Evolving Programmatic Skill Networks

Haochen Shi[1,2] **Xingdi Yuan**[3*] **Bang Liu**[1,2,4*]

[1] DIRO & Institut Courtois, Université de Montréal [2] Mila – Québec AI Institute
[3] Microsoft Research [4] Canada CIFAR AI Chair

haochen.shi@umontreal.ca eric.yuan@microsoft.com bang.liu@umontreal.ca

Abstract

We study continual skill acquisition in open-ended embodied environments where an agent must construct, refine, and reuse an expanding library of executable skills. We introduce the Programmatic Skill Network (PSN), a framework in which skills are executable symbolic programs forming a compositional network that evolves through experience. PSN defines three core mechanisms instantiated via large language models: (1) REFLECT for structured fault localization over skill compositions, (2) progressive optimization with maturity-aware update gating that stabilizes reliable skills while maintaining plasticity for uncertain ones, and (3) canonical structural refactoring under rollback validation that maintains network compactness. We further show that PSN's learning dynamics exhibit structural parallels to neural network training. Experiments on MineDojo and Crafter demonstrate robust skill reuse, rapid adaptation, and strong generalization across open-ended task distributions.[1]

1 Introduction

Embodied agents operating in open-ended environments must continually acquire, refine, and reuse a growing repertoire of skills. Existing approaches (Wang et al., 2024a; Yao et al., 2023) suffer from two limitations: (1) skills are typically represented as flat libraries or static graphs lacking principled mechanisms for continual improvement, and (2) agents lack unified frameworks for assigning credit over hierarchical skill compositions, repairing symbolic programs, and reorganizing structure as new tasks arise.

We introduce the Programmatic Skill Network (PSN), a framework for continually evolving skill libraries. In a PSN, each skill is a symbolic program (e.g., in JavaScript for Minecraft, Python for Crafter) with explicit control flow, parameters, and preconditions that specify applicability and effects. Skills invoke each other through dependency links, forming a directed graph that grows and reorganizes as the agent learns. While recent work has explored programmatic skill representations for agents (Wang et al., 2024b; Stengel-Eskin et al., 2024; Wang et al., 2025d), PSN uniquely maintains an explicit computational graph of executable programs that supports trace-based credit assignment, maturity-aware stabilization, and principled structural refactoring.

The framework structures continual learning through three components (Figure 1): a *network-aware planner* that prioritizes skill reuse via backward-chaining, a *fault localization mechanism* (REFLECT) that assigns credit over skill compositions by analyzing execution traces, and a *refactor module* that reorganizes network structure. These components are instantiated using LLMs for program synthesis, but the continual learning behavior emerges from the architectural scaffolding rather than the LLM itself.

A key insight is that PSN's learning dynamics exhibit structural parallels to neural network training. Fault localization over skill compositions resembles backpropagation through computational graphs (Rumelhart et al., 1986); maturity-based update gating induces stability-

[*] Equal advising
[1] Code: github.com/evolving-skill-networks/psn; Webpage: evolving-skill-networks.github.io.

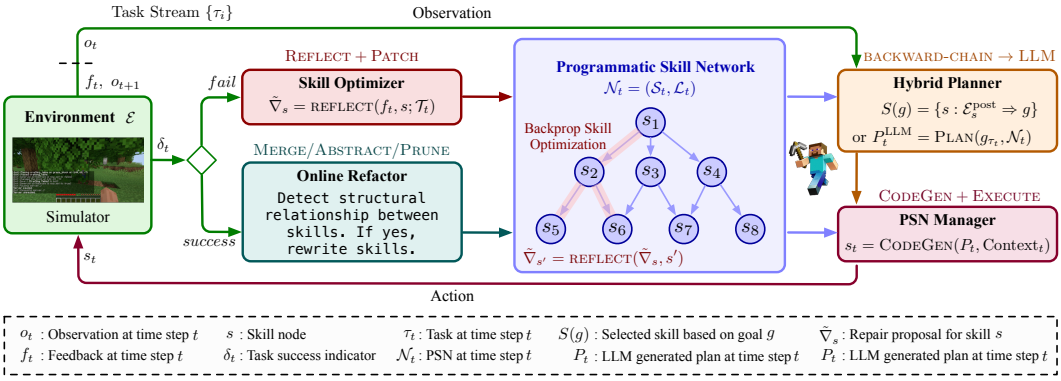


Figure 1: The Programmatic Skill Network (PSN) framework. The agent maintains a skill network $\mathcal{N}_t$ in which the *hybrid planner* selects or synthesizes skills and the *PSN manager* executes them. On failure the *skill optimizer* performs trace-based credit assignment; on success the *online refactor* restructures the network. This induces learning dynamics analogous to neural network training: fault localization as backpropagation, maturity gating as learning rate scheduling, refactoring as architecture search.

plasticity tradeoffs analogous to layer freezing and learning rate scheduling (Howard & Ruder, 2018; Yosinski et al., 2014; Rusu et al., 2016); and structural refactoring performs a form of symbolic neural architecture search (Zoph & Le, 2017; Han et al., 2016; Tan & Le, 2019).

The contributions of this work are threefold:

- **Programmatic Skill Networks.** A framework for continual skill learning in which skills are executable symbolic programs with explicit control flow, parameters, and pre/postconditions, linked by invocation into an inspectable computational graph that grows and reorganizes as the agent learns.
- **PSN learning mechanisms.** Three complementary mechanisms for continual skill improvement: (1) REFLECT for fault localization; (2) maturity-aware update gating that stabilizes reliable skills while keeping uncertain ones plastic; and (3) canonical structural refactoring with rollback validation that eliminates redundancy while preserving performance.
- **An optimization perspective.** We show that PSN's architectural design induces learning dynamics with structural parallels to neural network training, suggesting general principles for continual learning across representational paradigms.

2 Method

Problem setup. We consider an embodied agent acting in a partially observable Markov decision process (POMDP) (Kaelbling et al., 1998). The agent receives a stream of open-ended tasks $T = \{\tau_1, \tau_2, \ldots\}$, each specified in natural language and associated with a goal predicate $g_\tau : \mathcal{S} \to \{0, 1\}$ over the state space $\mathcal{S}$. Tasks arrive sequentially and vary in difficulty, horizon length, and compositional structure, so the agent must continually acquire, refine, and reorganize reusable skills to solve future tasks from past experience.

We present an online framework for continually constructing, optimizing, and refactoring a Programmatic Skill Network, evolving through a recurrent loop that couples symbolic planning, execution, failure-driven repair, and success-driven structural refactoring. We first define the core objects and operators, then the planning and learning mechanisms.

2.1 Programmatic Skill Networks (PSN)

A skill $s = (\mathcal{C}_s, \mathcal{P}_s, \mathcal{E}_s, \text{CHILDREN}(s))$ is a symbolic program where $\mathcal{C}_s$ denotes control flow, $\mathcal{P}_s$ parameters, $\mathcal{E}_s = (\mathcal{E}_s^{\text{pre}}, \mathcal{E}_s^{\text{post}})$ preconditions/postconditions, and $\text{CHILDREN}(s)$ invoked subskills. This precondition-effect structure is analogous to programmatic laws in symbolic world modeling (Khan et al., 2026). The agent maintains a directed network $\mathcal{N}_t = (\mathcal{S}_t, \mathcal{L}_t)$ where nodes $\mathcal{S}_t$ are skills and edges $\mathcal{L}_t$ represent invocations.

Executing skill s yields (f_s, δ_s) where $\delta_s \in \{0, 1\}$ indicates success and f_s aggregates feedback from the environment, and the system records a finite invocation trace $\mathcal{T}$. Given f_s, REFLECT computes a repair proposal $\tilde{\nabla}_s$ and propagates responsibility to invoked subskills as $\tilde{\nabla}_{s'} = \text{REFLECT}(\tilde{\nabla}_s, s')$ for $s' \in \text{CHILDREN}(s)$, yielding finite credit assignment over executed subgraphs (Section 2.4).

Each skill maintains a scalar value $V(s) = \hat{p}_s - u_s$, where $\hat{p}_s$ is its Laplace-smoothed success rate and u_s an uncertainty term that decreases with more executions. This value summarizes long-term reliability and plays a dual role: guiding skill selection during planning and modulating update frequency during optimization.

Beyond behavioral repair, the PSN evolves through structure-level rewrites such as merging redundant skills, abstracting shared routines, pruning irrelevant branches, and rewiring invocation links. These operations are treated as discrete architecture updates and are validated through rollback-based safety checks (Section 2.5).

LLM implementation. Operators such as REFLECT are instantiated via prompted LLMs: the framework defines the information flow *structure* (what information is available, output formats, update timing) while LLMs provide the generative capacity to synthesize, diagnose, and repair programs within it. Critically, the learning dynamics we observe (Section 3) emerge from PSN's *architectural choices* (the compositional network structure, trace-based credit assignment, maturity-gated updates, and canonical refactor operations) rather than from the internal mechanisms of the LLM, so the framework can be instantiated with different code generation backends while preserving its continual learning properties.

2.2 Network-aware hybrid planner

The planner prioritizes reuse of the existing PSN via symbolic backward-chaining before invoking LLM-based forward planning. Each skill s is treated as an operator with preconditions $\mathcal{E}_s^{\text{pre}}$ and postconditions $\mathcal{E}_s^{\text{post}}$. Starting from the goal predicate, the planner selects skills whose postconditions satisfy the current subgoal, $S(g) = \{s : \mathcal{E}_s^{\text{post}} \Rightarrow g\}$, and recursively expands unmet preconditions. When multiple skills satisfy a subgoal, ties are broken by $V(s)$, favoring skills with higher empirical reliability, and selection uses Boltzmann exploration (Sutton & Barto, 1998) over $V(s)$ to balance exploitation of reliable skills against exploration of uncertain ones. If no skill can reduce a subgoal, the planner invokes an LLM-based forward planner $P_t^{\text{LLM}} = \text{PLAN}(g_{\tau_t}, \mathcal{N}_t)$. Successful plans are distilled into new skills via the execution pipeline described next.

2.3 Execution and trace construction

Given a plan $P_t = [s_1, \ldots, s_k]$, the PSN manager synthesizes a candidate skill $s_t = \text{CODEGEN}(P_t, \text{Context}_t)$, where Context_t includes the task description, current network $\mathcal{N}_t$, and execution history. The synthesized skill defines control flow $\mathcal{C}_{s_t}$, parameters $\mathcal{P}_{s_t}$, and pre/postconditions $\mathcal{E}_{s_t}$, and is inserted into the PSN with invocation links to its children. Executing s_t produces a skill execution trace, $\text{EXECUTE}(s_t) \rightarrow (f_t, \delta_t, \mathcal{T}_t)$, where $\delta_t \in \{0, 1\}$ indicates task success, f_t aggregates environment feedback and critic signals, and the trace $\mathcal{T}_t$ records each invoked skill as a tuple $\langle s, \sigma^{\text{pre}}, \sigma^{\text{post}}, \text{status} \rangle$ with symbolic state snapshots σ. The trace serves as supervision for both optimization and refactoring. Preconditions and postconditions are incrementally calibrated from observed success/fail states and empirical transitions.

2.4 Skill optimization via trace-based credit assignment

When execution fails (i.e., $\delta_t = 0$), the skill optimizer performs localized behavioral repair via structured fault localization. Unlike approaches that discover world dynamics in natural language (Sun et al., 2024) or learn function libraries offline (Stengel-Eskin et al., 2024), PSN performs online, trace-based credit assignment over executable skill compositions. Given feedback f_t and trace $\mathcal{T}_t$, the REFLECT operator computes a repair proposal for each executed skill: $\tilde{\nabla}_s = \text{REFLECT}(f_t, s; \mathcal{T}_t)$, identifying faulty control flow, violated preconditions, mis-

aligned parameters, or incorrect subskill effects. Concretely, PSN separates *credit assignment* from *code modification* through a two-phase process: failure signals are first propagated *top-down* along the executed skill invocation trace to decompose responsibility across composite skills and their subskills (symbolic differentiation), after which localized symbolic edits are applied *bottom-up* to individual skills in a dependency-respecting order (gradient application). Proposals propagate in reverse execution order along the invocation trace; skills not in $\mathcal{T}_t$ receive no updates. Each affected skill is updated via $s \leftarrow \textsc{Patch}(s, \tilde{\nabla}_s)$. The complete two-phase procedure is given in Appendix B.

To stabilize learning, updates are constrained by a rolling buffer of the 5 most recent repair proposals, preventing contradictory edits, and update frequency is further modulated by skill maturity:

$$P(\text{update } s) = (1 - \epsilon) \cdot \sigma(\gamma(0.6 - V(s))) + \epsilon, \tag{1}$$

where σ is the sigmoid, $\gamma = 5.0$ controls threshold sharpness, and $\epsilon = 0.1$ ensures a minimum update probability. The constant 0.6 is a soft maturity pivot rather than a bound on $V(s)$: it marks the inflection point at which a skill is reliable enough to gradually reduce its update frequency, while still permitting occasional repairs under compositional failures. Mature skills ($V(s) \approx 1$) stabilize with low update probability, while immature ones remain plastic.

2.5 Online structural refactoring

The online skill refactor controls structural growth via semantics-preserving architecture-level rewrites that increase skill reuse and maintain network compactness. While code refactoring has been used to discover generalizable abstractions offline (Stengel-Eskin et al., 2024), PSN refactors online. Where the skill optimizer repairs individual skill programs, refactor operates at the network level, targeting the redundancy and missed abstractions that emerge over continual learning.

Canonical refactor cases. We restrict refactor to five structural relationships: (i) *Parametric coverage*: one skill is a strict specialization of another admitting parameterized generalization. (ii) *Behavioral coverage*: a composite skill reimplements existing functionality. (iii) *Sibling specializations*: multiple skills suggest a missing abstraction. (iv) *Common subskill extraction*: multiple skills share identical sub-operations. (v) *Duplication*: two skills are functionally equivalent. Each admits a fixed rewrite rule; visual illustrations are provided in Appendix C.

Candidate discovery and rewrites. Given a successfully executed skill s_t, refactor operates on a restricted candidate set: parents and children of s_t, plus top-5 semantically related skills by embedding similarity. For each detected relationship, deterministic rewrites are applied (wrapper conversion, call substitution, abstract skill synthesis, shared subskill extraction, or canonical merging). Refactor does not introduce new behavioral logic, it only reorganizes existing programs and invocation links.

Safety via rollback validation. All refactor proposals are tentative. Given a refactored candidate network $\mathcal{N}'_t$, the system evaluates short-horizon performance on a sliding window of 3 recent tasks involving affected skills. If the task success rate drops by more than 20%, the refactor is reverted using logged inverse operations. Empirical rollback rates and refactoring case distributions are reported in Appendix L and Appendix M.

3 An optimization perspective on PSN

Having presented PSN's concrete mechanisms (Section 2), we can observe that the system's learning dynamics exhibit structural parallels to neural network training. While other neuro-symbolic systems embed symbolic rules inside differentiable models (d'Avila Garcez et al., 2019; Manhaeve et al., 2018) or use gradient-free skill-based routing (Chen et al., 2026), PSN embeds learning dynamics inside symbolic programs. This interpretive lens clarifies how PSN's architectural choices collectively induce coherent continual learning behavior, independent of the LLM backend.

