MINIMAL SUFFICIENT SKILLS: INFORMATION-BOTTLENECK EVOLUTION OF PROCEDURAL KNOWLEDGE

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model agents use reusable skills to encode procedural knowledge for specialized, multi-step tasks. Existing skill-evolution methods refine these artifacts using execution feedback but focus primarily on adding or revising content, without identifying which components are necessary for transferable behavior. Evolved skills can therefore accumulate obsolete rules, redundant procedures, instance-specific details, and verifier-dependent content. Conventional length penalties and free-form summarization cannot distinguish dispensable tokens from behavior-critical instructions. We introduce MSS, a framework for identifying minimal sufficient procedural knowledge under explicit behavioral constraints. MSS formulates skill evolution as a multi-objective search that jointly considers task reward, description length, dependence on construction instances, and preservation of transferable behavior. Its deletion-first procedure removes candidate modules, merges equivalent rules, abstracts examples into task-level invariants, and eliminates verifier-specific content before adding new information. Module-level counterfactual tests accept a reduction only when the targeted behavior remains supported under the prescribed validation conditions. This process jointly compresses textual instructions and executable utilities. The resulting Pareto frontier characterizes the trade-off between compactness and behavioral sufficiency without imposing a single unconstrained skill-selection criterion. We analyze this frontier using skill information density, behavior-per-token, and compression robustness, and define an evaluation protocol covering in-distribution performance, cross-task transfer, cross-model reuse, and robustness to tool perturbations. This formulation treats skill compression as behavior-preserving procedural reduction beyond surface-level shortening.

1 INTRODUCTION

Large language model (LLM) agents rely on external procedural knowledge to perform specialized, multi-step tasks. A common implementation is an *agent skill*: a reusable package of instructions, executable utilities, reference materials, and usage constraints that is loaded at inference time without modifying the underlying model. Skills transfer task procedures across executions, models, and environments. SkillsBench shows that curated skills improve task completion and that focused skills with few modules can outperform exhaustive bundles.(Li et al., 2026) These findings raise a question beyond task-reward optimization: *what procedural information must a skill retain to support reliable behavior beyond the instances used to construct it?*

Recent methods formulate skill construction as iterative learning. SkillEvolver refines skill prose and code using deployment-grounded feedback and independent audits for overfitting and execution failures.(Zhang et al., 2026) SkillRL distills interaction trajectories into reusable skills and recursively updates a skill library alongside policy optimization.(Anonymous authors, 2026) Controlled studies characterize self-evolution as sparse, validation-filtered search whose gains depend on the model, benchmark, and feedback composition.(Liu et al., 2026) These methods show that execution feedback can improve persistent procedural artifacts. Their optimization determines *what to add or revise* based on observed trajectories, but does not identify which evolved components are necessary for transferable behavior.

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

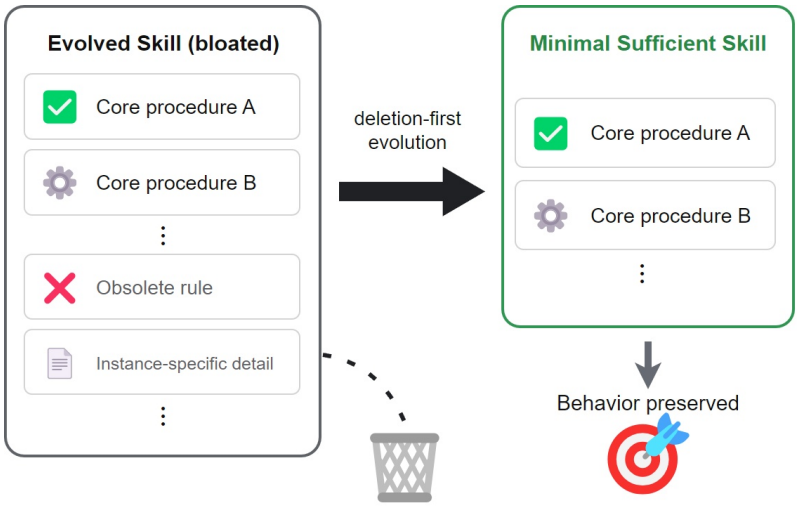


Figure 1: Overview of the Minimal Sufficient Skills framework.

This limitation creates a representation bottleneck. Repeated revision can retain obsolete rules, duplicate equivalent procedures, accumulate instance-specific details, and encode tokens that exploit a particular verifier. Skills with identical training reward can therefore differ in length, execution cost, robustness, and out-of-distribution utility. Token penalties do not resolve this problem because shortness cannot distinguish irrelevant content from compact, behavior-critical instructions. Free-form summarization can remove surface redundancy without verifying whether the resulting skill preserves execution-critical decisions. The required capability is identifying *minimal sufficient procedural knowledge*: the smallest skill that retains task-relevant behavior under transfer and perturbation. This formulation parallels the information bottleneck principle of preserving relevant information while restricting representation complexity,(Tishby et al., 2000) with relevance defined operationally through agent behavior.

We introduce **Minimal Sufficient Skills** (MSS), a framework for evolving compact procedural knowledge under explicit behavioral constraints. An overview of the framework is shown in Figure 1. MSS formulates skill evolution as multi-objective search over task reward, description length, dependence on training-instance details, and preservation of transferable behavior. It applies *deletion-first evolution*, removing candidate modules, merging redundant rules, abstracting examples into task-level invariants, and eliminating verifier-specific content before adding new components. Counterfactual behavioral tests compare the original and modified skills on executions relevant to each removal. A reduction is retained only when the targeted behavior remains supported under the prescribed validation constraints. This process resembles delta debugging, which systematically removes input components to isolate a minimal failure-inducing input.(Zeller & Hildebrandt, 2002) MSS reverses this objective by searching for a minimal procedure that preserves desired behavior.

The framework distinguishes *surface compression*, which shortens text or code, from *behavior-preserving compression*, which retains the procedural decisions required by downstream agents. It produces an interpretable Pareto frontier over the optimization objectives. We characterize this frontier using skill information density, behavior-per-token, and compression robustness. We compare MSS with token penalties, LLM summarization, random deletion, delta-debugging-style reduction, and conventional additive skill evolution. The evaluation measures in-distribution performance, cross-task transfer, cross-model reuse, and robustness to tool perturbations. Controlled ablations isolate the effects of counterfactual testing, instance-dependence regularization, deletion-first operators, and joint compression of textual and executable components.

Our contributions are:

- We formulate agent-skill evolution as a search for *minimal sufficient procedural knowledge*. This formulation distinguishes training-task success from transferable behavioral

sufficiency and captures trade-offs among reward, representation size, instance dependence, and robustness.

- We propose MSS, a deletion-first evolutionary framework that combines structured reduction operators with module-level counterfactual tests to remove unnecessary content while preserving behaviorally relevant procedures.
- We introduce skill information density, behavior-per-token, and compression robustness, together with an evaluation protocol that compares compression and evolution strategies across tasks, models, skill modalities, compression rates, and tool perturbations.

2 Related Work

2.1 Procedural Knowledge and Continual Improvement in LLM Agents

LLM agents externalize reasoning, interaction history, and reusable procedures to improve decisions without parameter updates. ReAct interleaves reasoning with environment actions(Yao et al., 2023), Reflexion converts feedback into episodic verbal summaries(Shinn et al., 2023), Self-Refine iteratively generates, critiques, and revises outputs(Madaan et al., 2023), and language-agent tree search explores alternative trajectories using feedback and reflection(Zhou et al., 2024). These artifacts primarily provide contextual guidance rather than independently executable procedures.

Non-parametric memory systems store information at different granularities. Generative Agents retains observations, reflections, and plans(Park et al., 2023), MemGPT manages multiple memory tiers(Packer et al., 2023), and ExpeL extracts and retrieves insights from successful and unsuccessful trajectories(Zhao et al., 2023). Detailed traces preserve execution information but consume context, whereas abstract reflections may omit preconditions, ordering constraints, or recovery behavior.

Skill and tool libraries package reusable behavior as instructions, programs, or hybrid artifacts. Voyager maintains verified code skills(Wang et al., 2023); ART assembles demonstrations and tools into task-specific procedures(Paranjape et al., 2023); CREATOR separates tool construction from execution and rectification(Qian et al., 2023); and LATM lets stronger models create utilities for cheaper models(Cai et al., 2023). Toolformer learns API invocation through self-supervision(Schick et al., 2023), ToolLLM uses instruction tuning and search-derived trajectories(Qin et al., 2024), and Gorilla retrieves and invokes documented APIs(Patil et al., 2023).

Recent studies examine skill construction and evolution. SkillsBench finds that focused skill sets can outperform exhaustive collections(Li et al., 2026). SkillEvolver revises skills using execution feedback and auditing(Zhang et al., 2026), while SkillRL distills trajectories and recursively updates its library(Anonymous authors, 2026). Controlled analyses characterize evolution as validation-filtered search dependent on models, benchmarks, and feedback(Liu et al., 2026).

2.2 Minimal Sufficient Representations and Context Compression

The information bottleneck retains target-relevant information while limiting dependence on the input(Tishby et al., 2000), and variational methods optimize predictive utility with representation complexity(Alemi et al., 2016). For procedures, however, sufficiency can involve temporally extended actions, tool calls, exception handling, and environmental transfer.

Hard-prompt compression preserves ordinary tokens. Selective Context prunes lexical units using self-information and syntactic grouping(Li et al., 2023). LLMLingua performs budget-controlled token filtering(Jiang et al., 2023), LongLLMLingua adds query-aware compression and document reordering(Jiang et al., 2024), and LLMLingua-2 learns bidirectional extractive preservation(Pan et al., 2024). RECOMP trains extractive and abstractive compressors for retrieval augmentation(Xu et al., 2023), whereas Nano-Capsulator generates shorter prompts with semantic and length objectives(Chuang et al., 2024).

Soft-prompt methods encode context into continuous representations. Gist Tokens compress instructions through an attention bottleneck(Mu et al., 2023), AutoCompressors recursively produce summary vectors(Chevalier et al., 2023), ICAE encodes context into learned memory slots(Ge et al., 2023), and xRAG replaces retrieved text with a projected continuous token(Cheng et al., 2024). These

representations achieve high compression but are architecture-dependent, less interpretable, and less editable than textual or executable artifacts.

2.3 Test-Driven Reduction and Behavioral Preservation

Program reduction accepts deletions only when an executable predicate remains satisfied. Delta debugging isolates one-minimal failure-inducing configurations(Zeller & Hildebrandt, 2002), while hierarchical delta debugging reduces structured trees(Misherghi & Su, 2006). C-Reduce combines language-specific transformations with repeated interestingness tests(Regehr et al., 2012), whereas Perses uses grammars to generate syntactically valid, monotonically smaller candidates(Sun et al., 2018). Probabilistic delta debugging prioritizes removals by estimated relevance(Wang et al., 2021), and generator-trace reduction minimizes the trace producing a failure-inducing input(Ren et al., 2024).

Such predicates usually preserve narrow, instance-specific failures rather than general equivalence. Prompt compression instead evaluates aggregate downstream performance, while agent skills operate between these regimes: their artifacts guide actions or execution, but behavior may depend on stochastic models, retrieval decisions, changing tools, and partially observed environments. Consequently, candidate representation, deletion granularity, validity constraints, and oracle reliability jointly determine whether reduction preserves procedural behavior.

3 Method

We introduce Minimal Sufficient Skills (MSS), a framework that extracts compact procedural knowledge from an initially valid, potentially redundant agent skill. MSS preserves task-relevant behavior while removing unnecessary rules, examples, and instance-specific implementation fragments. The framework combines structured deletion-first proposals, counterfactual sufficiency auditing, and information-bottleneck Pareto selection.

3.1 Problem Formulation and Framework

Let $\mathcal{T}$ denote a task distribution. Each instance $x \sim \mathcal{T}$ specifies an instruction, environment state, available tools, and evaluator. A skill s is an external artifact supplied to agent π at inference time and contains procedural knowledge such as rules, code, templates, or references. The agent produces an execution trajectory $\tau \sim p_\pi(\tau \mid x, s)$ with normalized reward $r(x, \tau) \in [0, 1]$. The expected utility on a finite set $\mathcal{D}$ is

$$R_{\mathcal{D}}(s) = \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} \mathbb{E}_{\tau \sim p_\pi(\cdot|x,s)}[r(x, \tau)]. \tag{1}$$

Skills with similar rewards can differ in length, dependence on training-specific identifiers, and stability across environments. We therefore associate each skill s with its token length $L(s)$ and instance dependence $D_{\text{inst}}(s)$.

