skipped generated artifacts, and environment-level recoveries:

```
MigrationReport:
  status: COMPLETE

  accepted_patches:
    - src/loaders/yaml_loader.py
      action: read the replacement key internally
      evidence: loader-specific validation completed

    - src/commands/run_job.py
      action: replace the stale keyword argument at the caller
      evidence: command-specific validation completed after repair

    - src/runtime/bootstrap.py
      action: propagate the replacement configuration field
      evidence: runtime-specific validation completed

    - src/compat/config_adapter.py
      action: preserve legacy input and normalize it to the new field
      evidence: compatibility validation completed

  preserved_boundaries:
    - src/compat/config_adapter.py::read_legacy_key

  skipped_artifacts:
    - tests/snapshots/config_schema.generated.json
      reason: generated artifact
    - vendor/legacy_plugin/
      reason: external compatibility surface
```

```
recoveries:
  - event: incomplete validation report
    response: rerun affected checks without parallel workers
    repository edits caused by event: none

  - event: unsupported test-runner option
    response: inspect harness and use a supported invocation
    repository edits caused by event: none

completion_evidence:
  - editable callers no longer depend on the deprecated interface
  - legacy input remains accepted at the declared boundary
  - protected artifacts were not rewritten
  - local and repository-level checks produced complete reports
```

Qualitative comparison with alternative search behaviors. Local mutation produced increasingly elaborate variants of the original sequence, but these skills continued to treat the repository as a flat set of textual matches. They lacked an intermediate representation of edit order and could not preserve accepted partial progress across recovery attempts. Random macro-mutation occasionally produced wrappers around search and test commands, but these wrappers were often structurally decorative: they changed function boundaries without changing when the agent branched, reverted, or stopped. Novelty-oriented search preserved candidates with unusual tool-call orders, including repeated validation and broad repository rescans, but some candidates failed to reproduce the behavior that motivated their admission. Text-based grouping also separated skills with different documentation terminology even when their traces implemented the same search–replace–test workflow.

OESS preserved the examined lineage because each structural transition addressed a concrete unresolved behavior. Procedural abstraction introduced typed patch outcomes. Planning made dependency order and compatibility boundaries explicit. Monitor extraction separated insufficient evidence from semantic failure. Controller construction coordinated these components through persistent state and failure-conditioned transitions. The final behavior emerged through composition across protected intermediate species, beyond the changes produced by a single prompt rewrite.

This case also shows that implementation complexity alone does not identify abstraction discovery. The controller contains more structure than its ancestor, but its relevant novelty lies in the decisions enabled by that structure. It skips protected artifacts, preserves compatibility code, acquires additional evidence after incomplete reports, changes tool invocation after harness errors, and repairs semantic failures at the smallest implicated scope. These behaviors change the organization of execution and recovery, which is the form of functional abstraction that OESS is designed to discover and retain.