Method	Wooden Tool	Stone Tool	Iron Tool	Diamond Tool
ReAct	N/A (0/3)	N/A (0/3)	N/A (0/3)	N/A (0/3)
Reflexion	N/A (0/3)	N/A (0/3)	N/A (0/3)	N/A (0/3)
AutoGPT	92 ± 72 (3/3)	94 ± 72 (3/3)	135 ± 103 (3/3)	N/A (0/3)
Voyager	6 ± 2 (3/3)	11 ± 2 (3/3)	21 ± 7 (3/3)	102 (1/3)
Voyager*	6 ± 3 (6/6)	13 ± 4 (6/6)	34 ± 10 (6/6)	99 ± 36 (2/6)
ADAM*	21 ± 4 (3/3)	36 ± 5 (3/3)	52 ± 15 (3/3)	74 ± 23 (3/3)
PSN w/o Optimizer	**5 ± 2 (3/3)**	12 ± 2 (3/3)	25 ± 4 (3/3)	N/A (0/3)
PSN (Ours)	**5 ± 2 (6/6)**	**10 ± 3 (6/6)**	**22 ± 5 (6/6)**	**35 ± 16 (6/6)**
Voyager†	9 ± 4 (6/6)	24 ± 23 (6/6)	60 ± 41 (4/6)	94 (1/6)
ADAM†	43 ± 8 (2/3)	65 (1/3)	N/A (0/3)	N/A (0/3)
PSN† (Ours)	**7 ± 3 (6/6)**	**18 ± 7 (6/6)**	**38 ± 9 (6/6)**	**49 ± 18 (6/6)**

Table 1: Tech tree mastery on Minecraft: mean/std iterations an agent uses to unlock an item, with the number of successful runs in parentheses. PSN and our Voyager reproductions (* and †) are averaged over six runs, the remaining baselines over three. For example, PSN unlocks the diamond tool in all six runs using 35 iterations on average, while Voyager (Wang et al., 2024a) succeeds in one of three runs using 102. * uses the open-sourced code with `GPT-5-mini` (same as ours); † uses `Qwen3-Coder-Next` (3B active / 80B total open-weight MoE). N/A: item never unlocked across all runs.

Implicit structure-behavior trade-off. Let $\mathcal{N} = (\mathcal{S}, \mathcal{L})$ denote the current PSN. The system's behavior can be viewed as implicitly optimizing a composite objective:

$$J(\mathcal{N}) = \mathcal{R}_{\text{task}} + \mathcal{R}_{\text{reliab}} + \mathcal{R}_{\text{struct}} + \mathcal{R}_{\text{cons}}, \tag{2}$$

balancing **task** success, skill **reliab**ility, **struct**ural compactness, and semantic **cons**istency. While never explicitly optimized, each PSN module performs localized improvements to different components of $J(\mathcal{N})$.

Operator-objective correspondence. REFLECT acts as *symbolic differentiation*: when a task fails, it identifies which control-flow branches, preconditions, parameters, and subskill compositions contributed to the error, producing structured repair proposals that reduce $\mathcal{R}_{\text{task}}$ and $\mathcal{R}_{\text{cons}}$. Like backpropagation, credit is assigned only along the executed path, which avoids the noise of updating uninvolved skills, mirroring how gradients flow only through activated paths in neural nets. A chain-rule view of this credit assignment makes a falsifiable prediction: a noisier repair signal must be propagated through more compositional layers to localize the same fault, so a weaker LLM should require deeper REFLECT chains. We observe exactly this (5.0 vs 2.7 skills per multi-skill repair; Section 4.6). Credit assignment patterns, error mode distributions, and attribution coherence are analyzed quantitatively in Appendix K. Maturity-aware gating functions as *adaptive learning rates*: mature skills with high $V(s)$ receive infrequent updates (analogous to freezing converged layers), while immature skills remain plastic, reducing $\mathcal{R}_{\text{reliab}}$ by preventing catastrophic forgetting. Refactor performs *symbolic neural architecture search*: merging redundant skills, extracting reusable abstractions, and pruning unnecessary branches to reduce $\mathcal{R}_{\text{struct}}$. Rollback-based validation functions as a symbolic trust region. The depth of this search is model-dependent: it converges to a hub-and-spoke graph with reuse ratio $\mathcal{R}_{\text{struct}} \approx 0.4$ for the stronger model but a flatter 0.15 ± 0.07 for the weaker one (Section 4.6), the structural counterpart of the deeper repair chains above.

Multi-scale learning dynamics. PSN learning unfolds across three coupled timescales: (1) *Fast*: fault localization performs frequent behavioral repair at every execution. (2) *Intermediate*: maturity-based stabilization progressively freezes reliable skills over 10–50 executions. (3) *Slow*: structural refactor reorganizes stabilized behaviors every 5–10 successful executions. This yields a coherent dynamic: optimize behavior locally and rapidly, stabilize reliable skills over time, and restructure only after behaviors have converged.

Scope of the analogy. The neural network analogy is **partial**. PSN operates over discrete symbolic programs rather than continuous parameters, produces structured edit proposals rather than numeric derivatives, and relies on binary success/failure signals rather than differentiable losses. Nevertheless, it reveals stability-plasticity tradeoffs, compositional

credit assignment, and architecture search as general principles when learning structured representations, suggesting that insights from neural network optimization may inform symbolic learning systems and vice versa. Appendix P decomposes $J(\mathcal{N})$ into its four components across training.

4 Experiments and analysis

We evaluate PSN on two complementary embodied benchmarks: **MineDojo** (Fan et al., 2022), which supports long-horizon open-ended Minecraft tasks with rich action spaces and diverse goal specifications, and **Crafter** (Hafner, 2022), a lightweight survival environment whose structured technology progression stresses continual learning and compositional reuse. Across both, we measure (i) end-task performance, (ii) continual learning dynamics (learning/forgetting), (iii) compositional generalization, and (iv) network structural properties (growth, reuse, redundancy) induced by refactor and maturity-aware optimization.

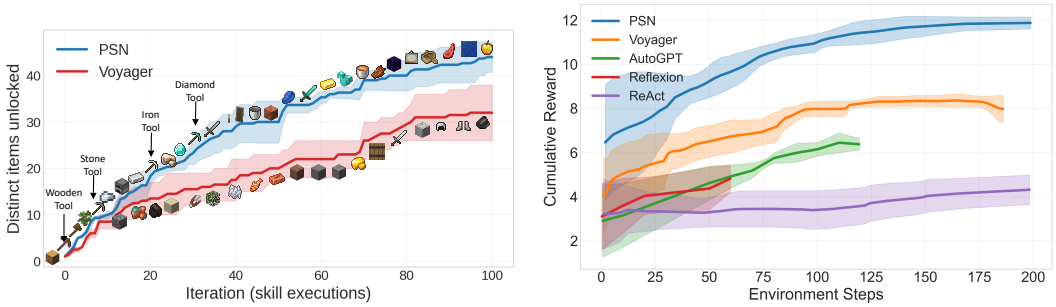
4.1 Experimental setup

We leverage OpenAI's `gpt-5-mini-2025-08-07` for all operators across both environments. To assess whether PSN's learning dynamics are robust across model families, we additionally run all Minecraft experiments with `Qwen3-Coder-Next` (Cao et al., 2026), an open-weight model with 3B active parameters (80B total), representing a fundamentally different model family from GPT-5-mini and orders of magnitude smaller in active model size. The Minecraft simulator is built on top of MineDojo and leverages Mineflayer JavaScript APIs for motor controls (PrismarineJS). For the Crafter environment, we implemented a Mineflayer-like Python API system for the control of the Crafter bot. PSN operators (e.g., CODEGEN and REFLECT) are instantiated by prompted LLMs (see Appendix E for example prompts). The continual learning dynamics we measure arise from PSN's architectural scaffolding—trace-based credit assignment, maturity-aware update gating, and canonical refactor operations—rather than from prompt-level tricks; a cross-LLM ablation of the inline knowledge scaffolding (Appendix F) verifies this attribution directly at the prompt level.

We compare PSN against representative LLM-agent baselines and ablations. **ReAct** (Yao et al., 2023) interleaves reasoning and action without persistent structured skills; **Reflexion** (Shinn et al., 2023) self-reflects over failures but maintains no compositional skill network; **AutoGPT** (Significant Gravitas, 2023) decomposes tasks into multi-step plans and autonomously executes generated code, but treats plans and code fragments as ephemeral artifacts rather than persistent, reusable skills. **Voyager** (Wang et al., 2024a) maintains a flat skill library retrieved by similarity, without trace-based symbolic credit assignment or canonical structural refactor. **ADAM** (Yu & Lu, 2025) discovers causal precondition-effect mappings for a fixed set of 47 hand-authored JavaScript skills via LLM-guided inference and intervention-based verification (do-calculus), then uses the learned causal graph to guide planning; unlike PSN it never generates or optimizes skill code. We run its original code with the same LLMs as PSN for controlled comparison. **ODYSSEY** (Liu et al., 2025) pre-builds 183 hand-authored JavaScript skills and reduces the LLM's role to retrieval-based selection rather than code generation (Appendix J). Other Minecraft LLM agents (e.g., DEPS (Wang et al., 2023), JARVIS-1 (Wang et al., 2025c), Optimus-1 (Li et al., 2024), RL-GPT (Liu et al., 2024)) operate in incompatible paradigms that preclude controlled comparison (Appendix O).

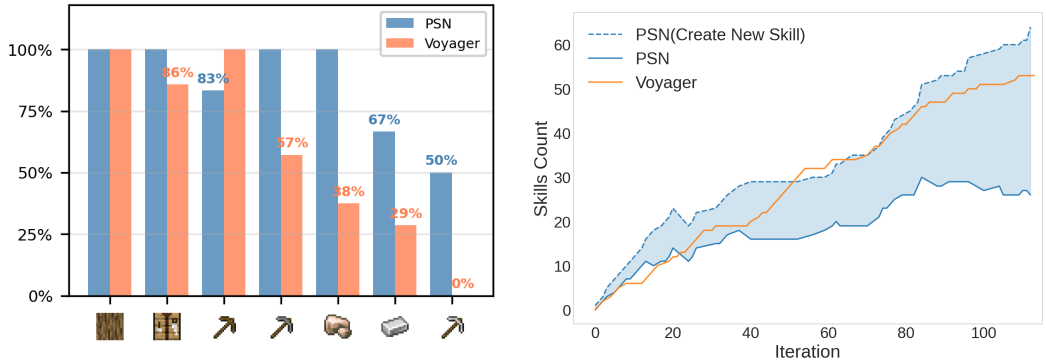
4.2 Main results

Minecraft tech tree mastery. Figure 2a and Table 1 compare agents in terms of technology tree progression, measured by the number of iterations. Progressing along the tech tree requires solving increasingly long-horizon and compositional tasks, where later-stage tools depend on reliable execution and reuse of earlier skills. ReAct and Reflexion fail to unlock any tool-level milestone; AutoGPT completes early-stage objectives but struggles beyond iron-level tools with high variance; Voyager progresses consistently through iron tools but slows sharply at the diamond stage. PSN instead keeps unlocking higher-tier items with fewer attempts and lower variance, indicating that persistent programmatic skills,



(a) Minecraft tech tree. (b) Crafter cumulative reward.

Figure 2: Main results on both benchmarks. **(a)** Tech tree mastery on Minecraft. **(b)** Cumulative reward on Crafter; shorter curves indicate earlier **agent death** due to Crafter's survival mechanics (hostile mobs, hunger, hazards).



(a) Skill Retention Rate. (b) Skill library growth.

Figure 3: Continual learning on Minecraft. **(a)** PSN preserves previously mastered skills, while Voyager exhibits severe catastrophic forgetting. **(b)** Skill library growth: PSN (Create New Skills) synthesizes a new skill for every task, whereas PSN reuses and optimizes existing skills and stays compact.

trace-based credit assignment, and structural refactoring enable sustained long-horizon competence. Beyond the milestones in Table 1, PSN is the only method that obtains obsidian at all, succeeding in 1 of 6 runs after 77 iterations, while every baseline, ablation and Qwen3-Coder-Next configuration fails in all runs; the result is too sparse to tabulate but marks the ceiling of the approach. PSN reaches it by executing a multi-step procedure (bucket crafting, water-lava interaction, diamond-pickaxe mining) encapsulated as a single composed skill that extensively reuses previously learned subskills, compressing long-horizon behavior into a reusable programmatic abstraction. A detailed comparison positioning PSN, ADAM, and ODYSSEY along an autonomy–engineering spectrum is provided in Appendix J. Crucially, these results are consistent across model families: under Qwen3-Coder-Next (3B active parameters), PSN reaches diamond tool in all 6 runs while Voyager† succeeds in only 1/6 (Table 1), confirming that the gains arise from PSN's architectural scaffolding rather than the specific LLM backend. By contrast, ADAM†'s causal inference degrades sharply under Qwen3, failing beyond stone tools, as its memoryless causal discovery mechanism is sensitive to LLM reasoning quality. We return to the prompt-level evidence for this in §4.5.

Crafter. Figure 2b reports cumulative episode reward on Crafter, reflecting the agent's ability to survive, gather resources, and make progress under dense feedback. Unlike Minecraft benchmarks emphasizing sparse milestone completion, Crafter requires sustained stability where early mistakes compound. Voyager achieves stabler returns than planning-only baselines but remains limited by its flat skill library, whereas PSN maintains steadily increasing reward throughout training, showing broader generalization to dense continual learning.

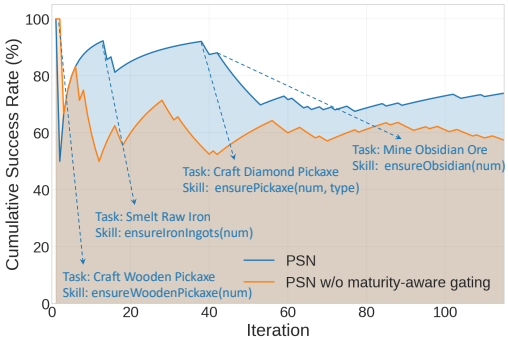


Figure 4: Cumulative task success rate of PSN w/ and w/o maturity-aware update gating.

4.3 Generalization

Continual learning over task streams (temporal generalization). Beyond the acquisition efficiency visible in Figure 2a, we evaluate PSN's ability to acquire increasingly complex skills from a sequential task stream while avoiding catastrophic forgetting. Tasks are presented in a fixed curriculum following the technology tree (Appendix I), each trained until its success rate exceeds a predefined threshold (marked as mastered) or a maximum number of attempts is reached. To measure forgetting, we introduce the *Skill Retention Rate (SRR)*: once a task is mastered it is re-evaluated after every subsequent task is mastered, and SRR is the cumulative success rate across those re-evaluations. As shown in Figure 3a, PSN consistently preserves earlier skills as training progresses, whereas Voyager exhibits severe backward interference, with retention rapidly degrading as new skills are learned. Structured credit assignment and maturity-aware stabilization are thus critical for continual skill acquisition.

Compositional generalization via network-aware skill reuse. We hypothesize that PSN solves unseen compositional tasks by reusing and recombining existing skills rather than synthesizing new ones. To test this, we introduce a controlled baseline, PSN (Create New Skills), which bypasses backward chaining and always synthesizes a new skill per task. In Figure 3b both variants grow similarly early on as foundational skills are acquired, but the gap widens over time: PSN's repertoire plateaus while PSN (Create New Skills) keeps accumulating skills. PSN thus increasingly grounds new tasks in its existing network via backward chaining, achieving compositional generalization through reuse rather than proliferation; its repertoire even shrinks in later iterations, as refactoring merges redundant helper functions.

4.4 Ablation study

End-to-end optimizer. We ablate the symbolic optimizer to disentangle the effect of optimization from that of skill representation. As shown in Table 1, PSN without the optimizer matches Voyager on early- and mid-stage tools (wooden, stone, iron) but fails to reliably progress to the diamond tool, mirroring Voyager's degradation under increasing task depth, whereas the full PSN consistently unlocks higher-tier items with substantially fewer iterations. The optimizer is thus not required to make skills functional, but is critical for repairing brittle behaviors and scaling stably to long-horizon, deeply compositional tasks.

Maturity-aware update gating gradually stabilizes learned skills. Figure 4 compares cumulative success rates for PSN with and without maturity-aware update gating. Without stabilization, converged skills are repeatedly modified by downstream failures, producing oscillatory behavior; gating instead damps updates to reliable skills while leaving immature ones plastic, yielding higher cumulative success and stabler learning dynamics.