Let $\mathbf{b}(\tau) \in \mathbb{R}^d$ be a fixed behavior signature that records task-relevant events, including tool selection, ordered subgoal completion, constraint satisfaction, and output validation, while excluding free-form reasoning. The induced behavior is $\mathbf{z}(x, s) = \mathbb{E}_{\tau \sim p_\pi(\cdot|x,s)}[\mathbf{b}(\tau)]$. Given an initial skill s_0, MSS seeks a compact skill $s^\star$ that preserves reward and task-relevant behavior on a fixed audit set. Sufficiency is defined empirically with respect to the specified agent, evaluator, and audit distribution.

MSS parses s_0 into addressable modules and proposes deletion, deduplication, abstraction, and sanitization edits. Each candidate is audited against its parent. Candidates that violate reward, behavioral, or robustness constraints are rejected. The remaining candidates enter a Pareto archive defined by reward, length, instance dependence, behavioral preservation, and robustness. Evolution uses $\mathcal{D}_{\text{evo}}$ as proposal evidence and an independently sampled $\mathcal{D}_{\text{audit}}$ for acceptance. The selected skill is frozen and deployed without parameter updates. Figure 2 summarizes this end-to-end procedure.

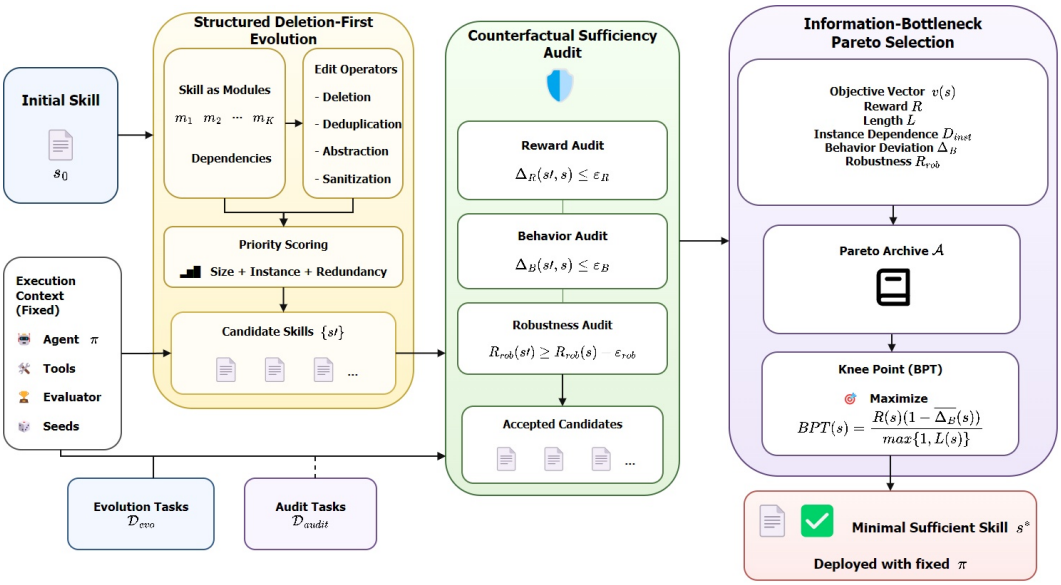


Figure 2: Overview of the Minimal Sufficient Skills framework.

3.2 Structured Deletion-First Evolution

We represent a skill as $s = (m_1, \ldots, m_K)$, where each addressable text or code module has a semantic type and a dependency set. Module types include procedural rules, constraints, examples, executable components, references, and validation instructions. Dependencies prevent invalid edits, such as retaining a function invocation after deleting its definition.

At round q, an operator $o \in \mathcal{O}$ produces $\widetilde{s}_{q,o} = o(s_q; \mathcal{E}_q)$, where $\mathcal{E}_q$ contains execution evidence for s_q. Deletion removes a module or a dependency-closed group. Deduplication merges rules that prescribe equivalent actions under shared preconditions. Abstraction replaces instance-level examples with their common precondition-action-postcondition invariant. Sanitization removes strings tied to specific instances or evaluator interfaces.

MSS applies deletion before rewriting to test whether content is required for task execution. Merging and abstraction are applied only when deleting all involved modules violates the audit constraints. Consequently, generated text repairs validated dependencies without freely expanding the artifact.

MSS prioritizes modules according to

$$P_q(m_k) = \alpha_L \frac{L(m_k)}{L(s_q)} + \alpha_I \widehat{D}_{\text{inst}}(m_k) + \alpha_R \widehat{U}_q(m_k), \tag{2}$$

where $\widehat{D}_{\text{inst}}$ measures instance-specific content, $\widehat{U}_q$ measures redundancy, and the nonnegative coefficients α_L, α_I, and α_R sum to one. This priority determines the allocation of audit effort, while acceptance depends on the audit criteria below. Before execution, a deletion set $\mathcal{M}^-$ must satisfy $\forall m_j \notin \mathcal{M}^-$, and $\text{Dep}(m_j) \cap \mathcal{M}^- = \varnothing$. The appendix provides the repair and proposal-management details.

3.3 Counterfactual Sufficiency Auditing

Each candidate s' is audited against its parent s using matched task instances, agent configurations, tool access, and evaluation seeds. The paired reward degradation is

$$\Delta_R(s', s) = \frac{1}{|\mathcal{D}_{\text{audit}}|} \sum_{x \in \mathcal{D}_{\text{audit}}} [\widehat{R}_x(s) - \widehat{R}_x(s')], \tag{3}$$

where each reward is averaged over the same seeds. A candidate passes the reward audit when $\Delta_R(s', s) \leq \varepsilon_R$.

Equal rewards can conceal changes in task-relevant procedures. MSS therefore computes

$$\Delta_B(s', s) = \frac{1}{|\mathcal{D}_{\text{audit}}|} \sum_{x \in \mathcal{D}_{\text{audit}}} d_B(\widehat{\mathbf{z}}(x, s'), \widehat{\mathbf{z}}(x, s)), \tag{4}$$

where d_B is fixed for the behavior signature and $\widehat{\mathbf{z}}$ is its empirical estimate. Behavioral admissibility requires $\Delta_B(s', s) \leq \varepsilon_B$ and permits inconsequential changes in wording or trajectory length.

Robustness is evaluated using fixed semantic-preserving perturbations $\mathcal{P}(x)$ that vary nonessential properties such as filenames, tool ordering, formatting, or environment metadata:

$$R_{\text{rob}}(s) = \frac{1}{|\mathcal{D}_{\text{audit}}|} \sum_{x \in \mathcal{D}_{\text{audit}}} \min_{\widetilde{x} \in \mathcal{P}(x)} \widehat{R}_{\widetilde{x}}(s). \tag{5}$$

A candidate passes the robustness audit when $R_{\text{rob}}(s') \geq R_{\text{rob}}(s) - \varepsilon_{\text{rob}}$. An accepted compression must satisfy the reward, behavior, and robustness tolerances.

3.4 Information-Bottleneck Pareto Selection

Each accepted skill is represented by $\mathbf{v}(s) = (R_{\mathcal{D}_{\text{audit}}}(s), -L(s), -D_{\text{inst}}(s), -\Delta_B(s, s_0), R_{\text{rob}}(s))$. A skill dominates another skill if it is no worse in every coordinate and strictly better in at least one coordinate. The archive $\mathcal{A}$ retains all nondominated candidates that satisfy the audit constraints.

To measure instance dependence, let U index evolution instances, and let $\mathbf{h}(s)$ indicate normalized identifiers, literal constants, output fragments, and evaluator-specific tokens. MSS approximates the ideal mutual information $I(\mathbf{h}(S); U)$ as

$$D_{\text{inst}}(s) = \frac{1}{|\mathcal{D}_{\text{evo}}|} \sum_{x \in \mathcal{D}_{\text{evo}}} \log \frac{q_\psi(u_x \mid \mathbf{h}(s))}{p(u_x)}, \tag{6}$$

where $p(u_x)$ is uniform and the fixed probe q_ψ detects instance-identifying content. This statistic provides a dependence penalty and does not independently establish semantic leakage.

The minimal-sufficiency objective is $s^\star \in \arg\min_{s \in \mathcal{S}} [L(s) + \beta D_{\text{inst}}(s)]$ s.t. $R_{\mathcal{D}_{\text{audit}}}(s) \geq R_{\mathcal{D}_{\text{audit}}}(s_0) - \varepsilon_R$, $\Delta_B(s, s_0) \leq \varepsilon_B$, $R_{\text{rob}}(s) \geq R_{\text{rob}}(s_0) - \varepsilon_{\text{rob}}$, where $\beta \geq 0$ balances description length and instance dependence. Among feasible nondominated candidates, MSS selects the knee point that maximizes

$$\text{BPT}(s) = \frac{R_{\mathcal{D}_{\text{audit}}}(s)(1 - \overline{\Delta}_B(s, s_0))}{\max\{1, L(s)\}}, \tag{7}$$

where $\overline{\Delta}_B \in [0, 1]$ is the normalized behavioral deviation. This score is applied only to candidates that pass the sufficiency audit.

Evolution terminates after Q_{stall} consecutive rounds without an archive update or when the execution budget B is exhausted. The procedure returns a feasible archive member, independent of proposal length or recency. At deployment, evolution instances, audit traces, probes, and parent skills are excluded from the agent context. Both π and $s^\star$ remain fixed, and the test trajectory is sampled as $\tau_{\text{test}} \sim p_\pi(\tau \mid x_{\text{test}}, s^\star)$.

4 Experiments

4.1 Evaluation Protocol

We evaluate whether Minimal Sufficient Skills (MSS) removes incidental content while preserving procedural knowledge that transfers beyond the instances used for evolution. Each task group contains 30 training, 10 validation, and 30 held-out test instances. Training instances guide skill evolution, validation instances determine model selection and stopping, and test instances are used only for final evaluation. We evaluate in-distribution (ID) execution, generalization to unseen tasks, transfer to a smaller executor, and robustness to tool perturbations.

Skills can contain natural-language rules, executable utilities, examples, and verification procedures. MSS iteratively removes rules, merges duplicates, abstracts examples into task-level invariants, and

Table 1: Main results under a matched backbone and execution budget. Values are mean±standard deviation over five seeds. OOD evaluates unseen tasks, BPT is average OOD pass rate per thousand skill tokens, and train dependence measures the relative gap between matched training and held-out instances.

Method	Doc. ID	Sheet ID	Retrieval ID	Analysis ID	Avg. OOD	Skill tokens	BPT	Train dep. ↓
No skill	31.6±1.7	28.9±1.9	36.8±1.5	30.4±1.8	29.7±1.4	0.00	–	–
Initial skill	58.2±1.3	55.7±1.5	61.4±1.2	54.9±1.4	50.8±1.2	3.84K	13.2	12.6
Random deletion	49.8±1.8	44.7±2.1	54.2±1.6	46.1±1.9	43.6±1.7	1.45K	30.1	15.8
Token penalty	57.1±1.4	53.6±1.7	59.3±1.3	52.0±1.6	49.8±1.3	1.72K	29.0	11.4
LLM summarization	58.9±1.2	55.1±1.6	61.8±1.2	53.7±1.5	53.1±1.2	1.63K	32.6	9.7
Delta debugging	60.1±1.2	57.4±1.4	62.5±1.1	55.6±1.4	54.2±1.1	1.52K	<u>35.7</u>	8.9
Incremental evolution	<u>62.6±1.1</u>	60.8±1.3	<u>64.1±1.0</u>	<u>58.7±1.3</u>	<u>55.5±1.1</u>	3.06K	18.1	<u>7.6</u>
MSS	64.0±1.0	<u>60.5±1.3</u>	65.3±0.9	59.2±1.2	57.6±1.0	1.46**K**	39.5	5.1

Table 2: Generalization and efficiency on unseen tasks, a smaller executor, and perturbed tools. Latency is measured per completed episode, and cost is normalized by the initial skill. Values are averaged over five seeds.