Refactor regulates the network growth. Figure 3b shows how the size of the skill library evolves as learning progresses. Without structural refactoring, Voyager's library grows

rapidly, accumulating redundant or overly specialized skills, which increases planning complexity and degrades efficiency. PSN instead maintains a substantially more compact network by identifying canonical redundancy patterns and applying semantics-preserving rewrites, reducing the effective growth rate even as task complexity increases.

Offline refactor vs. online refactor. To test whether structural compression alone is sufficient, we apply an *offline refactor* to Voyager's learned skill library using a strong LLM (Claude Opus 4.5), which turned its 58 skills into 7 generic skills, 20 lightweight wrappers, and 38 unchanged skills (65 total), denoted Voyager-R. This markedly reduces redundant code blocks but not the behavioral brittleness: on a fixed sequence of compositional tasks (Appendix I) Voyager-R reaches a success rate of 0.6875, against 0.8462 for PSN with online refactoring. Refactoring is thus most effective when performed *online* and tightly coupled with execution feedback, rather than applied once to a static library.

4.5 REFLECT under LLM noise: a three-layer view

REFLECT is instantiated by a prompted LLM, so its repair signal is noisy by construction. The question is not how to make it noise-free, which is impossible, but how the surrounding architecture keeps learning stable despite it. We decompose REFLECT noise into three layers and locate the source of tolerance in each (Figure 5 in Appendix A).

Knowledge layer (missing/incorrect facts). The model misremembers a domain fact needed for diagnosis (e.g. Qwen3 asserting "raw_iron does not exist in vanilla Minecraft"). This is the only layer whose mitigation is model-specific: on synthetic tier-mismatch diagnoses, one injected `tool_tier_rules` corrigendum contributes +33pp on Qwen3 but only +2pp on GPT-5-mini (Appendix F). Such noise therefore all but vanishes on a stronger model; the corrigendum is a pretraining patch, not an architectural mechanism.

Reasoning layer (mis-attributed causes). With the right facts available, the model can still build a plausible but wrong causal story (e.g. blaming lava for an obsidian non-drop that is actually a tool-tier mismatch). No per-call fix exists; tolerance is architectural and operates at the loop level. Because every diagnosis is applied and re-executed ($s_{i+1} = s_i + \delta s_i$), a wrong fix fails its effect-check and is re-diagnosed against fresh evidence, turning single-shot diagnostic noise into a convergent process: first-attempt task success of 67% rises to 100% within three attempts (Table 9).

Interface layer (fabricated/misused APIs). The model emits code that calls non-existent methods or misuses real ones. PSN gates this before execution with a static reference checker (Babel parse and symbol resolution) plus explicit API-contract documentation, so fabricated calls are rejected rather than cascading into the trace; on a production-prompt replay (n=20) this lowers the interface-layer bug rate from 65% to 35%, an effect that holds across LLMs.

Two of the three defenses, the env-validated loop and the static interface gate, are architectural and hold across LLMs; only the knowledge corrigendum is model-specific.

4.6 Cross-model robustness and compositional reuse

PSN reaches the diamond tool on both a strong closed model (GPT-5-mini) and a weaker open mixture-of-experts model (Qwen3-Coder-Next, 3B active / 80B total), in 35 ± 16 and 49 ± 18 iterations over six runs each, with no architectural changes (Table 1). Three independent measurements all shift with model strength, yet the same architecture absorbs the difference.

Three signals scale with the model. (i) REFLECT *depth*: the weaker model needs deeper credit assignment, propagating failure feedback through 5.0 skills per multi-skill repair on average, versus 2.7 for the stronger model (Figure 6a in Appendix A). (ii) *Knowledge dependence*: the `tool_tier_rules` corrigendum contributes +33pp to the weaker model's diagnostic correctness but only +2pp to the stronger one's (§4.5). (iii) *Compositional reuse*: the reuse ratio $\mathcal{R}_{\text{struct}}$, the fraction of skills with fan-in > 1, asymptotes near 0.4 for the stronger model but only 0.15 ± 0.07 for the weaker one (Figure 6b). A weaker model thus emits a noisier repair signal, leans harder on explicit knowledge, and grows a flatter, less-reused

network. None of this breaks learning: both models complete the tech tree under the same loop, gating, and refactor operators. The weaker model simply works harder, diagnosing deeper and reusing less, while the surrounding training stack converts its noisier signal into the same outcome. Reliability comes from the architecture rather than from the model being strong.

Reuse topology. Reuse is also visible in the shape of the learned graph: PSN develops a hub-and-spoke structure in which a few utility skills (e.g. `setupCraftingTable`, `ensureResource`) are invoked by many parents, whereas a flat retrieval library such as Voyager's keeps nearly every skill at fan-in 0 by construction (Figure 7 in Appendix A).

5 Related work

Skill learning and hierarchical RL. Hierarchical RL studies temporal abstraction via options (Sutton et al., 1999; Barto & Mahadevan, 2003; Bacon et al., 2017; Eysenbach et al., 2019) and modular routing (Andreas et al., 2016; Xu et al., 2018; Zhang et al., 2018; Shazeer et al., 2017; Riquelme et al., 2021), and LLM-guided approaches segment trajectories into reusable skills via variational inference (Fu et al., 2024). Unlike these, PSN represents skills as executable programs with explicit control flow and pre/postconditions.

LLM-based agents and program synthesis. LLM agents maintain code memories or skill repositories (Yao et al., 2023; Schick et al., 2023; Ahn et al., 2022; Wang et al., 2024a; Prabhu et al., 2025). CodeAct (Wang et al., 2024b) uses executable code as a unified action space; Re-GAL (Stengel-Eskin et al., 2024) learns function libraries via refactoring; MINDcraft (White et al., 2025) studies multi-agent task solving; ASI (Wang et al., 2025d) induces programmatic skills on-the-fly for web agents; AgentCoder (Huang et al., 2023) uses multi-agent code generation; DiVE (Sun et al., 2024) builds natural language knowledge libraries; Wang et al. (2025a) show refactoring facilitates coding agents. Self-improving agents learn via RL-based skill accumulation (Wang et al., 2025b), reasoning memory (Ouyang et al., 2026), or skill disclosure (Anthropic, 2025). ADAM (Yu & Lu, 2025) learns causal input-output mappings for 47 predefined skills but does not generate or optimize code; ODYSSEY (Liu et al., 2025) relies on 183 hand-authored skills with LLM-based retrieval. Differently, PSN autonomously generates, composes and optimizes skill code within a compositional network, without hand-authored task-specific skills.

Neuro-symbolic learning and architecture optimization. Neuro-symbolic systems integrate symbolic structures with differentiable computation (d'Avila Garcez et al., 2019; Baydin et al., 2018; Badreddine et al., 2022; Manhaeve et al., 2018). OneLife (Khan et al., 2026) models dynamics via programmatic laws with precondition-effect structures analogous to PSN's skill representation; Skill-MoE (Chen et al., 2026) routes through skill-based experts; EFA (Khan et al., 2025) infers executable abstractions for math. Neural architecture search prunes and restructures networks (Zoph & Le, 2017; Han et al., 2016; Tan & Le, 2019), with learning rate scheduling enabling stability-plasticity tradeoffs (Howard & Ruder, 2018; Yosinski et al., 2014; Rusu et al., 2016). PSN draws on both traditions: it embeds learning dynamics inside symbolic programs rather than differentiable models, while performing architecture-search-like refactoring under rollback validation.

6 Conclusion

We introduced PSN, a framework for continual skill acquisition in which executable symbolic programs form a compositional network that evolves through experience. Its three mechanisms (trace-based credit assignment, maturity-aware update gating, and canonical structural refactoring) induce learning dynamics with structural parallels to neural network training. Experiments on Minecraft and Crafter demonstrated faster skill acquisition, reduced forgetting, and superior compositional generalization, suggesting that principles from neural network optimization can inform the design of symbolic learning systems.

Acknowledgments

This work is supported by the Canada CIFAR AI Chair Program and the Canada NSERC Discovery Grant (RGPIN-2021-03115). We also acknowledge the funding support from the Amazon Research Award (ARA) program.

Ethics Statement

This work studies autonomous skill acquisition in simulated game environments (Minecraft and Crafter). The agents operate in sandboxed simulators with no real-world deployment, posing no direct safety risks. Our framework uses prompted LLMs as code generation backends; we do not fine-tune models or train on private data. All experiments use publicly available game environments and APIs. We acknowledge that autonomous code generation systems could, in principle, produce unintended behaviors; PSN mitigates this through rollback validation and execution sandboxing.

Reproducibility Statement

To ensure reproducibility, we provide: (1) complete algorithmic specifications of all PSN operators in the main text and appendix, including the two-phase optimization algorithm (Appendix B); (2) example prompt templates for all LLM-instantiated operators (Appendix E); (3) exact model identifiers (`gpt-5-mini-2025-08-07` and `Qwen3-Coder-Next` with 3B active / 80B total parameters); (4) all hyperparameters (γ=5.0, ϵ=0.1, maturity pivot 0.6, rollback threshold 20%, momentum window 5); (5) task sequences used for evaluation (Appendix I); and (6) detailed code diffs for representative optimization cases (Appendix H). All experiments are conducted on Minecraft 1.19.4 via MineDojo/Mineflayer and Crafter.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as I can, not as I say: Grounding language in robotic affordances. In *Conference on Robot Learning (CoRL)*, 2022.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 39–48, 2016.

Anthropic. Equipping agents for the real world with agent skills, October 2025. URL https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills. Anthropic Engineering Blog.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2017.

Samy Badreddine, Artur d'Avila Garcez, Luciano Serafini, and Michael Spranger. Logic tensor networks. *Artificial Intelligence*, 303:103649, 2022. ISSN 0004-3702. doi: 10.1016/j.artint.2021.103649. URL https://www.sciencedirect.com/science/article/pii/S0004370221002009.

Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 2003.

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018. URL http://jmlr.org/papers/v18/17-468.html.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, et al. Qwen3-Coder-Next technical report. *arXiv preprint arXiv:2603.00729*, 2026.

Justin Chih-Yao Chen, Sukwon Yun, Elias Stengel-Eskin, Tianlong Chen, and Mohit Bansal. Skill-based mixture-of-experts: Adaptive routing for heterogeneous reasoning via inferred skills. In *Forty-third International Conference on Machine Learning*, 2026.

Artur d'Avila Garcez, Marco Gori, Luis C. Lamb, Luciano Serafini, Michael Spranger, and Son N. Tran. Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning. *arXiv preprint arXiv:1905.06088*, 2019. URL https://arxiv.org/abs/1905.06088.

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 (ICLR)*, 2019.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. MineDojo: Building open-ended embodied agents with internet-scale knowledge. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2022. Outstanding Paper Award.

Haotian Fu, Pratyusha Sharma, Elias Stengel-Eskin, George Konidaris, Nicolas Le Roux, Marc-Alexandre Côté, and Xingdi Yuan. Language-guided skill learning with temporal variational inference. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 14135–14156. PMLR, July 2024. URL https://proceedings.mlr.press/v235/fu24e.html. ICML 2024.

Danijar Hafner. Benchmarking the spectrum of agent capabilities. In *International Conference on Learning Representations (ICLR)*, 2022.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *International Conference on Learning Representations (ICLR)*, 2016.

Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2018.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998.

Zaid Khan, Elias Stengel-Eskin, Archiki Prasad, Jaemin Cho, and Mohit Bansal. Executable functional abstractions: Inferring generative programs for advanced math problems. *arXiv preprint arXiv:2504.09763*, 2025.

Zaid Khan, Archiki Prasad, Elias Stengel-Eskin, Jaemin Cho, and Mohit Bansal. One life to learn: Inferring symbolic world models for stochastic environments from unguided exploration. In *International Conference on Learning Representations*, pp. 99794–99836, 2026.

Zaijing Li, Yuquan Xie, Rui Shao, Gongwei Chen, Dongmei Jiang, and Liqiang Nie. Optimus-1: Hybrid multimodal memory empowered agents excel in long-horizon tasks. *Advances in neural information processing systems*, 37:49881–49913, 2024.

Shaoteng Liu, Haoqi Yuan, Minda Hu, Yanwei Li, Yukang Chen, Shu Liu, Zongqing Lu, and Jiaya Jia. RL-GPT: Integrating reinforcement learning and code-as-policy. *Advances in Neural Information Processing Systems*, 37:28430–28459, 2024.

Shunyu Liu, Yaoru Li, Kongcheng Zhang, Zhenyu Cui, Wenkai Fang, Yuxuan Zheng, Tongya Zheng, and Mingli Song. ODYSSEY: Empowering Minecraft agents with open-world skills. In James Kwok (ed.), *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pp. 187–195. International Joint Conferences on Artificial Intelligence Organization, 8 2025. doi: 10.24963/ijcai.2025/22. URL https://doi.org/10.24963/ijcai.2025/22. Main Track.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. DeepProbLog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems*, volume 31, pp. 3753–3763. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/hash/dc5d637ed5e62c36ecb73b654b05ba2a-Abstract.html.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, et al. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, pp. 94327–94354, 2026.

Viraj Prabhu, Yutong Dai, Matthew Fernandez, Jing Gu, Krithika Ramakrishnan, Yanqi Luo, Silvio Savarese, Caiming Xiong, Junnan Li, Zeyuan Chen, et al. WALT: Web agents that learn tools. *arXiv preprint arXiv:2510.01524*, 2025.

PrismarineJS. Mineflayer: A Minecraft bot API for Node.js. GitHub repository. URL https://github.com/PrismarineJS/mineflayer.

Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling vision with sparse mixture of experts. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

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. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations (ICLR)*, 2017.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Significant Gravitas. AutoGPT, 2023. URL https://github.com/Significant-Gravitas/AutoGPT. Open-source software.

Elias Stengel-Eskin, Archiki Prasad, and Mohit Bansal. ReGAL: Refactoring programs to discover generalizable abstractions. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 46605–46624. PMLR, July 2024. URL https://proceedings.mlr.press/v235/stengel-eskin24a.html. ICML 2024.

Zhiyuan Sun, Haochen Shi, Marc-Alexandre Côté, Glen Berseth, Xingdi Yuan, and Bang Liu. Enhancing agent learning through world dynamics modeling. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 3534–3568, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.202. URL https://aclanthology.org/2024.findings-emnlp.202/.

Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT Press, 1998.

Richard S Sutton, Doina Precup, and Satinder Singh. Between MDPs and Semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112 (1–2):181–211, 1999.

Mingxing Tan and Quoc V. Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2019.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research (TMLR)*, 2024a.

Haonan Wang, Junfeng Sun, Xingdi Yuan, Ruoyao Wang, and Ziang Xiao. Byte-Sized32Refactored: Towards an extensible interactive text games corpus for LLM world modeling and evaluation. *arXiv preprint arXiv:2509.23979*, 2025a.

Jiongxiao Wang, Qiaojing Yan, Yawei Wang, Yijun Tian, Soumya Smruti Mishra, Zhichao Xu, Megha Gandhi, Panpan Xu, and Lin Lee Cheong. Reinforcement learning for self-improving agent with skill library. *arXiv preprint arXiv:2512.17102*, 2025b.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232. PMLR, July 2024b. URL https://proceedings.mlr.press/v235/wang24h.html. ICML 2024.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Shawn Ma, and Yitao Liang. Describe, explain, plan and select: interactive planning with LLMs enables open-world multi-task agents. *Advances in Neural Information Processing Systems*, 36:34153–34189, 2023.

Zihao Wang, Shaofei Cai, Anji Liu, Yonggang Jin, Jinbing Hou, Bowei Zhang, Haowei Lin, Zhaofeng He, Zilong Zheng, Yaodong Yang, et al. JARVIS-1: Open-world multi-task agents with memory-augmented multimodal language models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 47(3):1894–1907, 2025c.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks. *arXiv preprint arXiv:2504.06821*, 2025d.