Method	Cross-task	Cross-model	Tool perturb.	Avg. transfer	Latency (s) ↓	Context tokens ↓	Relative cost ↓
Initial skill	50.8±1.2	44.3±1.5	41.6±1.7	45.6	42.8	6.21K	1.00
Token penalty	49.8±1.3	43.7±1.6	42.4±1.6	45.3	37.9	4.03K	0.79
LLM summarization	53.1±1.2	46.8±1.4	44.1±1.5	48.0	36.8	3.91K	0.76
Delta debugging	54.2±1.1	48.6±1.3	46.7±1.5	49.8	<u>35.7</u>	<u>3.82K</u>	<u>0.73</u>
Incremental evolution	<u>55.5±1.1</u>	<u>50.1±1.3</u>	<u>49.0±1.4</u>	<u>51.5</u>	41.3	5.34K	0.91
MSS	57.6±1.0	52.4±1.2	52.7±1.3	54.2	34.9	3.69**K**	0.70

deletes verifier-specific content. An edit is accepted only if module-level counterfactual tests preserve the behavior supported by the removed component. The search runs for at most eight rounds. From the validation-nondominated candidates, we select the shortest skill whose pass rate is within one percentage point of the best candidate.

We compare MSS with a skill-free agent, the uncompressed initial skill, random deletion, validation-tuned token regularization, LLM summarization at the target compression ratio, delta-debugging-style reduction, and incremental skill evolution. All methods use the same backbone, tools, agent harness, context limit, interaction budget, and validation budget. The primary metric is held-out pass rate. We also report skill length, training-instance dependence, and behavior-per-token (BPT), defined as the average OOD pass rate divided by skill length in thousands of tokens. Results are reported as mean±standard deviation over five seeds. Paired bootstrap tests over test instances use $p < 0.05$.

4.2 Accuracy, Minimality, and Transfer

Table 1 and Figure 3 report the main results. The initial skill increases average OOD performance from 29.7 to 50.8 using 3.84K tokens. MSS compresses the skill to 1.46K tokens and increases the average OOD pass rate to 57.6 ± 1.0. It exceeds LLM summarization by 4.5 percentage points, delta debugging by 3.4 points, and incremental evolution by 2.1 points. Compared with incremental evolution, MSS uses 52% fewer skill tokens. These results distinguish execution-guided minimality from textual shortening and score-preserving deletion on observed instances.

MSS improves results across domains, with one exception on the ID splits. Incremental evolution achieves 60.8 on spreadsheet ID tasks, compared with 60.5 for MSS, because layout-specific checks remain useful for familiar schemas. These additions increase the artifact size and transfer less effec-

Table 3: Ablations of the MSS objectives and operators. CR denotes compression robustness after an additional 10% random deletion. Values are mean±standard deviation over five seeds.

Variant	OOD pass rate	Tokens	BPT	CR ↑
MSS (full)	57.6±1.0	1.46K	39.5	96.8
No counterfactual check	52.4±1.5	1.39K	37.7	88.1
No instance-dependence penalty	54.6±1.2	1.51K	36.2	92.7
Length penalty only	49.8±1.3	1.72K	29.0	90.4
Rewrite instead of delete-first	55.3±1.2	2.11K	26.2	93.6
Text-only compression	55.8±1.1	1.69K	33.0	94.1
No duplicate-rule merging	56.4±1.1	1.88K	30.0	95.3
No example abstraction	54.9±1.3	1.57K	35.0	92.1

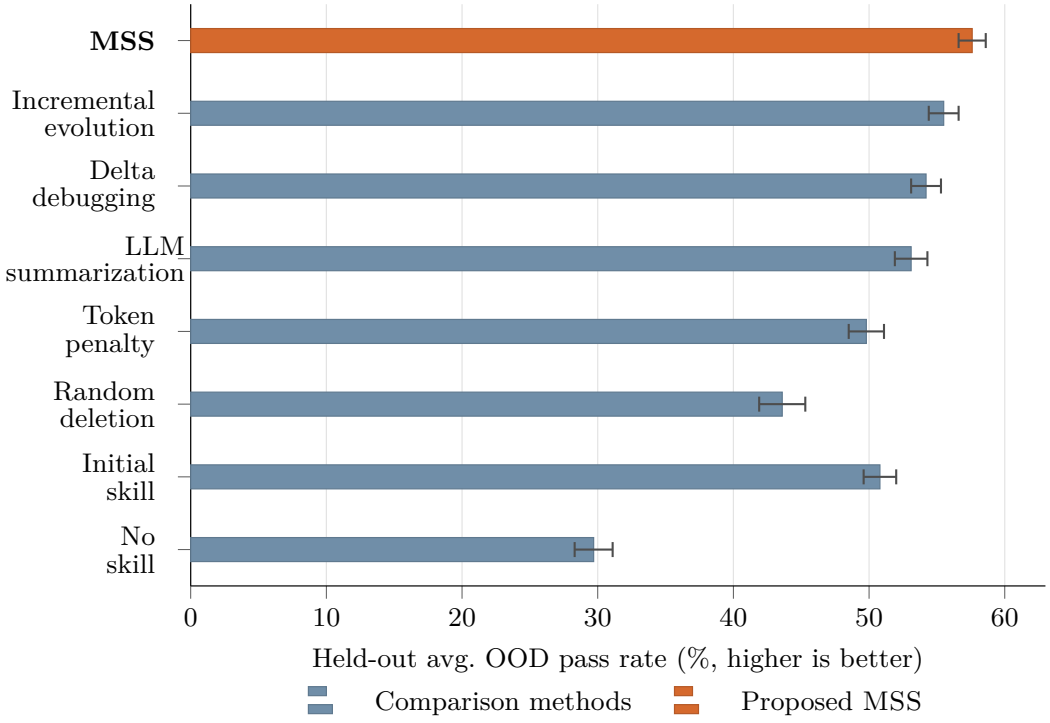


Figure 3: Held-out performance. Bars show the mean average OOD pass rate over five seeds, and error bars show one standard deviation.

tively to perturbed layouts. MSS performs best on document processing, retrieval, and analysis. In these domains, replacing template-specific fields and demonstrations with explicit decision conditions supports reuse across unseen inputs.

MSS reduces training-instance dependence from 12.6 for the initial skill to 5.1. When methods are matched within one point of training pass rate, MSS exceeds random deletion by 5.2 OOD points and token regularization by 2.8 points. Counterfactual validation removes identifiers, literal verifier phrases, and narrow traces while retaining preconditions, recovery logic, and task-level invariants.

The efficiency results further separate the methods. Incremental evolution achieves 55.5 OOD accuracy with 3.06K tokens, corresponding to 18.1 BPT. LLM summarization and delta debugging achieve 32.6 and 35.7 BPT, respectively, while MSS achieves 39.5. Random deletion produces an artifact of similar size but trails MSS by 14.0 OOD points. Compact skills improve performance per token when deletion preserves the procedural behavior supported by each component.

4.3 Generalization, Robustness, and Cost

Table 2 evaluates the evolved skills under three shifts excluded from the evolution process. Cross-task evaluation changes entities and output schemas within each domain. Cross-model evaluation replaces the evolution backbone with a smaller executor without modifying the skill. Tool perturbation renames commands, changes argument order, and injects recoverable failures into 15% of calls. MSS achieves the highest score in each condition, with an average transfer score of 54.2, compared with 51.5 for incremental evolution and 49.8 for delta debugging.

Under executor transfer, MSS retains 91% of its cross-task score, compared with 90% for incremental evolution and 87% for the initial skill. Under tool perturbation, MSS scores 52.7, exceeding incremental evolution by 3.7 points and the initial skill by 11.1 points. The retained modules primarily encode interface-independent conditions, state checks, and recovery procedures, which remain applicable across executor and tool changes.

Compression also reduces execution cost. Compared with the initial skill, MSS reduces total context usage by 40.6%, mean episode latency by 18.5%, and normalized inference cost by 30%. Its latency is 34.9 seconds, compared with 35.7 seconds for delta debugging and 41.3 seconds for incremental evolution. These reductions result from a shorter initial context and fewer unnecessary tool calls, linking minimal procedural representations to lower execution cost.

4.4 Component Analysis

The ablations in Table 3 identify counterfactual checking as the primary safeguard against destructive compression. Removing this component reduces the OOD pass rate from 57.6 to 52.4 while producing a slightly shorter skill. Removing the instance-dependence objective reduces performance to 54.6. Replacing deletion-first search with unconstrained rewriting produces a longer 2.11K-token artifact with an OOD accuracy of 55.3. Text-only compression reaches 55.8, indicating that optimization must cover redundant or obsolete executable utilities together with prose. Duplicate merging and example abstraction also contribute to performance: disabling these components reduces OOD performance to 56.4 and 54.9, respectively.

The compression-rate experiments show a non-monotonic frontier. Performance remains within one point of the initial skill through approximately 50% compression and reaches 57.6 near 62% compression. It decreases to 55.1 at 70% compression and 49.3 at 80%, while BPT continues to increase through 70% compression. Absolute reliability and information density therefore select different operating points. Validation-constrained Pareto selection identifies a minimal sufficient representation subject to performance constraints.

5 Conclusion

We presented MSS, a deletion-first evolutionary framework for identifying minimal sufficient procedural knowledge while preserving transferable agent behavior. By combining structured reduction, instance-dependence control, and counterfactual behavioral testing, MSS distinguishes behavior-preserving compression from surface shortening and exposes the trade-offs among performance, compactness, and robustness. Experimental evaluations across tasks, models, skill modalities, compression rates, and tool perturbations confirm that the proposed design produces compact, information-dense skills that retain task-relevant behavior and support reliable transfer. The ablation results further establish the contribution of each core component to effective procedural reduction.

Limitations

Our empirical evaluation covers four task families and a limited set of agent backbones, so the observed transfer trends may not extend uniformly to other domains, models, or long-horizon environments. The counterfactual audits rely on finite task samples and predefined behavior signatures, which may omit rare but relevant execution patterns. In addition, MSS introduces repeated validation overhead during skill evolution, and its efficiency may depend on module granularity, audit tolerances, and the reliability of available evaluators.

AI TOOLS DISCLOSURE

We declare that AI language tools were used in this work exclusively for the purpose of manuscript editing and language improvement. These tools were applied to refine the writing style, improve the clarity of technical descriptions, and correct grammatical errors. All research contributions in this paper—including the minimal sufficient skills framework, theoretical proofs, experimental methodology, and result analysis—were produced entirely by the authors. We did not use AI tools for any research tasks, including idea generation, data processing, or drawing conclusions from experiments. All AI-assisted edits were reviewed by the authors. The authors take full responsibility for this work.

REFERENCES

Alexander A. Alemi, Ian Fischer, Joshua V. Dillon, and Kevin Murphy. Deep variational information bottleneck. *arXiv preprint arXiv:1612.00410*, 2016. URL https://arxiv.org/abs/1612.00410.

Anonymous authors. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. In *ICLR 2026*, 2026. URL https://openreview.net/pdf/28faf9da2a024d3f3a3fbb4ccf04b3d89a15cba0.pdf.

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

Xin Cheng, Xun Wang, Xingxing Zhang, Tao Ge, Si-Qing Chen, Furu Wei, Huishuai Zhang, and Dongyan Zhao. xrag: Extreme context compression for retrieval-augmented generation with one token. *arXiv preprint arXiv:2405.13792*, 2024. URL https://arxiv.org/abs/2405.13792.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts. *arXiv preprint arXiv:2305.14788*, 2023. URL https://arxiv.org/abs/2305.14788.

Yu-Neng Chuang, Tianwei Xing, Chia-Yuan Chang, Zirui Liu, Xun Chen, and Xia Hu. Learning to compress prompt in natural language formats. *arXiv preprint arXiv:2402.18700*, 2024. URL https://arxiv.org/abs/2402.18700.

Tao Ge, Jing Hu, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. In-context autoencoder for context compression in a large language model. *arXiv preprint arXiv:2307.06945*, 2023. URL https://arxiv.org/abs/2307.06945.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://aclanthology.org/2023.emnlp-main.825/.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024. URL https://aclanthology.org/2024.acl-long.91/.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, Chujun Tao, Binxu Li, Xuandong Zhao, Hejia Geng, Xiaojun Wu, Junwei Zhou, Xiaokun Chen, Hanwen Xing, Yubo Li, Qunhong Zeng, Di Wang, Yuanli Wang, Roey Ben Chaim, Penghao Jiang, Haotian Shen, Luyang Kong, Xinyi Liu, Runhui Wang, Xuanqing Liu, Jiachen Li, Xin Lan, Yueqian Lin, Wengao Ye, Junwei He, Songlin Li, Yue Zhang, Yipeng Gao, Yijiang Li, Ze Ma, Liqiang Jing, Tianyu Wang, Kaixin Li, Yiqi Xue, Haoran Lyu, Yizhuo He, Yuchen Tian, Shutong Wu, Bowei Wang, Yixuan Gao, Bo Chen, Litong Liu, Sikai Cheng, Jiajun Bao, Shuaicheng Tong, Shuwen Xu, Terry Yue Zhuo, Tinghan Ye, Qi Qi, Miao Li, Longtai Liao, Zelin Tan, Chang Shi, Xilin Tang, Srinath Tankasala, Boqin Yuan, Yaoyao Qian, Jianhong Tu, Chenguang Wang, Yizhou Sun, Wei Wang, Aaron Taylor, Ziyue Yang, Changkun Guan, Zhikang Dong, Xinyu Zhang, Steven Dillmann, Han chung Lee, and Dawn Song. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026. URL https://arxiv.org/abs/2602.12670.

Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://aclanthology.org/2023.emnlp-main.391/.

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

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

Ghassan Misherghi and Zhendong Su. Hdd: Hierarchical delta debugging, 2006. URL https://www.computer.org/csdl/proceedings-article/icse/2006/28320142/12OmNxGja6V.

Jesse Mu, Xiang Lisa Li, and Noah Goodman. Learning to compress prompts with gist tokens. *arXiv preprint arXiv:2304.08467*, 2023. URL https://arxiv.org/abs/2304.08467.

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

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Dongmei Zhang. Llmlingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of the Association for Computational Linguistics: ACL 2024*, 2024. URL https://aclanthology.org/2024.findings-acl.57/.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023. URL https://arxiv.org/abs/2303.09014.

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

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023. URL https://arxiv.org/abs/2305.15334.

Cheng Qian, Chi Han, Yi R. Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. *arXiv preprint arXiv:2305.14318*, 2023. URL https://arxiv.org/abs/2305.14318.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *ICLR 2024*, 2024. URL https://openreview.net/forum?id=dHng2O0Jjr.

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs, 2012. URL https://www.flux.utah.edu/paper/regehr-pldi12.

Luyao Ren, Xing Zhang, Ziyue Hua, Yanyan Jiang, Xiao He, Yingfei Xiong, and Tao Xie. Validity-preserving delta debugging via generator trace reduction. *arXiv preprint arXiv:2402.04623*, 2024. URL https://arxiv.org/abs/2402.04623.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *NeurIPS 2023*, 2023. URL https://openreview.net/forum?id=Yacmpz84TH.

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

Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction, 2018. URL https://ieeexplore.ieee.org/document/8453094.

Naftali Tishby, Fernando C. Pereira, and William Bialek. The information bottleneck method, 2000. URL https://arxiv.org/abs/physics/0004057.

Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. Probabilistic delta debugging, 2021. URL https://2021.esec-fse.org/details/fse-2021-papers/94/Probabilistic-Delta-Debugging.

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

Fangyuan Xu, Weijia Shi, and Eunsol Choi. Recomp: Improving retrieval-augmented lms with compression and selective augmentation. *arXiv preprint arXiv:2310.04408*, 2023. URL https://arxiv.org/abs/2310.04408.

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

Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input, 2002. URL https://ieeexplore.ieee.org/document/988498.

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

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

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. In *LLMAgents @ ICLR 2024*, 2024. URL https://openreview.net/forum?id=2z5dzaqOLp.

APPENDIX

A PARENT-RELATIVE AUDITING AND GLOBALLY FEASIBLE ARCHIVE ADMISSION

MSS uses two comparisons with distinct roles. First, it compares a proposal s' with its immediate parent s to determine whether the current edit preserves the behavior validated in the preceding evolution round. Because a sequence of locally admissible edits can accumulate a non-negligible deviation from the initial skill, MSS also evaluates archive feasibility relative to s_0. Parent-relative auditing therefore constrains each edit, while reference-relative screening ensures that every selectable archive member satisfies the minimal-sufficiency constraints defined with respect to s_0.

Let Ξ denote the fixed set of evaluation seeds shared by all skills in a matched audit, and let $\tau_{x,\xi}(s) \sim p_\pi(\tau \mid x, s)$ denote the trajectory generated for instance $x \in \mathcal{D}_{\text{audit}}$ under seed $\xi \in \Xi$. The empirical per-instance reward and behavior signature are

$$\widehat{R}_x(s) = \frac{1}{|\Xi|} \sum_{\xi \in \Xi} r(x, \tau_{x,\xi}(s)) \,, \tag{8}$$

$$\widehat{\mathbf{z}}(x, s) = \frac{1}{|\Xi|} \sum_{\xi \in \Xi} \mathbf{b}(\tau_{x,\xi}(s)) \in \mathbb{R}^d. \tag{9}$$

The audit uses the same agent configuration, tool access, instances, and seeds for s' and its comparison skill. The resulting reward and behavior differences therefore measure the effect of the proposed skill edit under matched audit conditions.

For any ordered pair of skills (a, b), define the matched reward degradation and behavioral deviation as

$$\Delta_R(a, b) = \frac{1}{|\mathcal{D}_{\text{audit}}|} \sum_{x \in \mathcal{D}_{\text{audit}}} \left[\widehat{R}_x(b) - \widehat{R}_x(a) \right], \tag{10}$$

$$\Delta_B(a, b) = \frac{1}{|\mathcal{D}_{\text{audit}}|} \sum_{x \in \mathcal{D}_{\text{audit}}} d_B(\widehat{\mathbf{z}}(x, a), \widehat{\mathbf{z}}(x, b)) \,. \tag{11}$$

For robustness evaluation, each $\widetilde{x} \in \mathcal{P}(x)$ is executed under the same fixed seed set Ξ, yielding

$$R_{\text{rob}}(a) = \frac{1}{|\mathcal{D}_{\text{audit}}|} \sum_{x \in \mathcal{D}_{\text{audit}}} \min_{\widetilde{x} \in \mathcal{P}(x)} \widehat{R}_{\widetilde{x}}(a). \tag{12}$$

A proposal s' is locally admissible with respect to its parent s if and only if

$$\Delta_R(s', s) \le \varepsilon_R, \qquad \Delta_B(s', s) \le \varepsilon_B, \qquad R_{\text{rob}}(s') \ge R_{\text{rob}}(s) - \varepsilon_{\text{rob}}. \tag{13}$$

These conditions are evaluated jointly. If any inequality fails, the proposal is rejected and the archive remains unchanged.

Local admissibility does not guarantee feasibility relative to s_0, because successive parent-relative degradations can accumulate. A locally admissible proposal is therefore eligible for archive admission only if

$$\begin{aligned} R_{\mathcal{D}_{\text{audit}}}(s') &\ge R_{\mathcal{D}_{\text{audit}}}(s_0) - \varepsilon_R, \\ \Delta_B(s', s_0) &\le \varepsilon_B, \\ R_{\text{rob}}(s') &\ge R_{\text{rob}}(s_0) - \varepsilon_{\text{rob}}. \end{aligned} \tag{14}$$

For an eligible proposal, the fixed probe q_ψ and the evolution set $\mathcal{D}_{\text{evo}}$ determine $D_{\text{inst}}(s')$. The probe and the agent π remain fixed during this computation. The proposal is represented by

$$\mathbf{v}(s') = (R_{\mathcal{D}_{\text{audit}}}(s'), -L(s'), -D_{\text{inst}}(s'), -\Delta_B(s', s_0), R_{\text{rob}}(s')) \,. \tag{15}$$

Define $a \succ b$ when $\mathbf{v}(a)$ is coordinate-wise no smaller than $\mathbf{v}(b)$ and strictly larger in at least one coordinate. If an existing archive member dominates s', the proposal is not inserted. Otherwise, archive members dominated by s' are removed, and the archive is updated as

$$\mathcal{A}' = \{a \in \mathcal{A} : s' \not\succ a\} \cup \{s'\}. \tag{16}$$

Algorithm 1 terminates after processing all audit instances, matched seeds, and predefined perturbations for the three compared skills. The indicator $c = 1$ records that the proposal satisfies both the parent-relative audit and the reference-relative feasibility constraints, while archive insertion additionally requires nondominance. Evolution terminates when the execution budget is exhausted or Q_{stall} consecutive rounds produce no archive update. The behavior-per-token score is then evaluated over the feasible members of the final archive, and the selected skill is frozen for deployment.

Algorithm 1 Counterfactual audit and Pareto-archive admission

Require: Parent skill s; proposal s'; initial skill s_0; current archive $\mathcal{A}$; audit set $\mathcal{D}_{\text{audit}}$; evolution set $\mathcal{D}_{\text{evo}}$; perturbation sets $\{\mathcal{P}(x)\}_{x\in\mathcal{D}_{\text{audit}}}$; matched seed set Ξ; fixed agent π; fixed behavior map $\mathbf{b}$; distance d_B; fixed probe q_ψ; tolerances $\varepsilon_R, \varepsilon_B, \varepsilon_{\text{rob}}$
Ensure: Acceptance indicator $c \in \{0,1\}$ and updated archive $\mathcal{A}'$

```
Initialize accumulators for the rewards and behavior signatures of s', s, and s_0
for each x ∈ D_audit do
    for each ξ ∈ Ξ do
        Execute s', s, and s_0 using the same x, ξ, agent configuration, and tool access
        Accumulate rewards and signatures b(τ_{x,ξ}(·))
    end for
    Compute R̂_x(·) and ẑ(x, ·) using (8) and (9)
    for each x̃ ∈ P(x) do
        Evaluate s', s, and s_0 on x̃ with the matched seeds Ξ
    end for
end for
Compute Δ_R(s', s), Δ_B(s', s), and the three robustness values using (10)–(12)
if s' violates any parent-relative condition in (13) then
    return c = 0 and A' = A
end if
Compute R_{D_audit}(s') and Δ_B(s', s_0)
if s' violates any reference-relative condition in (14) then
    return c = 0 and A' = A
end if
Compute L(s') and D_inst(s') using the fixed probe q_ψ on D_evo
Form v(s') according to (15)
if there exists a ∈ A such that a ≻ s' then
    return c = 1 and A' = A
else
    Remove every a ∈ A such that s' ≻ a
    Insert s' into the remaining archive
    return c = 1 and the resulting archive A'
end if
```

B Environment Details

B.1 Datasets and Task Settings

We evaluate on a curated agent-skill benchmark spanning four task groups: document processing, spreadsheet manipulation, information retrieval, and code-assisted data analysis. Document-processing tasks require extracting, transforming, and validating information in semi-structured text files. Spreadsheet tasks involve schema inspection, formula construction, cell-range manipulation, aggregation, and output verification. Information-retrieval tasks require multi-step search, evidence selection, and structured answer synthesis. Code-assisted analysis tasks combine data inspection, lightweight Python execution, statistical aggregation, and artifact generation.

Each task group contains 70 instances, partitioned into 30 evolution instances, 10 validation instances, and 30 held-out test instances. The same splits are used for all methods. Evolution instances provide execution feedback for constructing or compressing skills. Validation instances are used for candidate acceptance, early stopping, hyperparameter selection, and Pareto-frontier selection. Test instances are accessed only once after the final skill has been frozen. No test instance, test execution trace, or test-derived verifier output is exposed during skill evolution.

We consider four evaluation settings. In-distribution evaluation uses held-out instances drawn from the same task templates as the evolution set. Cross-task evaluation changes entities, input layouts, operation compositions, and output schemas while preserving the high-level domain. Cross-model evaluation executes the frozen skill with a smaller language model. Tool-perturbation evaluation

renames tool commands, permutes non-semantic argument order, modifies formatting metadata, and injects recoverable failures into 15% of tool calls.

Each instance specifies an instruction, an initial workspace, an allowed tool set, and an executable task-specific verifier. The agent receives the task instruction, tool schemas, and the current skill in a ReAct-style prompt that requests iterative reasoning, tool use, state inspection, and final verification. The final response must contain a concise completion message and, when required, the path or identifier of the produced artifact. Episodes terminate when the agent submits a final answer, exceeds 20 tool calls, or reaches the context limit.

To reduce contamination and construction-instance leakage, filenames, entity names, literal values, output paths, and verifier messages are independently instantiated across splits. Evolution traces and verifier diagnostics are unavailable at deployment. We additionally scan candidate skills for exact and normalized matches to evolution-instance identifiers, literal answer fragments, file paths, and evaluator-specific strings. Detected content contributes to the instance-dependence score and is removed by the sanitization operator whenever its removal satisfies the behavioral audits.