Isadora White, Kolby Nottingham, Ayush Maniar, Max Robinson, Hansen Lillemark, Mehul Maheshwari, Lianhui Qin, and Prithviraj Ammanabrolu. Collaborating action by action: A multi-agent LLM framework for embodied reasoning. *arXiv preprint arXiv:2504.17950*, 2025.

Danfei Xu, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, and Silvio Savarese. Neural task programming: Learning to generalize across hierarchical tasks. In *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1–8, 2018.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

Shu Yu and Chaochao Lu. ADAM: An embodied causal agent in open-world environments. In *International Conference on Learning Representations (ICLR)*, 2025.

Amy Zhang, Sainbayar Sukhbaatar, Adam Lerer, Arthur Szlam, and Rob Fergus. Composable planning with attributes. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pp. 5842–5851, 2018.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017.

A Additional experimental figures

This section collects the supporting figures for §4.5 and §4.6, whose quantitative results are reported inline in the main text.

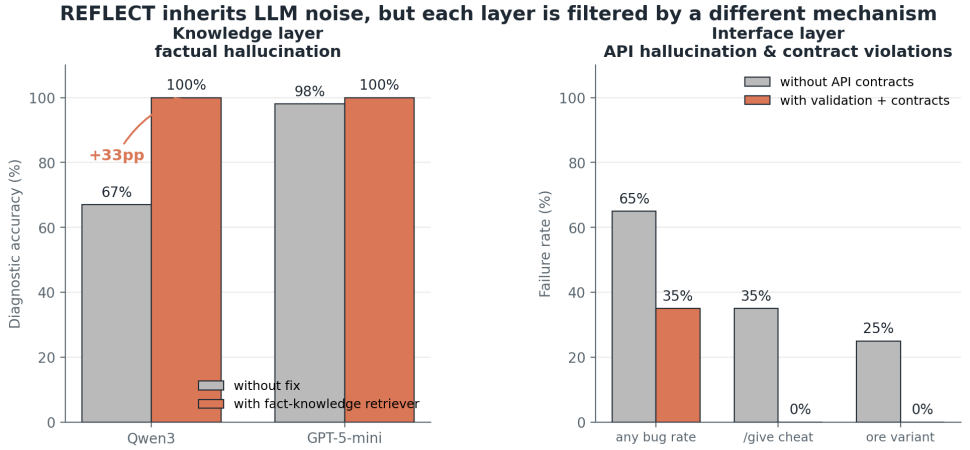
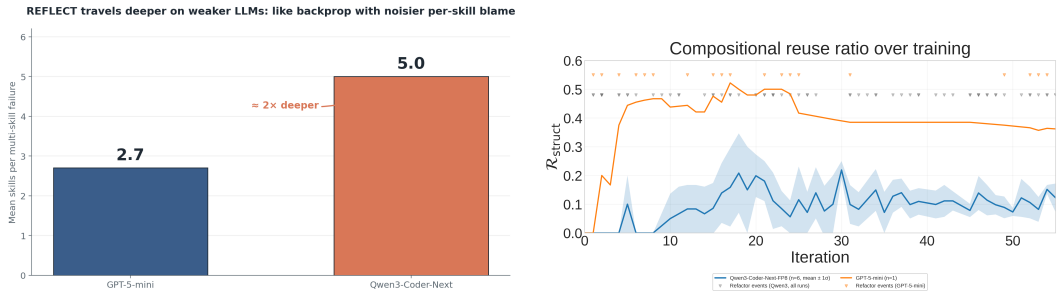


Figure 5: REFLECT inherits LLM noise, but each layer is filtered by a different mechanism (§4.5). **Left (knowledge layer):** factual hallucination is mitigated by a model-specific `tool_tier_rules` corrigendum, worth +33pp on Qwen3 but only +2pp on GPT-5-mini. **Right (interface layer):** fabricated or misused APIs are caught before execution by a static reference checker plus API contracts, lowering the bug rate across both LLMs. The reasoning layer is handled by env-validated iteration (Table 9).



(a) REFLECT depth.

(b) Reuse ratio $\mathcal{R}_{\text{struct}}$.

Figure 6: Two of the three model-dependent signals in §4.6. **(a)** REFLECT propagation depth scales with model strength (5.0 vs 2.7 skills per multi-skill repair), consistent with the chain-rule view of credit assignment (§3). **(b)** Compositional reuse ratio $\mathcal{R}_{\text{struct}}$ over training: the stronger model (GPT-5-mini, single hero run) climbs toward ≈ 0.4, the weaker one (Qwen3-Coder-Next, mean$\pm 1\sigma$ over six runs) plateaus markedly lower.

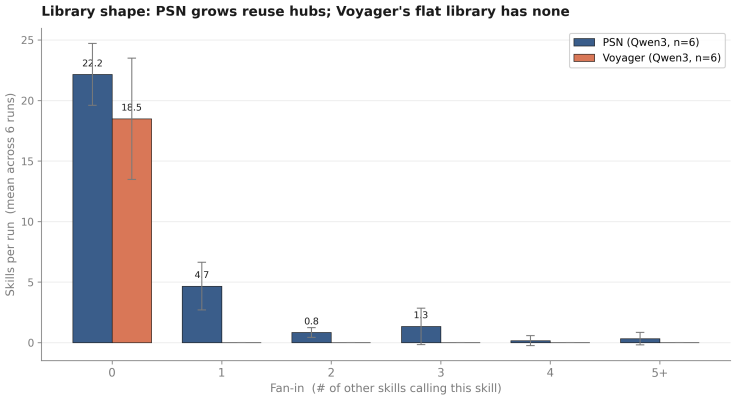


Figure 7: Library shape under the weaker model (Qwen3-Coder-Next), averaged over six runs each (§4.6). PSN grows reuse hubs (skills with fan-in ≥ 1), whereas Voyager's flat retrieval library nearly keeps every skill at fan-in 0.

B Two-phase optimization algorithm of skill optimizer

This section provides a formal algorithmic specification of the two-phase skill optimization process described in the main paper. Algorithm 1 summarizes the complete procedure. A key distinction in our framework is between a skill's *feedback* and its *gradients*: feedback indicates *what went wrong*, while gradients encode *how the skill should be modified*.

B.1 Feedback vs. gradients

For a skill s, we denote by f_s the feedback signal assigned to s after task execution. This feedback may arise from task failure, unmet subgoals, or trace-level diagnostics. Crucially, f_s does not directly specify how to modify s.

Instead, PSN performs a symbolic analysis step that converts feedback into gradients. We denote this process as:

$$\textsc{Reflect}(s, f_s, \mathrm{Subskill}(s)) \;\rightarrow\; \big(g_s,\ \{f_{s'}\}_{s' \in \mathrm{Subskill}(s)}\big),$$

where g_s (also written as $\tilde{\nabla}_s$) is a gradient-like modification proposal for s, and $f_{s'}$ are newly generated feedback signals for each sub-skill invoked by s.

This operation implements a symbolic form of differentiation over the skill invocation structure.

B.2 Phase I: top-down feedback backpropagation

Phase I performs *top-down feedback backpropagation* over the skill network. Starting from a skill that fails to complete a task, PSN recursively applies REFLECT following the invocation relations induced by the execution trace.

At each skill s, symbolic differentiation decomposes f_s into:

- a local gradient proposal g_s describing how s itself should be modified, and
- feedback signals $\{f_{s'}\}$ assigned to sub-skills $s' \in \mathrm{Subskill}(s)$.

This process continues until no further sub-skills require feedback propagation. The result of Phase I is a *pending optimization subgraph* consisting of:

$$\mathcal{G}_{\mathrm{opt}} = \{(s, g_s)\},$$

i.e., a connected subgraph of skills paired with their gradient proposals. No skill code is modified during this phase.

B.3 Phase II: bottom-up gradient application

Phase II applies gradients in a *bottom-up* manner over $\mathcal{G}_{\mathrm{opt}}$. Skills are updated in an order that respects dependency relations, starting from leaf skills and proceeding toward higher-level skills.

For a skill s with gradient proposal g_s, the update is performed via:

$$\textsc{ApplyGradients}\big(s,\ g_s,\ \mathcal{C}_s\big).$$

Here, $\mathcal{C}_s$ is a *context object* that aggregates optimization reports returned by sub-skills that have already been updated. Let

$$\mathcal{S}_s \;:=\; \mathrm{Subskill}(s)$$

denote the set of sub-skills invoked by s. The context $\mathcal{C}_s$ is constructed as:

$$\mathcal{C}_s \;:=\; \textsc{Consider}\big(\textsc{OptimizeReport}(\mathcal{S}_s)\big),$$

which summarizes feedback signals derived from the updated sub-skills.

Updates are realized through program-level rewrite, patch, or diff operations on the skill code. After updating s, the optimizer generates an *optimization report* summarizing the changes and their effects. This report is propagated upward and used to inform subsequent updates of parent skills, allowing higher-level skills to adapt consistently to changes in their dependencies.

Algorithm 1: Two-Phase Skill Optimization in PSN (*Phase I*: top-down feedback back-propagation; *Phase II*: bottom-up gradient application)

Input: Root skill s_{root}, task feedback $f_{s_{\text{root}}}$, execution trace $\mathcal{T}$
Output: Updated skills and optimization reports

Definitions. $\text{Subskill}(s; \mathcal{T})$: sub-skills invoked by s in $\mathcal{T}$;
$\textsc{Reflect}(s, f_s, \text{Subskill}) \to (g_s, \{f_{s'}\})$;
$\textsc{ApplyGradients}(s, g_s, \mathcal{C}) \to (s^+, r_s)$;

Phase I: top-down feedback backpropagation (symbolic differentiation).
Initialize maps $\mathcal{G} \leftarrow \emptyset$ (gradients), $\mathcal{F} \leftarrow \emptyset$ (feedback);
Initialize queue $Q \leftarrow [(s_{\text{root}}, f_{s_{\text{root}}})]$;
while $Q \neq \emptyset$ **do**
  Pop (s, f_s) from Q;
  $\mathcal{F}[s] \leftarrow f_s$;
  $\mathcal{S} \leftarrow \text{Subskill}(s; \mathcal{T})$;
  $(g_s, \{f_{s'}\}_{s' \in \mathcal{S}}) \leftarrow \textsc{Reflect}(s, f_s, \mathcal{S})$;
  $\mathcal{G}[s] \leftarrow g_s$;
  foreach $s' \in \mathcal{S}$ **do**
    if $f_{s'} \neq \varnothing$ **then**
      Push $(s', f_{s'})$ into Q;

Let $\mathcal{H}$ be the induced pending optimization subgraph over $\text{Dom}(\mathcal{G})$;

Phase II: bottom-up gradients application (dependency-respecting updates).
Compute bottom-up order $\pi \leftarrow \textsc{PostOrder}(\mathcal{H})$;
Initialize report map $\mathcal{R} \leftarrow \emptyset$;
foreach *s in* π **do**
  $\mathcal{C} \leftarrow \textsc{Consider}(\{\textsc{OptimizeFeedback}(s') \mid s' \in \text{Subskill}(s) \cap \text{Dom}(\mathcal{R})\})$;
  $(s^+, r_s) \leftarrow \textsc{ApplyGradients}(s, \mathcal{G}[s], \mathcal{C})$;
  Replace $s \leftarrow s^+$ in the skill net;
  $\mathcal{R}[s] \leftarrow r_s$;
return $\{s^+\}$ *and* $\mathcal{R}$;

B.4 Algorithmic interpretation

The complete optimization step thus consists of two strictly separated phases:

- **Phase I:** Top-down symbolic differentiation to propagate feedback $\{f_s\}$.
- **Phase II:** Bottom-up application of gradient proposals $\{g_s\}$.

This design explicitly decouples *credit assignment* from *code modification*. While Phase I follows a chain-rule-like decomposition of feedback signals, Phase II ensures that updates are applied in a dependency-consistent order, preventing interference between skills during optimization.

B.5 Discussion

By separating feedback propagation from gradient application, PSN generalizes the backward–forward separation of neural backpropagation to symbolic, programmatic skill networks. We find this two-phase structure essential for stable optimization in deeply compositional and long-horizon tasks.

C Refactor casebook

This appendix presents a visual casebook of the canonical refactor patterns supported by the Programmatic Skill Network (PSN). Each case corresponds to a distinct structural

Case	Pattern	Example and rewrite	Illustration
(A)	Parametric coverage	**Example:** `mineLogs(type,num)` generalizes `mineOakLogs(num)`. **Rewrite:** `mineOakLogs(num) := mineLogs(OAK,num)`.	Figure 8
(B)	Behavioral / subgraph coverage	**Example:** `craftCraftingTable` inlines routines that exist as skills. **Rewrite:** replace duplicated blocks by calls to `mineLogs` and `craftPlanks`.	Figure 9
(C)	Sibling specializations	**Example:** `mineOakLogs(num)` and `mineBirchLogs(num)` indicate a missing abstraction. **Rewrite:** synthesize `mineLogs(type,num)` and rewrite both as wrappers.	Figure 10
(D)	Extract common subskill	**Example:** both `craftSticks` and `craftTable` require `ensurePlanks(k)`. **Rewrite:** extract `ensurePlanks(k)` as a new skill and replace both occurrences by a call.	Figure 11
(E)	Duplication	**Example:** two skills are near-identical up to naming/surface variations. **Rewrite:** keep higher-$V(s)$ canonical skill; redirect incoming links; demote the other to an alias.	Figure 12

Table 2: Index of canonical refactor cases supported by PSN. Each case corresponds to a distinct structural relationship and rewrite rule, with detailed illustrations provided in Appendix C.

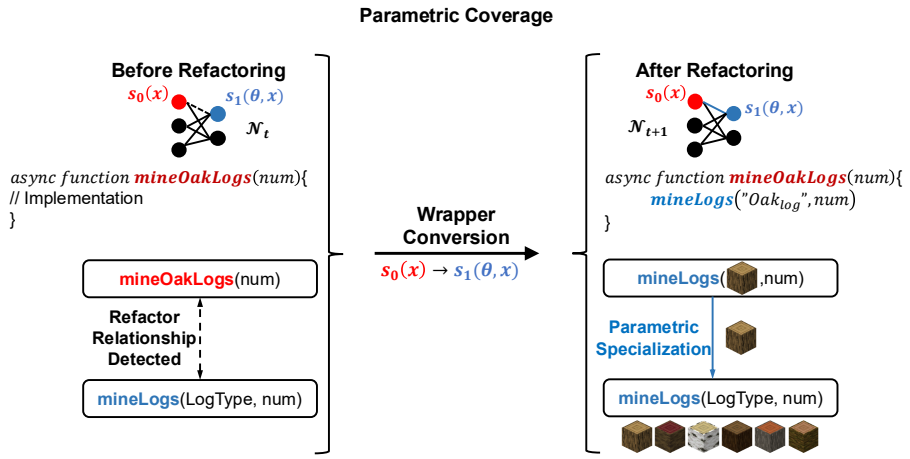


Figure 8: Parametric coverage. A specialized skill is rewritten as a wrapper around a more general, parameterized skill.

relationship between skills and induces a deterministic graph rewrite. All cases referenced in Section 2.5 are illustrated in the Table 2 and below.

These refactor cases are exhaustive with respect to the structural patterns observed in our experiments.

C.1 Case A: parametric coverage

Pattern. One skill is a strict specialization of another skill that admits a parameterized generalization.

Rewrite. The specialized skill is replaced by a thin wrapper that calls the generalized skill with fixed parameter values.

C.2 Case B: behavioral / subgraph coverage

Pattern. A composite skill reimplements functionality that already exists as an independent skill in the PSN, resulting in duplicated subgraphs.