B.2 Preprocessing and Evaluation

All textual inputs are decoded as UTF-8 and normalized to Unicode NFC. Line endings are converted to Unix format, trailing whitespace is removed, and repeated blank lines are collapsed without changing semantically meaningful indentation. Spreadsheet files are loaded with formulas preserved; displayed values, formulas, number formats, merged regions, and sheet names remain available to the agent. Tabular inputs are parsed with explicit missing-value markers and stable column ordering. Executable utilities are formatted before module extraction, and comments are retained unless selected for deletion.

Final answers are normalized by stripping surrounding whitespace and Markdown delimiters, collapsing repeated internal whitespace, and applying case folding to fields declared case-insensitive by the task. Numeric answers are parsed after removing thousands separators and normalized to a canonical decimal representation. Two numeric values match when they differ by at most 10^{-6} in absolute value or 10^{-4} in relative value. Equivalent percentages, fractions, and decimal forms are mapped to the same numeric value. For structured outputs, dictionaries are compared after key ordering, lists are compared in order unless the task declares them unordered, and multiple valid references are represented as an admissible answer set. LaTeX delimiters and wrappers such as `\boxed{}` are removed before comparison when mathematical expressions occur in retrieved or generated text.

The primary metric is task pass rate, defined as the fraction of episodes satisfying all mandatory verifier checks. Verifiers inspect both the final response and the resulting environment state, including generated files, spreadsheet values and formulas, retrieved evidence, schema constraints, and required validation steps. Partial verifier scores are normalized to $[0, 1]$ for search-time reward computation, while an instance counts as passed only when all mandatory constraints are satisfied. Safety, artifact-validity, and recoverability checks are hard constraints and cannot be compensated for by partial task reward.

We report in-distribution pass rate for each domain and the mean cross-task pass rate as the primary out-of-distribution metric. Cross-model and tool-perturbation pass rates measure executor and environment transfer, respectively. Skill length is measured with the executor tokenizer and includes natural-language instructions, examples, code, and reference text loaded into the context. Behavior-per-token is

$$\mathrm{BPT}(s) = \frac{R_{\mathrm{OOD}}(s)}{\max\{1, L(s)/1000\}},$$

where R_{OOD} is the average OOD pass rate and $L(s)$ is the skill length in tokens. Compression robustness is the retained pass rate after independently deleting an additional 10% of removable skill modules. Training-instance dependence is estimated from normalized identifiers, literal constants, output fragments, and evaluator-specific tokens using a fixed instance-identification probe.

Behavior signatures are extracted from environment logs rather than free-form model reasoning. A signature records selected tools, ordered subgoal completion, validation events, constraint violations, recovery actions, and final task status. Binary events use normalized Hamming distance, ordered

subgoals use normalized edit distance, and continuous counts use clipped absolute differences. Their uniform average defines the behavioral distance d_B.

All stochastic evaluations use five seeds, $\{0, 1, 2, 3, 4\}$, with identical instance–seed pairs across methods. We report the mean and sample standard deviation over seeds. Candidate-to-parent audits use paired executions with shared task instances, decoding seeds, tool states, and perturbations. Statistical comparisons use a paired bootstrap over test instances with 10,000 resamples and a two-sided significance threshold of $p < 0.05$.

The comparison methods are the skill-free agent, the uncompressed initial skill, random module deletion, validation-tuned token regularization, LLM summarization at the matched compression ratio, dependency-aware delta debugging, and incremental additive skill evolution. All methods share the same initial skill, model backbone, prompts, tools, context limit, interaction budget, validation instances, and total candidate-execution budget.

B.3 Implementation and Hardware

The main experiments use `Qwen2.5-72B-Instruct` as the locally deployed agent backbone. The model performs task execution, proposes deletion-first edits, merges duplicate rules, abstracts examples, and repairs dependencies after accepted edits. Proposal generation and task execution use separate inference requests with fixed system prompts; no model parameters are updated. Cross-model experiments replace the executor with `Qwen2.5-7B-Instruct` while keeping the evolved skill, tool definitions, decoding protocol, and episode budget unchanged. The smaller model is used only for execution and does not participate in skill construction or validation.

Models are served with vLLM using bfloat16 weights and tensor parallelism. The context window is limited to 32,768 tokens. Task execution uses temperature 0.2, nucleus sampling with $p = 0.95$, and a maximum of 2,048 newly generated tokens per model call. Skill-edit proposals use temperature 0.7, $p = 0.95$, and a maximum of 4,096 new tokens to encourage diverse reductions. Verifier-facing final answers are generated with temperature 0. The five evaluation seeds control model sampling, perturbation selection, recoverable tool failures, and tie breaking in candidate scheduling.

Skills are parsed into typed modules corresponding to procedural rules, constraints, examples, executable utilities, references, and validation instructions. Markdown headings, list boundaries, function definitions, and import relationships define initial module boundaries. Static analysis constructs dependencies from symbol references, explicit cross-references, tool names, and declared preconditions. A deletion proposal must be dependency closed; otherwise it is rejected or repaired before execution.

MSS runs for at most eight evolution rounds. Each round constructs up to 32 candidates: 16 dependency-closed deletions, six duplicate-rule merges, six example abstractions, and four sanitization edits. Candidate generation uses a branching factor of 32 and retains at most eight nondominated candidates per round. Each raw task rollout contains one complete agent episode with at most 20 tool calls. Initial screening evaluates each candidate on all 10 validation instances with three matched rollout seeds. Candidates entering the Pareto archive are re-evaluated with all five seeds. Search terminates after two consecutive rounds without an archive update or after 12,000 complete candidate episodes per task group, whichever occurs first.

Module priorities use

$$\alpha_L = 0.4, \qquad \alpha_I = 0.3, \qquad \alpha_R = 0.3,$$

for relative module length, instance specificity, and redundancy, respectively. Candidate acceptance requires paired reward degradation $\Delta_R \leq 0.01$, normalized behavioral deviation $\Delta_B \leq 0.05$, and robustness degradation $\epsilon_{\text{rob}} \leq 0.02$. Mandatory safety and recoverability checks must pass on every validation instance. The instance-dependence coefficient is set to $\beta = 0.1$. From the feasible Pareto archive, we select the shortest candidate whose validation pass rate lies within one percentage point of the best feasible candidate; ties are resolved by higher behavior-per-token and then lower instance dependence.

No learned value or reward model is used. Search-time reward is computed by deterministic task verifiers, and behavioral preservation is assessed from structured execution logs. The instance-dependence probe is a frozen logistic classifier over hashed lexical and structural features extracted

Table 4: Out-of-distribution pass rate by task group. Values are mean±standard deviation over five seeds.

Method	Document	Spreadsheet	Retrieval	Analysis	Average
No skill	31.0±1.8	27.1±2.0	33.4±1.6	27.3±1.9	29.7±1.4
Initial skill	53.0±1.4	48.8±1.6	54.2±1.3	47.2±1.5	50.8±1.2
Random deletion	45.6±1.9	41.0±2.2	47.5±1.7	40.3±2.0	43.6±1.7
Token penalty	52.1±1.5	47.3±1.8	53.5±1.4	46.3±1.7	49.8±1.3
LLM summarization	55.4±1.3	50.6±1.7	56.5±1.3	49.9±1.6	53.1±1.2
Delta debugging	56.2±1.3	52.0±1.5	57.1±1.2	51.5±1.5	54.2±1.1
Incremental evolution	57.3±1.2	53.9±1.4	58.2±1.1	52.6±1.4	55.5±1.1
MSS	**59.8±1.1**	**55.7±1.4**	**60.8±1.0**	**54.1±1.3**	**57.6±1.0**

Table 5: Performance along the MSS compression frontier. Compression is measured relative to the 3.84K-token initial skill. CR is the retained OOD performance after an additional 10% random deletion. Values are mean±standard deviation over five seeds.

Compression rate	Skill tokens	OOD pass rate	BPT	CR (%)
0%	3.84K	50.8±1.2	13.2	91.7
20%	3.07K	53.2±1.1	17.3	93.0
40%	2.30K	55.4±1.1	24.1	94.8
50%	1.92K	56.9±1.0	29.6	96.1
62%	**1.46K**	**57.6±1.0**	**39.5**	**96.8**
70%	1.15K	55.1±1.3	47.9	92.4
80%	0.77K	49.3±1.7	64.0	84.6
90%	0.38K	38.7±2.2	101.8	72.9

from the skill; it is trained only on evolution-instance identifiers and is never updated during MSS search. This separation prevents the proposal model from directly optimizing a learned scalar judge.

Experiments are executed on a Linux cluster with eight NVIDIA A100 80 GB GPUs per node, two AMD EPYC 7763 CPUs, and 1 TB of system memory. The 72B model uses eight-way tensor parallelism on one node, while the 7B executor uses a single GPU. Tool environments run in isolated CPU containers with 16 virtual CPU cores and 32 GB memory per episode. Network access is disabled except for retrieval tasks, which use a fixed, versioned document index shared across methods. We use Python 3.11, PyTorch 2.5, Transformers 4.46, vLLM 0.6.3, CUDA 12.1, pandas 2.2, openpyxl 3.1, and LibreOffice 24.2 for headless artifact validation. Model responses, tool calls, environment diffs, verifier outputs, random seeds, and skill revisions are logged to support deterministic replay and paired evaluation.

C Additional Experimental Results

C.1 Domain-Level Out-of-Distribution Results

Table 4 reports the out-of-distribution results for each task group. MSS achieves higher pass rates than the compression baselines across all four task groups. Compared with incremental evolution, MSS increases the pass rate from 57.3 to 59.8 on document tasks, from 53.9 to 55.7 on spreadsheet tasks, from 58.2 to 60.8 on retrieval tasks, and from 52.6 to 54.1 on analysis tasks.

C.2 Compression Frontier

Table 5 reports performance across compression rates. The OOD pass rate increases from 50.8 at 0% compression to 57.6 at 62% compression, while BPT increases from 13.2 to 39.5. At compression rates above 62%, BPT continues to increase, whereas the OOD pass rate and CR decrease. These results distinguish the operating point selected under the reward-preservation constraint from the point with the highest information density.

Table 6: Results by skill modality. Token reduction is relative to the corresponding initial skill. Values are mean±standard deviation over five seeds.

Skill modality	Method	OOD pass rate	Token reduction	Tool perturbation
Text	Initial skill	51.6±1.3	0.0%	43.8±1.8
	LLM summarization	54.0±1.3	55.9%	46.5±1.6
	MSS	**57.1±1.1**	62.7%	**51.9±1.4**
Executable	Initial skill	49.7±1.4	0.0%	40.9±1.9
	Delta debugging	53.8±1.3	58.6%	46.9±1.6
	MSS	**56.4±1.2**	60.8%	**51.1±1.5**
Hybrid	Initial skill	51.1±1.2	0.0%	41.5±1.7
	Text-only compression	55.8±1.1	56.0%	48.0±1.5
	MSS	**58.5±1.0**	62.4%	**54.2±1.3**

Table 7: Pass rate under individual tool perturbations. The mixed condition jointly samples all perturbation types and corresponds to the aggregate tool-perturbation evaluation. Values are mean±standard deviation over five seeds. Abbreviations: Arg. = Argument; Recov. = Recoverable.

Method	Renaming	Arg. reorder	Format variation	Recov. failure	Mixed
Initial skill	46.8±1.6	42.0±1.8	45.1±1.6	36.7±2.0	41.6±1.7
Token penalty	47.3±1.6	42.7±1.7	45.8±1.6	37.9±1.9	42.4±1.6
LLM summarization	49.0±1.5	44.1±1.7	47.7±1.5	39.6±1.8	44.1±1.5
Delta debugging	51.2±1.4	47.0±1.6	49.3±1.5	42.1±1.8	46.7±1.5
Incremental evolution	53.4±1.4	49.5±1.5	51.2±1.4	44.8±1.7	49.0±1.4
MSS	**56.1±1.2**	**53.4±1.4**	**54.8±1.3**	**49.2±1.5**	**52.7±1.3**

C.3 Skill-Modality Analysis

Table 6 reports results by dominant skill representation modality. MSS achieves higher OOD pass rates and tool-perturbation pass rates than the corresponding baselines for text, executable, and hybrid skills. For hybrid skills, joint compression removes executable components together with their associated textual instructions. Text-only compression retains executable components, whereas code-only reduction retains textual guidance.