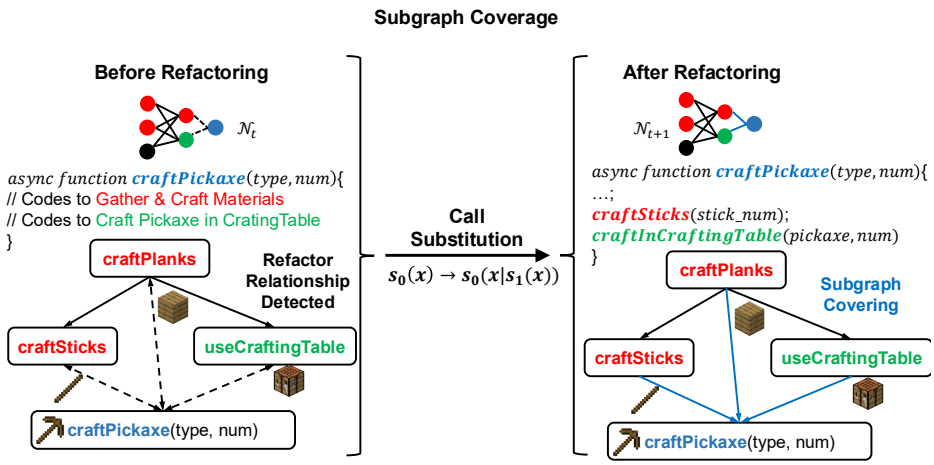


Figure 9: Behavioral (subgraph) coverage. Duplicated logic inside a composite skill is replaced by a call to an existing reusable skill, preserving behavior while reducing redundancy.

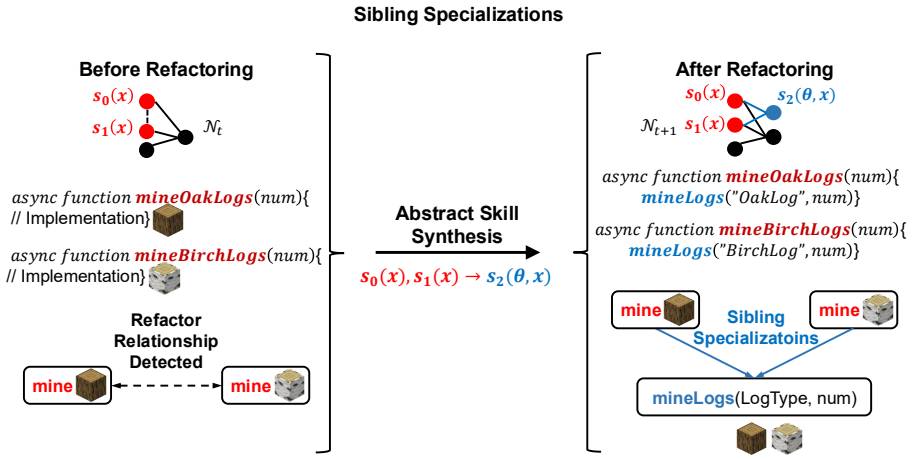


Figure 10: Sibling specializations. Multiple specialized skills expose a missing higher-level abstraction that can be explicitly synthesized and reused.

Rewrite. The duplicated subgraph is removed and replaced by a direct invocation of the existing skill, yielding a simpler and more compositional program structure.

C.3 Case C: sibling specializations

Pattern. Two or more skills are specializations of a latent, more general operation that is not yet represented as a standalone skill in the network.

Rewrite. A new abstract skill is synthesized to capture the shared structure, and all specialized skills are rewritten as thin wrappers that invoke the abstract skill with appropriate parameters.

C.4 Case D: common subskill extraction

Pattern. Multiple skills contain an identical or highly similar sub-operation that is implemented independently within each skill.

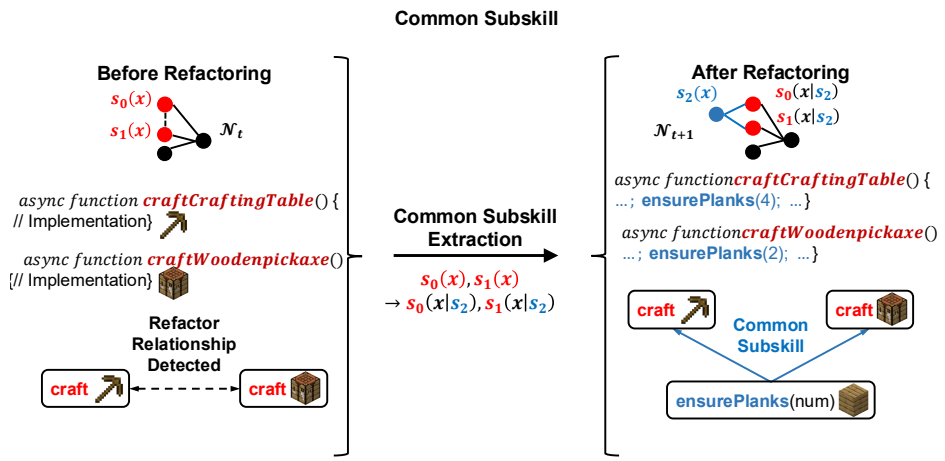


Figure 11: Common subskill extraction. Repeated sub-operations across different skills are factored into a shared subskill, improving reuse and reducing duplication.

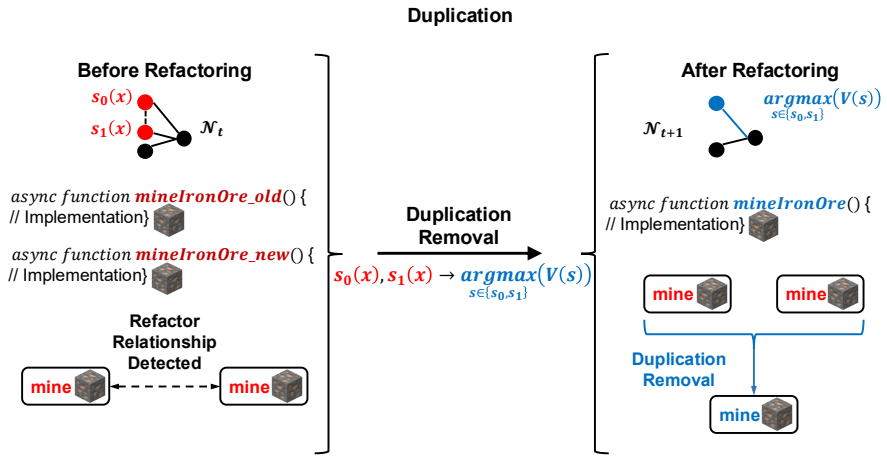


Figure 12: Duplication removal. Functionally equivalent skills are merged into a single canonical representation.

Rewrite. The shared subgraph is extracted into a new reusable skill, and all original skills are rewritten to invoke this subskill instead of duplicating its logic.

C.5 Case E: duplication removal

Pattern. Two skills are functionally equivalent up to naming differences or minor surface variations, leading to redundant representations in the PSN.

Rewrite. The skill with higher empirical value is retained as the canonical implementation, and all invocation links to the redundant skill are redirected. The redundant skill is demoted to an alias or removed from planning.

Operator	Domain → Codomain	Semantic role	Example in PSN
EXECUTE	$(s, E) \to (f_s, \delta_s)$	Symbolic forward operator that executes a skill program s in environment E, producing structured feedback f_s and a success flag $\delta_s \in \{0, 1\}$.	EXECUTE($s_{\texttt{craftTable}}$) runs the composed skill `craft crafting table`, records the invocation trace and state transitions, and returns whether the goal predicate g_τ is satisfied.
REFLECT	$(f_s, s) \to \widetilde{\nabla}_s$	Symbolic differentiation operator that performs top-down credit assignment over the PSN, yielding a finite, localized symbolic pseudo-gradient $\widetilde{\nabla}_s = \partial_s f_s$. The operator identifies faulty control flow, misaligned parameters, incorrect preconditions, or subskill effects, and serves as a discrete, structural analogue of backpropagation in neural networks.	REFLECT applied to `craftTable` detects that it failed due to missing planks and proposes edits to collect wood logs and craft planks for crafting CraftingTable.

Table 3: Symbolic operators defining forward execution and backward credit assignment over program-structured skills in the PSN.

D Operator summary

D.1 Symbolic operators

Table 3 summarizes the core symbolic operators used in the Programmatic Skill Network (PSN), which define the symbolic forward and backward passes over program-structured skills.

D.2 System operators

Table 4 summarizes the system-level operators that orchestrate planning, learning, and structural evolution of the Programmatic Skill Network (PSN).

E Example prompt templates

This appendix provides example prompt templates used to instantiate PSN operators in our implementation. We emphasize that PSN does not rely on specific prompt wording; the examples below serve only as concrete realizations of the abstract operator interfaces defined in Section 2.

E.1 REFLECT operator

The example prompt for the REFLECT operator is demonstrated in Figure 13. Note that, to accelerate the speed of REFLECT, we implement a hybrid REFLECT operator that combines the LLM REFLECT with a rule-based REFLECT function that extracts frequent patterns recognized by LLM REFLECT as a set of rules.

Input.

- Skill name and implementation code
- Execution feedback and failure signals

Operator	Domain → Codomain	System role	Example in PSN
PLAN	$(g_{\tau_t}, \mathcal{N}_t) \to P_t^{\text{LLM}}$	Fallback forward planner invoked when backward-chaining over existing skills cannot ground a subgoal, producing exploratory plans beyond the current PSN.	For the task "`obtain diamond`", PLAN proposes a long-horizon plan involving mining iron, smelting ingots, crafting pickaxes, and mining diamond ore.
CODEGEN	$(P_t, \text{Context}_t) \to s_t$	Skill synthesis operator that distills a high-level plan into a new symbolic skill neuron with control flow, parameters, and pre/postconditions.	Given a plan P_t = [`getWood`, `craftPlanks`, `craftTable`], CODEGEN creates a reusable skill `craftCraftingTable` with an explicit loop and parameterized inventory checks.
OPTIMIZE	$(\mathcal{N}_t, s_t, f_t) \to \mathcal{N}_{t+1}$	Skill optimizer that applies symbolic backpropagation when a task fails, repairing the faulty subnetwork $\mathbb{N}(s_t)$ via REFLECT.	If `craftStonePickaxe` fails due to insufficient cobblestone, OPTIMIZE propagates symbolic edits to `mineCobblestone`, inserting a loop until enough stone is collected.
REFACTOR	$(\mathcal{N}_t, s_t, f_t) \to \mathcal{N}_{t+1}$	Online structural refactor operator that performs symbolic neural architecture search (NAS) when a task succeeds, merging, abstracting, pruning, and rewiring skills.	After learning both `mineOakLogs` and `mineBirchLogs`, REFACTOR synthesizes a generalized `mineLogs(log_type, num)` and rewrites both original skills as wrappers.
embed	$s \mapsto \text{embed}(s)$	Semantic embedding operator used for similarity-based retrieval during refactor, enabling detection of related skills beyond local graph neighborhoods.	High similarity between $\text{embed}(s_{\texttt{craftStick}})$ and $\text{embed}(s_{\texttt{craftTable}})$ helps identify a common subroutine for ensuring plank availability.
$P(\text{update } s)$	$V(s) \mapsto [0, 1]$	Maturity-aware update gate that controls how frequently symbolic derivatives are applied to a skill, stabilizing mature skills while keeping immature ones plastic.	For a navigation skill with high $V(s)$, $P(\text{update } s)$ becomes small, so OPTIMIZE rarely modifies it; newly synthesized skills are updated aggressively until they stabilize.

Table 4: System-level operators that orchestrate planning, optimization, and structural evolution of the PSN.

- Optional execution state, environment context, and child-skill information

Output. A structured JSON record containing:

- Self-responsible issues with gradient type, magnitude, and direction
- Child-skill attributions with responsibility weights
- Concrete code-level modification suggestions

```

**Skill:** {input.skill_name}

**Code:**
```javascript
{input.skill_code}
```

Feedback:
{input.feedback_content}

Feedback Type: {input.feedback_type}
{execution_state_section}
{children_section}
{env_section}
{primitive_section}
{propagated_section}
{api_knowledge_section}
{reasoning_examples_section}

Analysis Tasks:
1. Identify the root cause of the failure
2. Determine if the issue is in THIS skill or in a child skill
3. For each identified issue, specify:
   - The type of gradient (logic, parameter_semantic, physical_constraint, error_handling, etc.)
   - The magnitude (0.0 to 1.0, higher = more urgent)
   - The direction (what needs to change)
   - The suggested_fix (REQUIRED: concrete code modification suggestions)

IMPORTANT: For physical_constraint issues (placement, resource depletion, pathfinding):
- Provide SPECIFIC code changes in suggested_fix
- Example: "Expand maxDistance from 6 to 16, expand vertical search from [-1,1] to [-2,2]"

Return JSON:
{{
    "self_issues": [
        {{
            "gradient_type": "logic|parameter_semantic|physical_constraint|error_handling|interface",
            "magnitude": 0.0-1.0,
            "direction": "what needs to change",
            "evidence": "supporting evidence from feedback",
            "suggested_fix": "REQUIRED: specific code changes to make"
        }}
    ],
    "child_issues": [
        {{
            "child_skill": "name",
            "issue_description": "...",
            "responsibility": "...",
            "weight": 0.0-1.0
        }}
    ],
    "reasoning": "overall analysis"
}}
```

Figure 13: Example prompt template instantiating the REFLECT operator.

E.2 Skill optimization operator

We instantiate the skill optimization operator as a patching procedure $s \leftarrow \textsc{Patch}(s, \tilde{\nabla}_s)$, where $\tilde{\nabla}_s$ is a structured set of issues and modification directions produced by REFLECT. The operator consumes a skill implementation together with layered constraints and execution feedback, and outputs a revised implementation along with an explicit requirement-by-requirement audit trail for mandatory fixes. The detailed prompt is demonstrated in Figure 14.

```
=== SYSTEM ===
You are a helpful assistant that optimizes Minecraft skill code.

READ THE LAYERED CONTEXT CAREFULLY!
The context is organized in layers of importance:
- LAYER 1 (MUST FIX): Critical issues that MUST be addressed. Your code will be REJECTED if not fixed.
- LAYER 2 (LOCALIZATION): Specific lines and areas to focus on.
- LAYER 3 (CONSTRAINTS): Rules you must follow (don't change signature, don't redefine external skills).

CRITICAL RULES:
1. Fix ALL issues mentioned in LAYER 1 - these are mandatory
2. Focus your changes on the areas mentioned in LAYER 2
3. Follow ALL constraints in LAYER 3
4. Return COMPLETE code with all brackets matched - do NOT truncate
5. Keep the function signature unchanged
6. Do NOT add new functions with same names as external skills
7. AUTOMATION ONLY - We only support fully automated skills:
   - Use Mineflayer APIs (bot.craft, bot.dig, bot.placeBlock, bot.equip, etc.)
   - Do NOT require user interaction (windowOpen events, "press E", manual operations)
   - Do NOT convert automated code to interactive/manual flows
   - All operations must be programmatic and automatic
8. CODE CONCISENESS: Keep code concise. Do NOT add unnecessary helper functions.
   - Only keep helper functions that are ACTUALLY USED
   - Remove redundant code. If optimized code is longer than original, review and simplify.
9. DO NOT REDEFINE SYSTEM CONTROL PRIMITIVES: The following functions are PROVIDED BY THE SYSTEM.
   DO NOT create local functions with these exact names - they already exist externally:

   mineBlock, craftItem, smeltItem, exploreUntil, placeItem,
   killMob, useChest, givePlacedItemBack, shoot, waitForMobRemoved

   CONTROL PRIMITIVE API SIGNATURES (CRITICAL - Parameter Types):
{primitives_knowledge}

SIMPLIFICATION PRINCIPLE (MANDATORY - Code Bloat Prevention):
{simplification_principle}

ENVIRONMENT KNOWLEDGE AWARENESS:
{environment_knowledge}

Return a JSON object:
{
  "issues": [
    { "type": "issue_type", "description": "brief description" }
  ],
  "optimized_code": "complete optimized code in JavaScript",
  "change_summary": "brief description of changes",
  "requirements_addressed": [
    {
      "requirement_index": 1,
      "how_addressed": "how LAYER 1 requirement was addressed",
      "code_location": "line number or function name"
    }
  ]
}

The "requirements_addressed" field is MANDATORY!
You must explain how EACH requirement from LAYER 1 was addressed.

=== HUMAN ===
Skill: {skill_name}

{edit_context}

FULL CODE (for reference):
{skill_code}
{wrapper_warning}

ADDITIONAL CONTEXT:
Skill description:
{skill_description}

Gradient:
{gradient_summary}

Child skills feedback:
```

```
{child_feedback_summary}

{forward_propagation_info}
{current_state_info}

Recent optimization history (last {momentum_window} feedbacks):
{optimization_history}

Statistics:
- Total executions: {total_executions}
- Success rate: {success_rate}
- Failed executions: {failed_executions}

CODE FORMATTING REQUIREMENTS:
- The optimized_code MUST be properly formatted with:
  - One statement per line
  - Proper indentation (2 spaces)
  - Newlines after { and before }
  - DO NOT compress multiple statements into a single line

Return only JSON.
```

Figure 14: Example prompt template instantiating the skill optimization operator ($s \leftarrow \text{PATCH}(s, \tilde{\nabla}_s)$) as a constrained program-repair step.