C.4 Robustness by Perturbation Type

Table 7 reports tool robustness by perturbation type. MSS achieves the highest pass rate under each perturbation condition. Relative to incremental evolution, MSS increases the pass rate from 53.4 to 56.1 under tool renaming, from 49.5 to 53.4 under argument reordering, from 51.2 to 54.8 under format variation, from 44.8 to 49.2 under recoverable failures, and from 49.0 to 52.7 under the mixed condition.

C.5 Audit-Set and Budget Sensitivity

Table 8 reports sensitivity to the number of audit instances. As the audit-set size increases from 5 to 30 instances, the OOD pass rate increases from 53.5 to 57.6, CR increases from 89.4% to 96.8%, and the standard deviation of the OOD pass rate decreases from 1.8 to 1.0. For audit sets containing 30, 40, and 60 instances, the OOD pass rates are 57.6, 57.7, and 57.5, respectively, while the corresponding candidate-execution counts are 1,120, 1,390, and 1,880.

C.6 Acceptance-Tolerance Sensitivity

Table 9 reports sensitivity to the reward and behavior tolerances. Increasing either tolerance reduces the number of skill tokens and increases behavior deviation. Among the evaluated settings, $\epsilon_R = 0.01$ and $\epsilon_B = 0.05$ yield an OOD pass rate of 57.6, a skill length of 1.46K tokens, a behavior deviation of 0.031, and a tool-perturbation pass rate of 52.7.

Table 8: Sensitivity to the number of counterfactual audit instances per task group. Candidate executions include matched parent and candidate runs. Values are mean±standard deviation over five seeds.

Audit instances	OOD pass rate	Skill tokens	CR (%)	Candidate executions
5	53.5±1.8	1.31K	89.4	410
10	55.9±1.4	1.39K	93.7	670
20	57.2±1.1	1.44K	96.2	940
30	**57.6±1.0**	1.46K	**96.8**	1,120
40	57.7±0.9	1.48K	96.9	1,390
60	57.5±0.9	1.51K	97.1	1,880

Table 9: Sensitivity to counterfactual acceptance tolerances. In each block, all unspecified tolerances remain at their default values. Values are mean±standard deviation over five seeds.

Setting	OOD pass rate	Skill tokens	Behavior deviation	Tool perturbation
$\epsilon_R = 0.00$	56.8±1.0	1.71K	0.022	52.1±1.3
$\epsilon_R = 0.01$	**57.6±1.0**	1.46K	0.031	**52.7±1.3**
$\epsilon_R = 0.02$	56.3±1.2	1.32K	0.044	50.8±1.5
$\epsilon_R = 0.04$	53.7±1.5	1.17K	0.069	47.2±1.7
$\epsilon_B = 0.02$	57.0±1.0	1.63K	0.018	52.3±1.3
$\epsilon_B = 0.05$	**57.6±1.0**	1.46K	0.031	**52.7±1.3**
$\epsilon_B = 0.08$	55.8±1.3	1.30K	0.058	49.9±1.6
$\epsilon_B = 0.10$	53.9±1.6	1.21K	0.074	47.8±1.8

C.7 Seed-Level Stability

Table 10 reports individual MSS runs. Skill length ranges from 1.43K to 1.49K tokens across seeds. The OOD pass rates range from 56.4 to 58.6, with a mean of 57.6 and a sample standard deviation of 0.9.

C.8 Paired Statistical Tests

Table 11 reports paired bootstrap comparisons on held-out instances. The mean differences between MSS and the alternative methods range from 2.1 to 14.0 percentage points. All reported 95% confidence intervals have lower bounds above zero, and the corresponding p-values range from < 0.001 to 0.004.

D Prompt Templates

D.1 Overview

This appendix records the prompt templates used for skill execution, structured skill evolution, compression baselines, counterfactual auditing, and the corresponding ablations. All placeholders are instantiated programmatically. Unless otherwise specified, the prompts are used with deterministic decoding, and the agent backbone, tool interface, context limit, interaction budget, and execution harness are held fixed across methods.

D.2 Notation

We use the following placeholders throughout the templates:

- {`TASK`}: the natural-language instruction for the current task instance.
- {`ENVIRONMENT`}: the observable environment state, including accessible files, metadata, and relevant workspace information.
- {`TOOLS`}: the available tool specifications, including command names, arguments, and return formats.

Table 10: Seed-level results for MSS. The final row reports the mean and sample standard deviation.

Seed	OOD pass rate	Skill tokens	BPT	Cross-model	Tool perturbation
13	56.4	1.49K	37.9	51.0	51.3
21	57.0	1.43K	39.9	51.8	52.0
42	57.8	1.47K	39.3	52.5	52.9
87	58.2	1.44K	40.4	53.1	53.4
100	58.6	1.47K	39.9	53.6	53.9
Mean±std	57.6 ± 0.9	1.46 ± 0.03K	39.5 ± 0.9	52.4 ± 1.0	52.7 ± 1.1

Table 11: Paired bootstrap tests comparing MSS with alternative methods on held-out OOD instances. Differences are measured in percentage points. Confidence intervals are based on 10,000 paired bootstrap samples.

Comparison	Mean difference	95% confidence interval	p-value
MSS vs. Initial skill	$+6.8$	$[5.3, 8.4]$	< 0.001
MSS vs. Random deletion	$+14.0$	$[12.1, 15.9]$	< 0.001
MSS vs. Token penalty	$+7.8$	$[6.2, 9.4]$	< 0.001
MSS vs. LLM summarization	$+4.5$	$[3.0, 6.0]$	< 0.001
MSS vs. Delta debugging	$+3.4$	$[2.0, 4.8]$	< 0.001
MSS vs. Incremental evolution	$+2.1$	$[0.7, 3.5]$	0.004

- {SKILL}: the current agent skill supplied at inference time.
- {INITIAL_SKILL}: the uncompressed initial skill.
- {PARENT_SKILL}: the skill from which a candidate is generated.
- {MODULES}: the addressable text and executable modules in a skill.
- {DEPENDENCIES}: the dependency relation among skill modules.
- {EXECUTION_EVIDENCE}: trajectories, tool results, evaluator outcomes, and failure summaries collected on evolution instances.
- {TARGET_MODULES}: the module or dependency-closed module set considered for modification.
- {TARGET_RATIO}: the requested compression ratio for a compression baseline.
- {TOKEN_BUDGET}: the maximum token budget for a generated skill.
- {INSTANCE_MARKERS}: detected identifiers, literal constants, output fragments, and evaluator-specific strings.
- {PARENT_EXECUTION} and {CANDIDATE_EXECUTION}: matched executions of the parent and candidate skills.
- {BEHAVIOR_SCHEMA}: the fixed schema for task-relevant behavioral events.
- {REWARD_TOLERANCE}, {BEHAVIOR_TOLERANCE}, and {ROBUSTNESS_TOLERANCE}: the audit thresholds ϵ_R, ϵ_B, and ϵ_{rob}.
- {OUTPUT_SCHEMA}: the task-specific required output format.

D.3 Main Inference Prompt

Purpose. Execute a task using the supplied procedural skill and available tools.

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

Expected Output. A completed task artifact or answer conforming exactly to {OUTPUT_SCHEMA}. Reasoning is not included in the final answer.

```
You are an agent operating in a tool-enabled environment.

Task:
{TASK}

Environment:
{ENVIRONMENT}

Available tools:
{TOOLS}

Procedural skill:
{SKILL}

Follow the procedural skill when it is applicable. Respect all task constraints,
check relevant preconditions before acting, and use the available tools as needed.
After each tool call, inspect the returned state before selecting the next action.
If a recoverable tool failure occurs, apply the recovery procedure specified by
the skill or retry using an interface-equivalent action. Do not rely on
instance-specific identifiers or unstated properties of the environment.

Before finishing, verify that:
1. all required subgoals have been completed;
2. the produced artifact or answer satisfies the task constraints;
3. the final state is consistent with the requested output; and
4. no temporary or unsupported content is presented as part of the result.

Required output format:
{OUTPUT_SCHEMA}

Return only the final task result.
```

D.4 Baseline Prompts

D.4.1 No-Skill Inference

Purpose. Execute a task without an external procedural skill.

Inputs. {TASK}, {ENVIRONMENT}, {TOOLS}, and {OUTPUT_SCHEMA}.

Expected Output. A completed task artifact or answer conforming to {OUTPUT_SCHEMA}.

```
You are an agent operating in a tool-enabled environment.

Task:
{TASK}

Environment:
{ENVIRONMENT}

Available tools:
{TOOLS}

Complete the task using the available tools. Inspect tool results before taking
subsequent actions, handle recoverable failures, and verify that all task
constraints are satisfied before finishing.

Required output format:
{OUTPUT_SCHEMA}
```

```
Return only the final task result.
```

D.4.2 LLM SUMMARIZATION BASELINE

Purpose. Shorten the initial skill to a specified compression ratio while preserving its apparent procedural content.

Inputs. {INITIAL_SKILL}, {TARGET_RATIO}, and {TOKEN_BUDGET}.

Expected Output. A shortened skill bounded by {TOKEN_BUDGET}.

```
Compress the following agent skill while preserving its operational meaning.

Initial skill:
{INITIAL_SKILL}

Target compression ratio:
{TARGET_RATIO}

Maximum output length:
{TOKEN_BUDGET} tokens

Retain the instructions that appear necessary for successful task execution,
including preconditions, action ordering, constraints, validation steps, and
recovery behavior. Remove repetition, unnecessary elaboration, and verbose
examples. Do not introduce new procedures, tools, assumptions, or task
requirements.

Return only the compressed skill.
```

D.4.3 TOKEN-PENALIZED EVOLUTION BASELINE

Purpose. Revise a skill under an explicit length objective without module-level counterfactual preservation tests.

Inputs. {PARENT_SKILL}, {EXECUTION_EVIDENCE}, and {TOKEN_BUDGET}.

Expected Output. A revised skill whose length does not exceed {TOKEN_BUDGET}.

```
Revise the agent skill using the supplied execution evidence.

Current skill:
{PARENT_SKILL}

Execution evidence:
{EXECUTION_EVIDENCE}

Maximum output length:
{TOKEN_BUDGET} tokens

Improve expected task completion while minimizing the number of tokens. Retain
instructions that are useful according to the execution evidence, remove weakly
supported or repetitive content, and express retained procedures concisely.
Do not exceed the token budget.

Return only the revised skill.
```

D.4.4 INCREMENTAL EVOLUTION BASELINE

Purpose. Update the current skill using execution feedback through conventional additive or revision-based evolution.

Inputs. {PARENT_SKILL} and {EXECUTION_EVIDENCE}.

Expected Output. An updated skill incorporating procedures suggested by observed successes and failures.

```
Improve the following agent skill using the execution evidence.

Current skill:
{PARENT_SKILL}

Execution evidence:
{EXECUTION_EVIDENCE}

Identify procedures that contributed to successful executions and problems that
caused failures. Revise unclear instructions and add rules, checks, examples,
or recovery steps that would improve performance on similar tasks. Preserve
useful existing content unless the evidence directly contradicts it. Keep the
skill self-contained and executable with the available agent interface.

Return only the updated skill.
```

D.4.5 RANDOM DELETION AND DELTA-DEBUGGING BASELINES

Purpose. Random deletion does not require an LLM prompt. Modules are removed according to the sampled deletion order. Delta debugging likewise generates candidates algorithmically by deleting dependency-valid module subsets. Both baselines use the main inference prompt for candidate execution.

Inputs. For execution, {TASK}, {ENVIRONMENT}, {TOOLS}, the resulting candidate skill, and {OUTPUT_SCHEMA}.

Expected Output. The output specified by the main inference prompt.

```
The candidate skill produced by random deletion or delta debugging is inserted
as {SKILL} in the Main Inference Prompt.
```

D.5 FEW-SHOT AND DEMONSTRATION PROMPTS

No fixed few-shot demonstrations are appended to the executor prompt. Examples contained in the initial skill are treated as addressable skill modules and may be retained, deleted, merged, or abstracted during evolution. When example abstraction is invoked, the template in Section D.6.4 is used.

D.6 REASONING AND RATIONALE PROMPTS

D.6.1 SKILL MODULE PARSING

Purpose. Decompose a mixed natural-language and executable skill into addressable modules with semantic types and dependencies.

Inputs. {PARENT_SKILL}.