F Inline knowledge scaffolding ablation

PSN's optimizer Phase 1 prompt builder retrieves and injects inline knowledge from a flat index of ∼60 `KnowledgeItem` dataclass instances covering Mineflayer API behavior (`api/pathfinder.py`, `api/bot_methods.py`, `api/inventory.py`) and high-level action guidance (`action_guidance.py`). Each item has a `fact` field (the core technical statement) and an optional `logical_implications` field listing deductions, diagnostic patterns, or strategy hints. The `logical_implications` content is rendered into the REFLECT prompt under a "Logical Implications:" header alongside the `fact`.
This appendix ablates the `logical_implications` field to test whether REFLECT's diagnostic capability comes from the architectural mechanism (trace-based credit assignment over execution traces) or from these injected hints.

Methodology. We use a *stored-prompt A/B* design: instead of running fresh end-to-end training, we replay Phase 1 production prompts (drawn from a previous Minecraft training run) through the live LLM under two conditions:

- **WITH_SCAFFOLDING**: full Phase 1 prompt with all `logical_implications` content rendered.
- **WITHOUT_SCAFFOLDING**: same prompt with `logical_implications` stripped from 52 of 56 items. Three items are retained: one weak-LLM corrigendum (`tool_tier_rules`), one production-tested fix (`bot.inventory.items`, whose `.reduce()` pattern guards against a known stacked-items counting bug), and one pure API contract reference (`checkRecipe_api_contract`).

Cases include 5 real Phase 1 mining/crafting cases where the actual fix direction is well-known (planted decoys for retrieval-mis-fire bias detection) and 3 synthetic tier-mismatch cases (`stone_pickaxe + diamond_ore`, `iron_pickaxe + obsidian`, `wooden_pickaxe + iron_ore`) where the correct diagnosis is "tool tier insufficient." Each (case, condition) pair is sampled repeatedly at temperature=1.0, and we report the fraction of correct diagnoses per condition.

Cross-LLM results. The `tool_tier_rules` corrigendum has sharply different value on the two models. On Qwen3-Coder-Next, removing it drops diagnostic correctness on the synthetic tier-mismatch cases from 100% to 67%, a +33pp contribution: the weaker model has genuine Minecraft pretraining gaps, constructing context-anchored hallucinations such as "lava destroys obsidian" or "raw_iron does not exist in vanilla Minecraft," and the corrigendum patches them. On GPT-5-mini the same corrigendum contributes only +2pp (the model scores near-100% with it and 98% without it): the strong model already knows the tier rules, so the scaffolding is almost inert. The contribution is documented in the codebase via an explicit `source` field annotation and is independent of the architectural mechanisms.

Baseline cleanliness. The Qwen3 figure is measured against a cleaned baseline. An earlier baseline left the 52 sibling `logical_implications` items in place; several indirectly cued tier-mismatch reasoning, inflating the no-corrigendum accuracy to 83% and under-reporting the rule's true contribution. Removing those siblings lowers the intrinsic-knowledge baseline to 67% and reveals the full +33pp. GPT-5-mini's baseline is insensitive to this prune (it stays near-100% either way), so its +2pp is unaffected.

Implications. On the strong-LLM evaluation, REFLECT's diagnostic capability is essentially architectural: removing all inline knowledge scaffolding leaves diagnostic correctness nearly unchanged (a +2pp gap). The Qwen3 corrigendum is honest scaffolding for a weaker model with documented Minecraft knowledge gaps, not a hidden algorithmic dependency. This decomposition strengthens the architectural-origin claim of Section 2.1 by providing direct ablation evidence: the "rather than prompt-level tricks" assertion in Section 4.1 is now verified at the prompt level, not just inferred from end-to-end performance.

G Additional optimization examples

This appendix provides representative examples of execution-level optimizations performed by PSN. All examples are drawn from actual training runs and are selected to illustrate recurring optimization patterns rather than to exhaustively enumerate all repairs. Together, they demonstrate how trace-based symbolic credit assignment enables both localized fixes and coordinated optimization across skill hierarchies. **Complete code diffs for optimization cases are provided in Section H.**

G.1 Optimization taxonomy

Across experiments, frequent optimizations of PSN fall into several recurring categories. Table 5 summarizes the most common failure signals and corresponding repair strategies.

Category	Failure Signal	Typical Repair
Resource miscalculation	insufficient materials	Correct resource accounting
Unsafe fallback	silent execution failure	Enforce fail-fast behavior
Boundary condition	inventory full	Add capacity-aware constraints
Missing preconditions	missing crafting station	Explicit precondition validation
API misuse	invalid recipe or action	Correct API invocation
Cross-skill contract	downstream semantic failure	Parent–child co-optimization

Table 5: Common optimization patterns discovered and repaired by PSN.

G.2 Representative optimization cases

Example 1: Resource Miscalculation (`craftWoodenPickaxe`). **Failure signal.** The skill fails during execution with an error indicating insufficient wooden planks. **Root cause.** The original implementation underestimates required resources by ignoring planks consumed during intermediate stick crafting. **Repair.** Using execution traces, PSN localizes the failure to the resource calculation logic and updates the material requirements to account for intermediate crafting steps. A validation check is added before execution to ensure sufficient materials are available. **Outcome.** After repair, the skill reliably computes correct resource requirements and succeeds across repeated executions.

Example 2: Unsafe Fallback (`ensureFlint`). **Failure signal.** The skill exhibits silent or inconsistent failures when attempting to mine gravel. **Root cause.** An unsafe fallback bypasses the system's primitive execution contract, preventing proper failure propagation to the planner. **Repair.** PSN removes the unsafe fallback and enforces fail-fast behavior, ensuring that execution failures are explicitly surfaced and handled by upstream skills. **Outcome.** The repaired skill behaves consistently and enables reliable replanning under failure.

Example 3: Boundary Condition (`openChestAndRetrieve`). **Failure signal.** Execution fails when attempting to retrieve items from a chest due to insufficient inventory capacity. **Root cause.** The skill assumes unlimited inventory space and does not model capacity constraints. **Repair.** The optimizer inserts an explicit capacity check and dynamically constrains the withdrawal amount based on available inventory slots. **Outcome.** The optimized skill adapts to varying inventory states and avoids execution-time errors.

Example 4: Missing Preconditions (`ensureMetalIngots`). **Failure signal.** The skill fails when attempting to smelt metal ingots without access to a crafting table or furnace. **Root cause.** The original implementation relies on implicit assumptions about environmental setup. **Repair.** PSN makes these assumptions explicit by validating the presence of required crafting stations and inserting corrective actions to locate or construct them when missing. **Outcome.** The repaired skill succeeds robustly across diverse environment configurations.

G.3 Advanced optimization: cross-skill credit assignment

Beyond single-skill repairs, PSN is able to propagate optimization signals across skill boundaries.In particular, failures in a parent skill can trigger coordinated updates to both the parent and its dependent subskills.

Example 5: Parent–Child Co-Optimization (`ensureRawIronAndFuel` → `ensureFuel`). **Context.** The parent skill `ensureRawIronAndFuel` invokes the subskill `ensureFuel` to acquire sufficient fuel before mining and smelting iron. **Failure signal.** Execution traces show that the parent skill proceeds despite insufficient fuel being present in the inventory, leading to cascading failures in downstream steps. **Root cause.** The parent skill implicitly assumes that successful completion of `ensureFuel` guarantees the availability of the required fuel.However, the subskill employs coarse fallback behaviors and does not explicitly verify that the desired fuel items are obtained. **Coordinated repair.** PSN assigns credit to both levels of the skill hierarchy and performs simultaneous optimizations:

- **Parent skill repair:** the parent skill is updated to explicitly verify postconditions after invoking the subskill, checking for the presence of coal or charcoal and triggering targeted recovery actions when verification fails.
- **Subskill repair:** the subskill `ensureFuel` is refined to reduce overly coarse fallbacks, prioritize specific fuel types, and handle inventory-capacity constraints more robustly.

Outcome. After co-optimization, the parent skill reliably enforces its fuel preconditions, and the refined subskill consistently delivers the required resources. This example demonstrates PSN's ability to localize responsibility across skill boundaries and to perform coordinated, semantics-preserving optimization over compositional skill hierarchies.

H Detailed code diffs for optimization examples

This section provides complete code diffs for the representative optimization cases described in Section G. Table 6 summarizes all cases, and Table 7 shows the mapping from gradient signals to implemented fixes.

Skill	Bug Type	Error Pattern	Key Fix
`craftWoodenPickaxe`	Resource Calc	`insufficient materials`	Count planks for sticks
`ensureFlint`	Unsafe Fallback	`Invalid token`	Remove bot.dig() fallback
`openChestAndRetrieve`	Boundary	`destination full`	Pre-check capacity
`ensureMetalIngots`	Precondition	`requires crafting table`	Validate & place table

Table 6: Summary of optimization cases with bug types and key fixes.

Gradient Signal	Interpretation	Resulting Fix
"Fix resource_management"	Math error in counting	Add plank calculation for sticks
"fail loudly rather than fallback"	Unsafe silent failure	Replace fallback with explicit error
"Limit withdraw amounts"	Boundary violation	Add capacity calculation
"guarantee crafting table present"	Missing precondition	Add validation and placement logic

Table 7: Mapping from gradient signals to implemented fixes.

H.1 Example 1: craftWoodenPickaxe (resource miscalculation)

Failure signal.

```
Error: Cannot craft wooden_pickaxe: insufficient planks. Needed 3, have 0.
```

Root cause. The original implementation underestimates required resources by ignoring planks consumed during intermediate stick crafting.

Gradient signal.

```
{"gradient_type": "resource_management", "magnitude": 0.9,
 "direction": "Fix plank calculation to include planks consumed by stick crafting"}
```

Code diff.

```diff
--- craftWoodenPickaxe.js (original)
+++ craftWoodenPickaxe.js (optimized)
@@ -26,9 +24,19 @@

-    // 2) Ensure we have a crafting_table item in inventory
+    // 2) Ensure crafting_table item by invoking external skill
     bot.chat("No nearby crafting table. Ensuring crafting_table...");
-    await ensureCraftingTable(bot, 1);
+    await ensureCraftingTable(bot, 1, plankType);
+
+    // 2b) Validate ensureCraftingTable result
+    const tableCount = countItemByName("crafting_table");
+    const tableBlock2 = bot.findBlock({
+      matching: mcData.blocksByName["crafting_table"].id,
+      maxDistance: maxDistance
+    });
+    if (tableCount <= 0 && !tableBlock2) {
+      throw new Error("ensureCraftingTable failed.");
+    }
```

```diff
@@ -126,9 +135,11 @@

-    // Totals needed
-    const totalPlanksNeeded = count * planksPerPick;
+    // FIXED: include planks consumed to craft sticks
     const totalSticksNeeded = count * sticksPerPick;
+    const stickRecipesNeeded = Math.ceil(totalSticksNeeded / 4);
+    const planksNeededForSticks = stickRecipesNeeded * 2;
+    const totalPlanksNeeded = (count * planksPerPick) + planksNeededForSticks;
```

```diff
@@ -175,11 +184,44 @@

+    // Recompute planks after crafting sticks
+    havePlanks = countItemByName(plankName);
+    if (havePlanks < totalPlanksNeeded) {
+      const missingPlanks = totalPlanksNeeded - havePlanks;
+      const craftsNeeded2 = Math.ceil(missingPlanks / 4);
+      bot.chat(`Need ${missingPlanks} more ${plankName}.`);
+      for (let j = 0; j < craftsNeeded2; j++) {
+        const check2 = bot.checkRecipe(plankName, 1, null);
+        if (!check2.available) {
+          throw new Error(`Cannot craft ${plankName}: ${check2.message}`);
+        }
+        await bot.craft(check2.recipe, 1, null);
+        await bot.waitForTicks(2);
+      }
+    }

     // 4) Craft wooden_pickaxe at the crafting table
     for (let i = 0; i < count; i++) {
```

```
+       // Re-validate resources prior to each craft
+       havePlanks = countItemByName(plankName);
+       haveSticks = countItemByName(stickName);
+       if (havePlanks < planksPerPick) {
+         throw new Error(`Insufficient planks: ${havePlanks}/${planksPerPick}`);
+       }
+       if (haveSticks < sticksPerPick) {
+         throw new Error(`Insufficient sticks: ${haveSticks}/${sticksPerPick}`);
+       }
        const check = bot.checkRecipe("wooden_pickaxe", 1, craftingTableBlock);
```

H.2 Example 2: ensureFlint (unsafe fallback)

Failure signal.

```
bot.dig failed: Invalid token
Path to gravel failed: Cannot read property 'position' of null
```

Root cause. An unsafe fallback using `bot.dig()` directly bypasses the system's primitive execution contract, preventing proper failure propagation.

Gradient signal.

```
{"gradient_type": "error_handling", "magnitude": 0.85,
 "direction": "fail loudly rather than naive fallback"}
```

Code diff.

```diff
--- ensureFlint.js (original)
+++ ensureFlint.js (optimized)
@@ -2,13 +2,12 @@

-  // find a nearby gravel block (within 32)
+  // find a nearby gravel block (within SEARCH_RADIUS)
+  const SEARCH_RADIUS = 48;
   function findNearbyGravel() {
     const gravelDef = mcData.blocksByName['gravel'];
     if (!gravelDef) return null;
     return bot.findBlock({
       matching: gravelDef.id,
-      maxDistance: 32
+      maxDistance: SEARCH_RADIUS
     });
   }
```

```diff
@@ -17,61 +16,37 @@

-  // Attempt to mine gravel using mineBlock if available, otherwise fallback
+  // Attempt to mine gravel using mineBlock control primitive
+  // If unavailable, fail loudly so harness can surface the error
   async function mineOneGravelAt(blockPos) {
-    if (typeof mineBlock === "function") {
-      await mineBlock(bot, "gravel", 1);
+    if (typeof mineBlock === 'function') {
+      await mineBlock(bot, 'gravel', 1);
       return;
     }
-    // Fallback manual approach:
-    const targetBlock = bot.blockAt(blockPos);
-    if (!targetBlock) throw new Error("Target gravel disappeared.");
-    try {
-      await bot.pathfinder.goto(new GoalGetToBlock(
-        targetBlock.position.x, targetBlock.position.y, targetBlock.position.z));
-    } catch (e) {
-      bot.chat(`Path to gravel failed: ${e.message}.`);
-    }
-    try {
-      await bot.dig(targetBlock, true);
-    } catch (e) {
-      bot.chat(`dig failed: ${e.message}`);
-      throw e;
-    }
-    await bot.waitForTicks(4);
+    // Deliberately fail if primitive unavailable
+    throw new Error('Required primitive mineBlock is not available.');
   }
```

```
@@ -101,10 +78,12 @@

     try {
       await mineOneGravelAt(nearby.position);
     } catch (e) {
+      // Propagate fatal errors for proper handling
       bot.chat(`Failed to mine gravel: ${e.message}`);
+      throw e;
     }
```

H.3 Example 3: openChestAndRetrieve (boundary condition)

Failure signal.

```
Error: Destination full while withdrawing items from chest
```

Root cause. The skill assumes unlimited inventory space and does not model capacity constraints.