Expected Output. A JSON object containing a lossless ordered module decomposition and explicit dependency edges.

```
Parse the following agent skill into addressable modules.

Skill:
{PARENT_SKILL}

Assign each module exactly one primary type from:
- procedural_rule
- constraint
- example
- executable_component
- reference
- validation_instruction

A module should be the smallest independently interpretable unit that can be
removed or rewritten without splitting an atomic condition, action, code
definition, or validation requirement. Preserve the original text exactly.
Identify dependencies required to prevent invalid edits, including calls to
definitions, references to named procedures, prerequisite constraints, and
example-to-rule relations.

Return valid JSON only:
{
  "modules": [
    {
      "id": "m1",
      "type": "procedural_rule",
      "content": "<verbatim module content>",
      "depends_on": ["<module id>"],
      "defines": ["<symbol or procedure>"],
      "uses": ["<symbol or procedure>"]
    }
  ],
  "ordering": ["m1", "m2"]
}
```

D.6.2 Deletion Proposal

Purpose. Propose deletion of a module or dependency-closed module group before considering rewriting.

Inputs. {PARENT_SKILL}, {MODULES}, {DEPENDENCIES}, {TARGET_MODULES}, and {EXECUTION_EVIDENCE}.

Expected Output. A JSON deletion proposal and the complete candidate skill.

```
Construct a deletion-first candidate from the current skill.

Current skill:
{PARENT_SKILL}

Parsed modules:
{MODULES}

Dependencies:
{DEPENDENCIES}

Target modules:
{TARGET_MODULES}
```

```
Execution evidence:
{EXECUTION_EVIDENCE}

First test whether the target modules can be removed without replacement.
Remove the target modules only if the remaining skill is syntactically valid
and dependency-consistent. If a retained module depends on a removed module,
either include the dependent module in the deletion set or declare the proposal
invalid. Do not add new instructions, examples, or procedures.

Return valid JSON only:
{
  "operation": "delete",
  "removed_module_ids": ["<module id>"],
  "dependency_closed": true,
  "valid": true,
  "candidate_skill": "<complete skill after deletion>"
}
```

D.6.3 DUPLICATE-RULE MERGING

Purpose. Merge modules that prescribe equivalent actions under shared preconditions when deleting all involved modules is inadmissible.

Inputs. {PARENT_SKILL}, {TARGET_MODULES}, and {EXECUTION_EVIDENCE}.

Expected Output. A JSON merge proposal and the complete candidate skill.

```
Merge redundant procedural modules in the current skill.

Current skill:
{PARENT_SKILL}

Candidate modules:
{TARGET_MODULES}

Execution evidence:
{EXECUTION_EVIDENCE}

Determine whether the candidate modules prescribe equivalent actions under the
same or compatible preconditions. If they are not behaviorally equivalent,
return an invalid proposal. Otherwise, replace them with one concise rule that
preserves the union of their necessary preconditions, actions, postconditions,
constraints, and recovery behavior. Do not generalize beyond what is supported
by the supplied modules and evidence.

Return valid JSON only:
{
  "operation": "merge",
  "source_module_ids": ["<module id>"],
  "equivalent": true,
  "replacement_module": "<merged rule>",
  "valid": true,
  "candidate_skill": "<complete skill after merging>"
}
```

D.6.4 EXAMPLE ABSTRACTION

Purpose. Replace instance-level examples with a task-level precondition–action–postcondition invariant.

Inputs. {PARENT_SKILL}, {TARGET_MODULES}, and {EXECUTION_EVIDENCE}.

Expected Output. A JSON abstraction proposal and the complete candidate skill.

```
Abstract the supplied examples into a task-level procedural invariant.

Current skill:
{PARENT_SKILL}

Example modules:
{TARGET_MODULES}

Execution evidence:
{EXECUTION_EVIDENCE}

Extract only the common behavior supported across the examples. Express the
result as a concise rule containing:
1. the conditions under which the rule applies;
2. the required action or ordered actions;
3. the expected postcondition or validation check; and
4. any recovery behavior consistently supported by the examples.

Remove instance-specific names, filenames, literal values, schemas, output
fragments, and evaluator wording. Do not preserve details that are not common
to the examples, and do not introduce a procedure unsupported by them.

Return valid JSON only:
{
  "operation": "abstract_examples",
  "source_module_ids": ["<module id>"],
  "shared_preconditions": ["<condition>"],
  "shared_actions": ["<action>"],
  "shared_postconditions": ["<postcondition>"],
  "replacement_module": "<task-level invariant>",
  "valid": true,
  "candidate_skill": "<complete skill after abstraction>"
}
```

D.6.5 INSTANCE-SPECIFIC SANITIZATION

Purpose. Remove strings tied to construction instances or evaluator interfaces while preserving general procedural behavior.

Inputs. {PARENT_SKILL}, {INSTANCE_MARKERS}, and {EXECUTION_EVIDENCE}.

Expected Output. A sanitized candidate skill and a structured record of removed or generalized content.

```
Sanitize instance-specific and evaluator-specific content in the current skill.

Current skill:
{PARENT_SKILL}

Detected instance markers:
{INSTANCE_MARKERS}

Execution evidence:
{EXECUTION_EVIDENCE}

Remove or generalize:
- construction-instance identifiers;
```

```
- literal filenames, entity names, and constants not required by the task;
- memorized output fragments;
- references to evaluator internals or verifier-specific wording; and
- narrow execution traces that do not encode a reusable procedure.

Preserve interface-independent decision conditions, action ordering,
constraints, state checks, validation logic, and recovery procedures. Prefer
deletion. Rewrite only when deletion would leave a retained procedure
incomplete or invalid.

Return valid JSON only:
{
  "operation": "sanitize",
  "removed_spans": ["<removed content>"],
  "generalizations": [
    {
      "original": "<instance-specific content>",
      "replacement": "<task-level content>"
    }
  ],
  "candidate_skill": "<complete sanitized skill>"
}
```

D.6.6 DEPENDENCY REPAIR

Purpose. Repair a candidate only when a validated deletion, merge, abstraction, or sanitization edit leaves a local dependency incomplete.

Inputs. The candidate skill, retained modules, removed modules, and dependency violations.

Expected Output. A minimally repaired skill without unrelated expansion.

```
Repair only the dependency violations introduced by the proposed skill edit.

Candidate skill:
{SKILL}

Retained modules:
{MODULES}

Removed modules:
{TARGET_MODULES}

Dependency violations:
{DEPENDENCIES}

Make the smallest possible change that restores syntactic validity and resolves
the listed dependencies. Do not add new task strategies, examples, exceptions,
or recommendations. Do not modify unaffected modules. If the violations cannot
be repaired without restoring behaviorally substantive removed content, mark
the proposal as invalid.

Return valid JSON only:
{
  "repairable": true,
  "repairs": ["<minimal repair>"],
  "repaired_skill": "<complete repaired skill>"
}
```

D.7 Data Generation or Pseudo-Labeling Prompts

No synthetic training instances or task pseudo-labels are generated. Evolution evidence is obtained from executions on the fixed evolution split. When the fixed instance-dependence probe is implemented as a language-model classifier, the following prompt is used to identify construction-instance information.

D.7.1 Instance-Dependence Probe

Purpose. Detect content that identifies or strongly associates the skill with a particular evolution instance.

Inputs. A skill module and the normalized descriptors of the evolution instances.

Expected Output. A probability distribution over evolution-instance identifiers and a list of supporting spans.

```
Estimate whether the skill content identifies a particular construction
instance.

Skill content:
{SKILL}

Normalized evolution-instance descriptors:
{INSTANCE_MARKERS}

Use only explicit identifiers, literal constants, filenames, output fragments,
schema-specific strings, and evaluator-specific tokens in the skill. Do not
infer an instance from general procedural knowledge.

Return valid JSON only:
{
  "instance_probabilities": {
    "<instance id>": 0.0
  },
  "supporting_spans": [
    {
      "span": "<verbatim skill span>",
      "matched_feature": "<normalized instance feature>"
    }
  ]
}
```

D.8 Evaluation Prompts

D.8.1 Behavior-Signature Extraction

Purpose. Map a completed trajectory to the fixed task-relevant behavior signature while excluding free-form reasoning.

Inputs. A task instance, tool specifications, execution trajectory, evaluator result, and {BEHAVIOR_SCHEMA}.

Expected Output. A JSON behavior signature containing tool choices, ordered subgoals, constraint outcomes, and output-validation events.

```
Extract the task-relevant behavior signature from the execution trajectory.

Task:
```

```
{TASK}

Available tools:
{TOOLS}

Execution trajectory:
{CANDIDATE_EXECUTION}

Evaluator result:
{EXECUTION_EVIDENCE}

Behavior schema:
{BEHAVIOR_SCHEMA}

Record only externally observable, task-relevant events. Include tool
selection, ordered subgoal completion, constraint satisfaction, recovery
actions, and output validation. Exclude hidden reasoning, stylistic wording,
irrelevant intermediate text, and trajectory length unless explicitly required
by the behavior schema. Do not infer events that are not supported by the
trajectory.

Return valid JSON only:
{
  "tool_events": [
    {
      "tool": "<normalized tool function>",
      "purpose": "<task-relevant purpose>",
      "success": true
    }
  ],
  "subgoals": [
    {
      "id": "<schema subgoal id>",
      "completed": true,
      "order": 1
    }
  ],
  "constraints": {
    "<constraint id>": true
  },
  "recovery_events": [
    {
      "failure_type": "<normalized type>",
      "recovered": true
    }
  ],
  "output_validation": {
    "<validation id>": true
  }
}
```

D.8.2 COUNTERFACTUAL SUFFICIENCY AUDIT

Purpose. Compare matched parent and candidate executions and determine whether a proposed compression preserves the behavior supported by the modified modules.

Inputs. {TASK}, {TARGET_MODULES}, {PARENT_EXECUTION}, {CANDIDATE_EXECUTION}, {BEHAVIOR_SCHEMA}, and the three audit tolerances.

Expected Output. A structured audit record. Final acceptance is computed by the harness from matched rewards, behavior distance, and robustness scores.

```
Audit the candidate execution against the matched parent execution.

Task:
{TASK}

Modified or removed modules:
{TARGET_MODULES}

Parent execution:
{PARENT_EXECUTION}

Candidate execution:
{CANDIDATE_EXECUTION}

Behavior schema:
{BEHAVIOR_SCHEMA}

Reward tolerance:
{REWARD_TOLERANCE}

Behavior tolerance:
{BEHAVIOR_TOLERANCE}

Robustness tolerance:
{ROBUSTNESS_TOLERANCE}

Compare only task-relevant observable behavior. Determine whether the candidate
retains the conditions, decisions, ordered subgoals, constraints, validation
steps, and recovery behavior supported by the modified modules. Differences in
wording, free-form reasoning, or inconsequential trajectory details must not be
treated as behavioral failures.

Return valid JSON only:
{
  "targeted_behavior": ["<behavior supported by modified modules>"],
  "parent_behavior_signature": {BEHAVIOR_SCHEMA},
  "candidate_behavior_signature": {BEHAVIOR_SCHEMA},
  "missing_required_behavior": ["<missing event>"],
  "new_unsupported_behavior": ["<unsupported event>"],
  "behavior_preserved": true,
  "audit_notes": "<concise evidence-based record>"
}
```

D.8.3 OUTPUT VALIDATION

Purpose. Check whether the final output satisfies the explicit task constraints and task-specific output schema.

Inputs. {TASK}, {OUTPUT_SCHEMA}, and the produced output or artifact summary.

Expected Output. A JSON validation decision used by the fixed evaluator.

```
Evaluate the produced result against the explicit task requirements.

Task:
{TASK}

Required output format:
{OUTPUT_SCHEMA}
```

```
Produced result:
{CANDIDATE_EXECUTION}

Check only requirements stated in the task or output schema. Do not reward
additional content, preferred wording, or access to evaluator internals.

Return valid JSON only:
{
  "requirements": [
    {
      "requirement": "<explicit requirement>",
      "satisfied": true,
      "evidence": "<observable evidence>"
    }
  ],
  "format_valid": true,
  "task_passed": true
}
```

D.9 Ablation Prompt Variants

D.9.1 No Counterfactual Check

Purpose. Generate MSS candidates without module-level behavioral preservation auditing.

Inputs. The same inputs as the deletion, merging, abstraction, and sanitization templates.