Gradient signal.

```
{"gradient_type": "physical_constraint", "magnitude": 0.8,
 "direction": "Limit withdraw amounts based on available inventory capacity"}
```

Code diff.

```diff
--- openChestAndRetrieve.js (original)
+++ openChestAndRetrieve.js (optimized)
@@ -1,13 +1,38 @@

+  // Helper: compute available inventory space for a specific item
+  function availableInventorySpaceFor(itemId) {
+    const def = mcData.items[itemId] || {};
+    const maxStack = def.stackSize || 64;
+    let free = bot.inventory.emptySlotCount() * maxStack;
+    for (const slot of bot.inventory.items()) {
+      if (slot.type === itemId) {
+        free += (maxStack - slot.count);
+      }
+    }
+    return free;
+  }
+
+  // Helper: get container items across MC versions
+  function getContainerItems(win) {
+    if (!win) return [];
+    if (typeof win.containerItems === 'function') return win.containerItems();
+    if (Array.isArray(win.slots)) return win.slots.filter(Boolean);
+    try { return win.items(); } catch (e) { return []; }
+  }
```

```diff
@@ -65,56 +88,81 @@

+        // Compute how many items are actually in the chest
+        const availableInChest = containerItems
+          .filter(i => i && i.type === itemDef.id)
+          .reduce((s, i) => s + (i.count || 0), 0);
+
+        const capacity = availableInventorySpaceFor(itemDef.id);
+        const toWithdraw = Math.min(want, availableInChest, capacity);
+
+        if (toWithdraw <= 0) {
+          await bot.chat(`No space or chest lacks ${name}, skipping.`);
+          withdrawn[name] = bot.inventory.count(itemDef.id, null);
+          continue;
+        }

         try {
-          await chestWindow.withdraw(itemDef.id, null, want);
+          await chestWindow.withdraw(itemDef.id, null, toWithdraw);
           await bot.waitForTicks(3);
+          withdrawn[name] = bot.inventory.count(itemDef.id, null);
         } catch (err) {
+          if (err.message.includes('destination full')) {
+            const controlled = new Error('Destination full');
+            controlled.code = 'DESTINATION_FULL';
+            throw controlled;
```

```
+           }
            throw err;
          }
```

H.4 Example 4: ensureMetalIngots (missing precondition)

Failure signal.

```
Error: No furnace recipe available (missing materials).
Error: Failed to find or place a crafting table before crafting furnace.
```

Root cause. The original implementation relies on implicit assumptions about environmental setup without validating the presence of required crafting stations.

Gradient signal.

```
{"gradient_type": "precondition", "magnitude": 0.9,
 "direction": "guarantee crafting table is present before furnace craft"}
```

Code diff.

```diff
--- ensureMetalIngots.js (original)
+++ ensureMetalIngots.js (optimized)
@@ -68,81 +70,144 @@

+      // If no placed table and no table item, craft one (2x2 recipe)
+      if (!craftingBlock && !craftingItemInv && craftingItemDef) {
+        try {
+          const tableRecipes = bot.recipesFor(craftingItemDef.id, null, null) ||
     [];
+          if (tableRecipes.length > 0) {
+            await bot.craft(tableRecipes[0], 1, null);
+            await bot.waitForTicks(4);
+            craftingItemInv = bot.inventory.findInventoryItem(craftingItemDef.id);
+          }
+        } catch (e) {
+          bot.chat(`Crafting crafting_table failed: ${e.message}`);
+        }
+      }
+
+      // If we have table item but no placed block, place it
+      if (!craftingBlock && craftingItemInv) {
+        const botFoot = bot.entity.position.floored();
+        const searchOffsets = [
+          new Vec3(1, 0, 0), new Vec3(-1, 0, 0),
+          new Vec3(0, 0, 1), new Vec3(0, 0, -1),
+        ];
+        let candidate = null;
+        for (const off of searchOffsets) {
+          const cand = botFoot.offset(off.x, off.y, off.z);
+          if (cand.equals(botFoot)) continue;
+          candidate = cand;
+          break;
+        }
+
+        try {
+          await bot.pathfinder.goto(new GoalPlaceBlock(candidate, bot.world, {}));
+        } catch (e) {
+          bot.chat(`Path to table spot failed: ${e.message}`);
+        }
+
+        await bot.equip(craftingItemInv, "hand");
+        await bot.placeBlock(ref, new Vec3(0, 1, 0));
+        await bot.waitForTicks(4);
+        craftingBlock = bot.blockAt(candidate);
+
+        // Verify placement succeeded
+        if (!craftingBlock || craftingBlock.name !== "crafting_table") {
+          throw new Error("Crafting table placement failed.");
+        }
+      }
+
```

```
+      // Ensure placed crafting table before furnace craft
+      if (!craftingBlock) {
+        throw new Error("Failed to find or place a crafting table.");
+      }
+
+      // Move within interaction distance
+      try {
+        await bot.pathfinder.goto(new GoalNear(
+          craftingBlock.position.x, craftingBlock.position.y,
+          craftingBlock.position.z, 2));
+      } catch (e) {
+        bot.chat(`Could not move near crafting table: ${e.message}`);
+      }
+
+      // Use recipesFor instead of checkRecipe
+      if (!furnaceItemDef) throw new Error('Furnace definition missing.');
+      const furnaceRecipes = bot.recipesFor(furnaceItemDef.id, null, craftingBlock)
     || [];
+      if (furnaceRecipes.length === 0) {
+        throw new Error('No furnace recipe available.');
+      }
+      try {
+        await bot.craft(furnaceRecipes[0], 1, craftingBlock);
+        await bot.waitForTicks(4);
+      } catch (e) {
+        throw new Error(`Crafting furnace failed: ${e.message}`);
+      }
```

H.5 Example 5: cross-skill co-optimization

Beyond single-skill repairs, PSN propagates optimization signals across skill boundaries. This example shows coordinated parent–child optimization between `ensureRawIronAndFuel` (parent) and `ensureFuel` (child).

Failure signal. The parent skill proceeds despite insufficient fuel, causing cascading failures in downstream smelting steps.
Coordinated repair. PSN assigns credit to both levels of the hierarchy and performs simultaneous optimizations.

Parent skill repair (`ensureRawIronAndFuel`):

```
+    // Verify fuel postcondition after calling ensureFuel
+    const fuelCount = countItemByName("coal") + countItemByName("charcoal");
+    if (fuelCount < requiredFuel) {
+      bot.chat(`ensureFuel insufficient: ${fuelCount}/${requiredFuel}`);
+      await ensureFuel(bot, requiredFuel - fuelCount, "coal");
+    }
```

Child skill repair (`ensureFuel`):

```
-    const fuels = ["coal", "charcoal", "oak_log", "birch_log"];
+    // Prioritize efficient fuel sources
+    const fuels = preferredFuel
+      ? [preferredFuel, "coal", "charcoal"]
+      : ["coal", "charcoal"];
     for (const fuel of fuels) {
-      if (tryGetFuel(fuel)) return;
+      const obtained = await tryGetFuel(fuel, needed - currentFuel);
+      currentFuel += obtained;
+      if (currentFuel >= needed) break;
     }
+    // Explicit postcondition check
+    if (currentFuel < needed) {
+      throw new Error(`ensureFuel failed: ${currentFuel}/${needed}`);
+    }
```

This demonstrates PSN's ability to localize responsibility across skill boundaries and perform coordinated optimization over compositional skill hierarchies.

I Task sequences

Temporal Generalization Curriculum (Section 4.1). *Mine wood → Craft table → Craft wooden pickaxe → Craft stone pickaxe → Mine iron → Smelt iron → Craft iron pickaxe.*

Offline vs. Online Refactor Evaluation (Section 4.4). *Mine wood → Craft planks → Craft table → Craft wooden pickaxe → Mine cobblestone → Craft stone pickaxe → Mine iron → Smelt iron → Craft iron pickaxe.* All methods are evaluated on the identical task sequence without retraining.

J Comparison with ODYSSEY

Autonomy–engineering spectrum. PSN, ADAM, and ODYSSEY represent three positions along an autonomy–engineering spectrum. ODYSSEY represents the *maximum-engineering* endpoint: 183 hand-authored JavaScript skills achieve deterministic first-attempt success but require extensive per-domain human programming and cannot adapt beyond their pre-built repertoire. ADAM occupies an intermediate position, relying on 47 hand-authored skills while learning their causal input-output mappings via intervention-based verification. PSN represents the *maximum-autonomy* endpoint: without hand-authored task-specific skills, it generates all skill code from scratch, reaching diamond tool in 35±16 iterations (6/6 runs) with 30 autonomously learned skills—fewer iterations and higher reliability than ADAM (74±23) and Voyager* (99±36, 2/6). Table 8 provides a systematic comparison.

Dimension	PSN (Ours)	ODYSSEY	ADAM
Skill Source	LLM-generated & optimized (**30** learned)	**183** hand-authored JS skills	47 hand-authored JS skills + learned causal I/O mappings
Per-Task Skill Engineering	**None**	High (manual JS authoring per task domain)	High (47 JS skills + unlock tree + item/action dictionaries)
Skill Evolution	Yes (REFLECT + maturity gating)	No (static, fixed at authoring time)	No (causal graph is static once learned)
Code-Level Optimization	Yes (trace-based fault localization)	N/A (pre-built code)	N/A (no code generation)
Failure Recovery	Trace-based iterative repair	N/A (pre-built)	Memoryless restart (no cross-attempt accumulation)
Learning Capability	Generates, composes, optimizes code	None (retrieval of pre-built skills)	Shallow (discovers precondition-effect pairs only)
Tech Tree Completion	100% (6/6, both LLMs)	100% (pre-built skills, 120 ep.)	100% (3/3, GPT-5-mini; 0/3 Qwen3)
Diamond Tool	35±16 iter (6/6) / 49±18 (Qwen3)	Not reported	Not reported

Table 8: Systematic comparison of PSN, ODYSSEY (Liu et al., 2025), and ADAM (Yu & Lu, 2025). ODYSSEY and ADAM both rely on hand-authored task-specific skills; PSN learns all skills autonomously from scratch.

Learning trajectory: PSN vs. ODYSSEY (SR@k). To illustrate PSN's self-improvement capability, we evaluate task success rate at k attempts (SR@k) on the same 6-subgoal tech tree (Table 9).

Method	Skills	SR@1	SR@2	SR@3	SR@4
ODYSSEY	183 pre-built	100%	100%	100%	100%
PSN	**30 learned**	67%	83%	**100%**	**100%**

Table 9: Success rate at k attempts (SR@k) on the Minecraft tech tree. Both methods use GPT-5-mini.

ODYSSEY's pre-built skills achieve deterministic first-attempt success, as expected for hand-engineered, pre-tested code. Notably, when we load ODYSSEY's 183 skills into our evaluation framework with Qwen3-Coder for planning, the system achieves 100% SR@1 across all 18 subgoal evaluations (3 trials $\times$ 6 subgoals) *without any optimization or adaptation*. This confirms that ODYSSEY's improvements stem from the quality of its hand-authored skill library rather than from algorithmic learning.

By contrast, PSN starts with lower SR@1 (skills are learned from scratch, not pre-built) but converges to 100% at SR@3 through autonomous trace-based optimization (REFLECT). This SR@1$\rightarrow$SR@3 progression directly demonstrates PSN's self-improvement capability (the core contribution of this work) is achieved **without hand-authored task-specific skills**. PSN achieves this with only 30 autonomously learned skills, compared to ODYSSEY's 183 hand-authored skills, which is a 6$\times$ reduction in skill count while matching final reliability, highlighting the compositional efficiency of PSN's learned skill network.

K Credit assignment analysis

This appendix provides a quantitative analysis of PSN's credit assignment mechanism, examining how trace-based fault localization propagates repairs across skill compositions.

K.1 Per-episode optimization scope

We measure the *actual scope* of each optimization episode: when a task fails, how many skills in the invocation trace does PSN repair? An episode groups all skill optimizations triggered by a single task failure. A *multi-skill episode* indicates that REFLECT's recursive trace analysis propagated repairs across multiple skills in the composition.

Model	Episodes	Single-skill	Multi-skill	Avg depth
GPT-5-mini	204	88 (43.1%)	116 (56.9%)	2.7
Qwen3-Coder-Next	78	15 (19.2%)	63 (80.8%)	5.0

Table 10: Per-episode optimization scope across all experimental runs. Multi-skill episodes indicate that PSN's trace-based credit assignment propagated repairs across multiple skills in a composition. Qwen3-Coder-Next exhibits deeper credit propagation (avg 5.0 skills per multi-skill episode) than GPT-5-mini (2.7).

Both models exhibit substantial cross-skill credit assignment: 56.9% (GPT-5-mini) and 80.8% (Qwen3-Coder-Next) of optimization episodes involve coordinated repairs across multiple skills in a composition. Qwen3 shows deeper propagation (average 5.0 skills per multi-skill episode vs. 2.7 for GPT-5-mini), reflecting that weaker code generators produce compositions requiring more extensive repair chains. Crucially, this cross-skill optimization occurs through PSN's recursive trace-based fault localization, which is the architectural mechanism, rather than the LLM's explicit attribution judgment, drives effective multi-skill repair.

K.2 Error mode taxonomy

We categorize the failure modes identified by REFLECT across 1,059 optimization records from all experimental runs. Since a single optimization record may exhibit multiple failure modes, percentages sum to greater than 100%.

Error Mode	%	Typical Scope
Resource miscalculation	49.9%	Single-skill
API contract violation	27.8%	Single-skill
Precondition gap	24.3%	Single-skill
Defensive validation	23.2%	Single-skill
Placement / movement	21.0%	Single-skill
Return value contract	8.8%	Multi-skill
Cross-skill postcondition	6.1%	Multi-skill
Parameter / interface mismatch	2.2%	Multi-skill
LLM hallucination	0.3%	Single-skill

Table 11: Error mode distribution across 1,059 optimization records. Single-skill errors (top 5 rows) are resolved within individual skills; inter-skill contract violations (rows 6–8) trigger multi-skill repair episodes where REFLECT propagates fixes across the invocation trace. $N = 1{,}059$; percentages sum to >100% due to multi-label classification.

Resource miscalculation dominates (49.9%), consistent with Minecraft's deep material dependency chains where intermediate crafting steps consume resources that must be accounted for. Cross-boundary error modes, including return value contracts (8.8%), cross-skill postconditions (6.1%), and parameter mismatches (2.2%), collectively account for ∼17% of failures and typically trigger multi-skill repair episodes, confirming that REFLECT's recursive trace analysis identifies and propagates repairs for inter-skill issues.
Notably, LLM hallucination (generating references to non-existent functions or APIs) is extremely rare (0.3%, 3 cases across all runs). PSN's structured prompting and validation effectively alleviates the LLM's hallucination in response.

K.3 Structural credit assignment

The per-episode analysis reveals that PSN's credit assignment operates at the *system* level rather than the *judgment* level. When a task fails, REFLECT recursively analyzes the invocation trace top-down, identifying faulty skills at each level. Each skill is then optimized bottom-up. This recursive structure naturally propagates repairs across the skill graph without requiring the LLM to explicitly categorize faults as "caller" or "callee" issues—the compositional credit assignment emerges from the trace-based architecture itself. This is evidenced by the high multi-skill episode rates for both GPT-5-mini (56.9%) and Qwen3-Coder-Next (80.8%), demonstrating that the architectural mechanism generalizes across LLM backends of varying capability.

L Rollback and safety statistics

PSN employs a two-stage safety pipeline for structural changes to the skill network:

1. **Pre-application validation** (Stage 1): Before any code is modified, each refactor proposal undergoes syntax checking, type safety verification, and semantic preservation analysis. Proposals that fail these checks are rejected outright, i.e., no skill code is changed. The rejection rate for this stage is reported in the refactoring case distribution (Table 13: 43 of 139 proposals rejected, 31%).
2. **Post-application performance rollback** (Stage 2): Proposals that pass Stage 1 validation are applied to the skill network. The system then evaluates performance on a sliding window of 3 recent tasks involving the affected skills. If the success rate drops by more than 20%, the refactor is reverted using logged inverse operations. Table 12 reports these post-application rollback events.

Post-application rollbacks (Table 12) are rare in both model settings (3.1% and 6.7% of iterations, respectively), because Stage 1 validation already filters most problematic proposals. The weaker open-source model experiences approximately double the rollback rate, with both performance degradation rollbacks and refactor failure rollbacks increasing substantially. This suggests that PSN's safety mechanisms scale with model capability: weaker

models produce more optimization regressions that the degradation monitor catches, as well as more refactoring errors that the rollback procedure handles.

Model	Iterations	Rollbacks	Rate	Perf. Degrad.	Refactor Fail
GPT-5-mini	131	4	3.1%	1	3
Qwen3-Coder-Next	180	12	6.7%	6	6

Table 12: Rollback statistics across model settings. "Perf. Degrad." indicates rollbacks triggered by success rate drops exceeding the 20% threshold on a sliding window of recent tasks. "Refactor Fail" indicates rollbacks due to refactoring errors (syntax, type, or semantic validation failures detected post-application).

This reflects a deliberate safety-flexibility tradeoff. The five canonical refactoring patterns (Table 2) are chosen to be semantics-preserving, each admitting a deterministic rewrite rule. The refactoring framework is modular: new canonical patterns can be added by defining (1) a detection criterion, (2) a deterministic rewrite rule, and (3) a rollback procedure. Extending the pattern set to cover additional structural relationships is a natural direction for future work.

M Refactoring case distribution

Table 13 reports the aggregate distribution of refactoring cases across all experimental runs with refactoring enabled.

Refactor Type	Count	%
Behavioral coverage	60	43%
Extract common subskill	48	35%
Parametric coverage	18	13%
Merge siblings	11	8%
Duplication removal	2	1%
Total proposed	**139**	**100%**
Successfully applied	**96**	**69%**

Table 13: Distribution of canonical refactoring cases. 96 of 139 proposed refactors (69%) pass pre-application validation and are applied; the remaining 31% are rejected by syntax, type safety, or semantic preservation checks before any code is modified.

Behavioral coverage and common subskill extraction dominate (43% and 35%, respectively), consistent with the organic growth pattern of a skill network: as new skills are synthesized, they frequently reimplement functionality that already exists (behavioral coverage) or share common sub-operations (common subskill extraction). Duplication removal is rare (1%) because earlier refactoring patterns catch overlap before it reaches full duplication. The 31% rejection rate by pre-application validation confirms that the safety mechanism is binding, preventing potentially harmful structural changes from being applied to the skill network.

N API cost and token consumption

Table 14 compares per-task token consumption and total cost between PSN and Voyager.

Metric	PSN	Voyager*
Avg. tokens per task	$\sim$107K	$\sim$30K
Overhead ratio	3.5$\times$	
Total cost to Diamond Tool	$\sim$\$1.76 (3/3 runs)	∞ (0/3 runs)
Most expensive run	\$2.27	–

Table 14: Token consumption and cost comparison. PSN incurs $\sim$3.5$\times$ per-task overhead but reaches Diamond Tool in all runs; Voyager* never reaches Diamond Tool. Costs computed at GPT-5-mini pricing.

Comparing per-task token consumption directly is not a fair comparison, because PSN deliberately invests more tokens in early stages to learn, optimize, and compose a high-quality skill network. This early investment is precisely what enables efficient skill reuse

in later stages, unlocking harder achievements and sustaining longer survival. If we only look at per-task cost in the early tech tree, PSN appears more expensive; but PSN reaches Diamond Tool in all six runs while Voyager* does so in only 2/6 runs.
This investment amortizes clearly over the run. We observe CODEGEN overhead dropping from 8.3 calls per task in the first third of training to 5.0 in the final third (a 40% reduction), because later tasks increasingly compose existing skills rather than generating code from scratch. Once optimized, skills incur zero marginal LLM cost at execution time. For instance, `ensureCraftingTable` (reused 11 times in one run) executes without any LLM call after initial optimization, and the planning phase composes existing skills via graph traversal rather than LLM generation.

O Minecraft agent paradigm taxonomy

Table 15 categorizes Minecraft LLM agents by their skill representation, environment backend, action space, learning paradigm, and primary evaluation metric. This taxonomy clarifies why only code-generating, Mineflayer-based agents support controlled comparison with PSN.

Method	Venue	Skill Representation	Env Backend	Action Space	Primary Metric	Comparable?
Voyager	TMLR'24	LLM-generated JS code	Mineflayer	Code generation	Iterations to unlock	✓
ADAM	ICLR'25	47 hand-authored JS skills	Mineflayer	Predefined action selection	Iterations to unlock	✓
ODYSSEY	IJCAI'25	183 hand-authored JS skills	Mineflayer	Code generation	SR@k	✓
DEPS	NeurIPS'23	Predefined goal-cond. policies	MC-Controller	Goal-conditioned	Success rate	–
JARVIS-1	T-PAMI'24	Predefined MineRL actions	MineRL / MCP	Keypresses	Success rate	–
RL-GPT	NeurIPS'24	RL policy	MineDojo	Keyboard / mouse	Reward	–
Optimus-1	NeurIPS'24	VLM + Steve-1 controller	MineRL	Vision policy	Success rate	–
ROCKET-1	arXiv'24	Visual-motor policy	MineDojo	Segmentation actions	Reward	–

Table 15: Taxonomy of Minecraft LLM agent paradigms. "Comparable?" indicates whether direct experimental comparison with PSN is methodologically sound, requiring the same environment backend (Mineflayer), compatible action space (code generation), and comparable evaluation metrics. Methods below the mid-rule operate in fundamentally different paradigms.

Non-Minecraft programmatic skill methods. Several recent works share PSN's interest in programmatic skill/tool learning but operate in fundamentally different domains, precluding controlled comparison. ReGAL (Stengel-Eskin et al., 2024) performs *offline* refactoring over static code benchmarks (including TextCraft, a text-only symbolic crafting simulator with no 3D world or real-time physics). WALT (Prabhu et al., 2025) discovers browser automation tools via offline demonstration-generation-validation cycles in web DOM environments. ASI (Wang et al., 2025d) induces short browser action scripts (2–5 steps) online during web task solving, which is the closest to PSN's online paradigm, but targeting flat browser primitives rather than hierarchical, long-horizon skill compositions in a 3D embodied world. These methods share methodological concepts (programmatic abstractions, skill reuse) but differ in environment (text/web vs. embodied 3D), action complexity (browser macros vs. async multi-step game programs), and evaluation infrastructure, making direct numerical comparison infeasible.

P Learning dynamics decomposition

We empirically track the four components of the composite objective $J(\mathcal{N})$ introduced in Section 3 across training iterations. This section defines how each component is computed, then analyzes the resulting learning trajectories.

P.1 Component definitions

Each component is a scalar in $[0, 1]$ computed from the current network state:

- $\mathcal{R}_{\text{task}}$: **Rolling task success rate** over a sliding window of the 20 most recent iterations: $\mathcal{R}_{\text{task}} = \frac{1}{|W|} \sum_{i \in W} \mathbb{1}[\delta_i = 1]$, where W is the window and δ_i is the binary success indicator.
- $\mathcal{R}_{\text{reliab}}$: **Mean skill reliability** across the current skill library: $\mathcal{R}_{\text{reliab}} = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \max(V(s), 0)$. Since $V(s)$ starts at 0 for newly created skills and accumulates slowly with successful executions, this component grows on a slower timescale than $\mathcal{R}_{\text{task}}$.

- $\mathcal{R}_{\text{struct}}$: **Compositional reuse ratio**, defined as the fraction of skills that are invoked by more than one parent in the network (fan-in > 1). A higher value indicates that skills are shared across compositions rather than duplicated.
- $\mathcal{R}_{\text{cons}}$: **Refactoring consistency**, measured as the success rate of attempted refactoring operations: $\mathcal{R}_{\text{cons}} = n_{\text{success}}/n_{\text{total}}$ when $n_{\text{total}} > 0$, and 1.0 when no refactoring has been attempted (no structural changes implies no inconsistency). Both pre-application validation rejections and post-application rollbacks count as failures.

The composite objective is a weighted sum:

$$J(\mathcal{N}) = 0.4\,\mathcal{R}_{\text{task}} + 0.3\,\mathcal{R}_{\text{reliab}} + 0.2\,\mathcal{R}_{\text{struct}} + 0.1\,\mathcal{R}_{\text{cons}}. \tag{3}$$

The weights reflect a priority ordering: task performance is paramount ($w = 0.4$), followed by long-term skill reliability ($w = 0.3$), compositional structure ($w = 0.2$), and refactoring consistency ($w = 0.1$). These weights are hand-selected for interpretability rather than tuned; the qualitative conclusion (stable J under increasing task complexity)is robust across alternative weightings (uniform, inverted, and task-only; see below). $J(\mathcal{N})$ is never explicitly optimized; it serves as a diagnostic that aggregates the health of the skill network across multiple dimensions.

P.2 Empirical trajectories

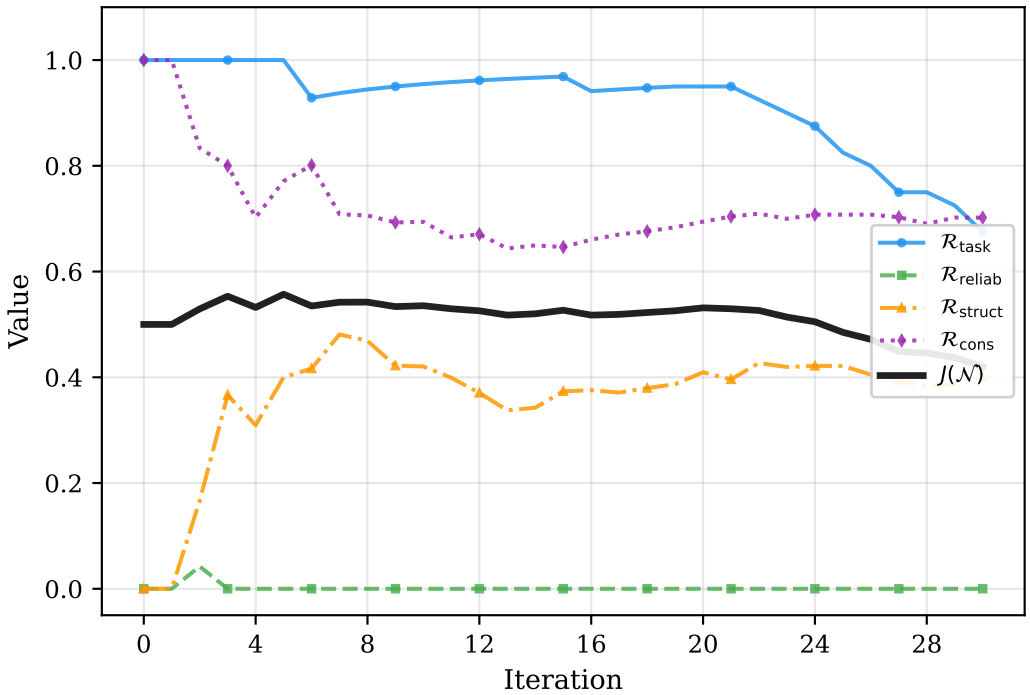


Figure 15: Decomposition of $J(\mathcal{N})$ over training iterations (GPT-5-mini, averaged over 2 runs with active refactoring). Each line shows an unweighted component; the black line shows the weighted composite $J(\mathcal{N})$. Despite tasks escalating from wood gathering to diamond tool crafting, J remains stable at ≈ 0.5: $\mathcal{R}_{\text{task}}$ stays high, $\mathcal{R}_{\text{struct}}$ grows with compositional reuse, and $\mathcal{R}_{\text{cons}}$ stabilizes as refactoring quality improves.

GPT-5-mini: stable learning under increasing complexity. The most striking feature is the stability of $J(\mathcal{N}) \approx 0.5$ throughout training (Figure 15). This stability is *not* because the tasks are easy: over the iterations, the agent progresses from simple resource gathering ("ensure 4 wood logs") through multi-step crafting (stone/iron pickaxe, furnace smelting) to deeply compositional objectives (diamond pickaxe, obsidian). The skill graph grows from 1 to ∼40 nodes with depth up to 6.

The stability arises from a compensatory dynamic among components. $\mathcal{R}_{\text{task}}$ remains near 1.0, indicating that PSN successfully solves increasingly difficult tasks without accumulating

failures. $\mathcal{R}_{\text{struct}}$ grows steadily from 0 to $\sim$0.4 as later skills invoke earlier ones, reflecting the network's increasing compositional reuse. The same trajectory on the weaker model (Qwen3) plateaus lower, at 0.15 ± 0.07, consistent with the flatter, less-reused graph reported in Section 4.6. $\mathcal{R}_{\text{cons}}$ starts high, dips during early refactoring when the network is still small and some proposals fail validation, then stabilizes around 0.7 as refactoring quality improves with a larger, more structured network. These three components collectively maintain J in a narrow band despite the escalating task difficulty.

This pattern is analogous to a well-regularized neural network whose training loss remains stable as it learns increasingly complex patterns: new representational capacity ($\mathcal{R}_{\text{struct}}$, via skill composition and reuse) is acquired at approximately the same rate that task complexity increases, preventing the objective from degrading.

$\mathcal{R}_{\text{reliab}}$ stays near zero because $V(s)$ accumulates on a slower timescale than task success: newly created skills start at $V(s) = 0$ and require many successful executions to build up Bayesian confidence.

Summary. The stability of $J(\mathcal{N})$ demonstrates that PSN's architectural mechanisms (compositional planning, trace-based optimization, maturity gating, and structural refactoring) enable the agent to absorb increasing task complexity into network structure without degrading task performance. This is a form of "loss stability" analogous to well-conditioned training in neural networks: new representational capacity ($\mathcal{R}_{\text{struct}}$) is acquired at approximately the same rate that task difficulty increases, while refactoring consistency ($\mathcal{R}_{\text{cons}}$) stabilizes as the network matures, collectively preventing the composite objective from degrading even as the agent tackles progressively harder compositions.

Weight sensitivity. To verify that the stability conclusion does not depend on the specific weight choice, we recompute J under four weighting schemes:

Weights (w_1, w_2, w_3, w_4)	J_{start}	J_{end}	Trend
$(0.4, 0.3, 0.2, 0.1)$	0.50	0.54	stable
$(0.25, 0.25, 0.25, 0.25)$	0.50	0.58	stable
$(0.1, 0.2, 0.3, 0.4)$	0.50	0.62	stable
$(1.0, 0.0, 0.0, 0.0)$	1.00	0.85	stable

Table 16: $J(\mathcal{N})$ under alternative weightings (GPT-5-mini). The stability of J is robust: under all tested weightings, J remains within a narrow range throughout training rather than degrading as task complexity increases.

Limitations

Our current implementation of PSN operates under constrained computational resources, resulting in an effectively batch-size-one online learning regime. This significantly limits the degree of parallelism in both skill execution and reflection-driven optimization, and prevents us from fully exploring large-scale network-level learning dynamics.

Moreover, the current reflection and refactoring process lacks a formal projection guarantee in the symbolic program space. While empirical improvements are consistently observed, the theoretical properties of symbolic projection, convergence, and optimality remain to be established.

Nevertheless, we believe these limitations are not fundamental to the PSN paradigm. With the continued scaling of large language models, increased computational budgets, and more efficient parallel execution infrastructures, future iterations of PSN are expected to support large-batch learning, stronger theoretical guarantees, and substantially improved optimization efficiency.