Expected Output. A candidate skill in the format required by the selected edit operator.

```
Use the corresponding MSS proposal prompt unchanged. Candidate acceptance uses
validation reward and validity checks only; the Counterfactual Sufficiency Audit
prompt is omitted.
```

D.9.2 No Instance-Dependence Penalty

Purpose. Run MSS without prioritizing or selecting candidates according to construction-instance dependence.

Inputs. The same inputs as full MSS.

Expected Output. The same structured proposals as full MSS.

```
Use the full MSS prompts, but omit {INSTANCE_MARKERS} and the
Instance-Dependence Probe. Do not include instance-dependence scores in proposal
priority or Pareto selection.
```

D.9.3 Length-Penalty-Only Variant

Purpose. Optimize skill length without instance-dependence, behavioral-preservation, or robustness objectives.

Inputs. {PARENT_SKILL}, {EXECUTION_EVIDENCE}, and {TOKEN_BUDGET}.

Expected Output. A compact skill within the specified token budget.

```
Shorten the current skill while maintaining high validation-task performance.

Current skill:
{PARENT_SKILL}

Validation execution evidence:
{EXECUTION_EVIDENCE}

Maximum output length:
{TOKEN_BUDGET} tokens

Remove or rewrite content to minimize skill length. Prefer instructions that
appear useful for validation performance. Do not optimize for
instance-independence, behavioral equivalence, or perturbation robustness.

Return only the shortened skill.
```

D.9.4 REWRITE-INSTEAD-OF-DELETE-FIRST VARIANT

Purpose. Replace structured deletion-first proposals with unconstrained skill rewriting.

Inputs. {PARENT_SKILL} and {EXECUTION_EVIDENCE}.

Expected Output. A fully rewritten skill.

```
Rewrite the complete agent skill using the supplied execution evidence.

Current skill:
{PARENT_SKILL}

Execution evidence:
{EXECUTION_EVIDENCE}

Produce a clearer and more effective skill. You may remove, merge, reorder,
generalize, or add instructions, examples, executable utilities, validation
steps, and recovery procedures. Preserve useful behavior while improving
validation performance and concision.

Return only the rewritten skill.
```

D.9.5 TEXT-ONLY COMPRESSION

Purpose. Apply MSS operators only to natural-language modules while leaving executable utilities unchanged.

Inputs. The same inputs as full MSS, with module types supplied by the parser.

Expected Output. A candidate skill whose executable modules are byte-identical to those in the parent skill.

```
Apply the corresponding MSS edit prompt only to modules whose type is one of:
procedural_rule, constraint, example, reference, or validation_instruction.

Executable components are immutable. Copy every executable component exactly
from the parent skill into the candidate skill. Reject any proposal that
changes executable code, command definitions, scripts, or utility files.
```

D.9.6 NO DUPLICATE-RULE MERGING

Purpose. Disable the deduplication operator.

Inputs. The same inputs as full MSS.

Expected Output. Deletion, abstraction, and sanitization proposals only.

```
Use the full MSS prompts except the Duplicate-Rule Merging prompt. Do not merge
multiple procedural modules into a replacement rule. If deleting the relevant
modules violates the audit constraints, retain them unchanged.
```

D.9.7 NO EXAMPLE ABSTRACTION

Purpose. Disable replacement of instance-level demonstrations with task-level invariants.

Inputs. The same inputs as full MSS.

Expected Output. Deletion, deduplication, and sanitization proposals only.

```
Use the full MSS prompts except the Example Abstraction prompt. Examples may be
deleted if the resulting candidate passes the audit, but they must not be
rewritten as task-level invariants. If deleting an example violates the audit
constraints, retain it unchanged.
```

D.10 OUTPUT FORMATTING

Prompts that directly generate a deployable skill return only the complete skill, without Markdown fences, commentary, scores, or analysis. Prompts used for parsing, proposal management, probing, or auditing return strict JSON. All JSON outputs must satisfy the following requirements:

- keys and string values use double quotation marks;
- no comments, trailing commas, or text outside the JSON object are permitted;
- module identifiers remain stable across edit rounds unless modules are deleted or merged;
- `candidate_skill` contains the complete candidate artifact rather than a patch;
- absent evidence is represented by an empty list or `null`, not by an inferred value; and
- numeric scores used by the acceptance test are computed by the execution harness rather than generated by the proposal model.

At deployment, only the selected skill, task instruction, observable environment, tool specifications, and required output format are inserted into the main inference prompt. Evolution instances, audit traces, parent skills, probe outputs, proposal rationales, and evaluator-specific information are excluded from the executor context.

E CASE STUDY

This section qualitatively examines how deletion-first evolution changes the procedural behavior induced by a skill. We focus on representative examples from document processing, spreadsheet manipulation, information retrieval, and code-assisted data analysis. The examples illustrate three recurring distinctions: removing instance-bound instructions without removing their underlying invariant, preserving recovery behavior that appears unnecessary on clean inputs, and jointly simplifying natural-language rules and executable utilities. All identifiers and file contents are anonymized, while the procedural decisions and output structures are preserved.

E.1 From Template-Specific Instructions to a Document-Level Invariant

Consider a document-processing task in which the agent must update the heading structure of a report while preserving the body text, embedded references, and existing style relationships. The initial skill was evolved from several reports that happened to use the heading names `Overview`, `Findings`, and `Next Steps`. It therefore contained both a general formatting procedure and several instructions tied to those observed documents.

Initial skill fragment.

```
When revising a report:
1. Find the sections "Overview", "Findings", and "Next Steps".
2. Apply Heading 1 to "Overview" and "Findings".
3. Apply Heading 2 to "Next Steps".
4. If the document is named review_packet.docx, preserve the blue
   title style used in the reference output.
5. Do not modify paragraph text.
6. Before saving, verify that every required section has the exact
   heading name used by the evaluator.
7. Keep the original runs when changing a paragraph style so that
   hyperlinks, citations, and inline emphasis are not reconstructed.
```

Several parts of this fragment are dispensable. The filenames, literal section names, and reference to the evaluator encode properties of the construction instances rather than the task. However, the final rule about preserving runs is behaviorally important. A superficially plausible implementation that replaces a paragraph with newly generated text can silently drop hyperlinks, citation fields, bookmarks, or mixed inline formatting.

A free-form summary produced the following shorter instruction:

```
Identify the report sections, assign the requested heading styles,
preserve the document text, and save the updated file.
```

This summary retains the apparent objective but removes the operational meaning of preservation. On a held-out report whose headings were `Background`, `Risk Assessment`, and `Mitigation Plan`, the agent correctly assigned heading styles but rebuilt each heading paragraph from its visible text. The resulting document looked correct when rendered, yet a bookmark attached to `Risk Assessment` and an inline reference marker in `Mitigation Plan` were lost. The summary preserved surface intent but not the document-editing action required to realize it.

MSS first proposed deleting the example-specific rules. Counterfactual executions showed no loss when the filename, fixed heading names, and evaluator phrase were removed. By contrast, deleting the run-preservation rule changed the document structure even though the rendered pages remained visually similar. MSS therefore retained the rule and abstracted the remaining examples into a structural invariant:

```
For each section identified by the task, change the paragraph-level
style rather than rebuilding its text runs. Preserve inline formatting,
fields, hyperlinks, bookmarks, and paragraph content. Resolve section
roles from the requested hierarchy, not from literal heading names.
After editing, reopen the saved document and verify the heading levels
and the continued presence of non-text elements attached to edited
paragraphs.
```

With this skill, the agent treated `Background` and `Risk Assessment` as top-level sections and `Mitigation Plan` as a nested section because that hierarchy was specified by the task, not because the names matched a stored example. Its validation step inspected both paragraph styles and attached

document objects. The important compression was therefore not merely the deletion of proper nouns. MSS replaced a list of remembered headings with a decision rule about section roles while retaining the low-level editing constraint that protected document semantics.

E.2 Preserving Recovery Logic in Spreadsheet Manipulation

A second case involves a workbook-editing task. The user requests a derived status column, consistent formatting, and preservation of all existing formulas. The held-out workbook differs from the evolution examples in two ways: the relevant worksheet has a previously unseen name, and the table contains a blank separator row followed by additional valid records.

The initial skill included redundant instructions, a hard-coded sheet name, and a useful but easily overlooked rule about determining the actual data region:

```
Open the worksheet "Quarterly Review".
The identifier is usually in column A and the due date is in column F.
Stop processing when an empty row is encountered.
Use the same fill color as cell H2.
Copy the formula from H2 downward.
Never overwrite an existing formula.
Before saving, inspect the final non-empty row in the identifier column.
If a blank row occurs inside the used range, continue scanning for
later records.
Reopen the workbook and check that formulas remain formulas.
```

The instructions "stop processing when an empty row is encountered" and "continue scanning for later records" conflict. The first rule was introduced by an early evolution instance containing a compact table; the second was added after a later failure involving sparse rows. An additive evolution method retained both because each had once been associated with a successful trajectory.

On the held-out workbook, an executor following the earlier rule added the status formula only above the separator. Its final response was:

```
Updated the requested status column and saved the completed workbook.
```

The response was fluent but misleading. Records after the internal blank row were untouched, and the agent had not checked whether the formula range covered all populated identifiers.

MSS treated the two stopping rules as candidates for deduplication and counterfactual testing. Removing the sparse-range rule caused the agent to terminate at the separator. Removing the first-empty-row rule did not harm behavior when the remaining rule was expressed in terms of the last semantically populated record. The worksheet name, fixed column letters, and reference to cell `H2` were also deleted because the task supplied equivalent information through headers and format relationships. The resulting skill was:

```
Locate the target table and required fields from worksheet content and
header semantics rather than stored sheet names or column letters.
Determine the processing range from the last row containing a record
identifier; an internal blank row is not an end-of-table marker.

For each record in that range:
- preserve every pre-existing formula and value;
- write the requested derived expression only into the designated
  output field;
- obtain formatting from the nearest valid peer cell in the same
  semantic column, not from a fixed coordinate.

After saving, reopen the workbook and verify:
```

```
- all identified records, including those after internal blanks, have
  been considered;
- cells that originally contained formulas still contain formulas;
- the new expressions reference their own rows.
```

This version changes the agent's behavior at three concrete decision points. It discovers the worksheet from content rather than a memorized tab name, uses record identity rather than visual contiguity to bound the table, and selects a formatting exemplar by semantic proximity rather than by coordinate. It also preserves a reopening check because the distinction between a formula and its cached value may not be visible in a rendered preview. The case demonstrates why a rule that is rarely activated on clean workbooks can still be part of minimal sufficient procedural knowledge.

E.3 Removing Verifier Language from Information Retrieval

In an information-retrieval task, the agent must identify an official policy, distinguish its publication date from its effective date, and return a concise answer with source support. During evolution, several instances were checked by a verifier that searched for literal phrases such as `effective on` and expected a particular ordering of fields. The evolved skill consequently mixed a legitimate evidence procedure with instructions that targeted the verifier.

Bloated retrieval skill.

```
Prefer official sources.
Search for the policy title followed by "effective on".
Use the sentence that contains the exact phrase "effective on".
Place the effective date before the publication date because the
checker reads the first date.
If the page contains both dates, copy the surrounding sentence.
If search results disagree, open the issuing organization's page and
inspect the document itself.
Distinguish announcement, publication, revision, and effective dates.
Record which source supports each date.
Do not infer an effective date from the article timestamp.
```

The literal-phrase rules work only when the source uses the same wording as the construction examples. In the case examined here, the official notice states that the policy "takes effect beginning" on a specified day, while a secondary article uses the phrase "effective on" but associates it with an earlier draft. An agent guided by the verifier language selected the secondary article because it provided a lexical match. Its answer conflated the draft announcement with the operative date:

```
The policy became effective when the draft was announced. The official
notice was published later.
```

MSS removed the phrase-matching and output-order instructions. This deletion was admissible because the behaviorally relevant components were source authority, event-type disambiguation, and claim-level provenance. Deleting the instruction to distinguish date roles was not admissible: without it, the agent again copied the first visible date. The minimal skill retained the following procedure:

```
Resolve temporal claims by event type, not by lexical form.

1. Identify the issuing authority and locate its official notice or
   policy document.
2. Extract each date together with the event it denotes, such as
   announcement, publication, revision, or entry into force.
3. Treat expressions such as "takes effect", "applies from", and
   "effective beginning" as possible realizations of the same event,
```

```
   but confirm the subject and scope of the clause.
4. Use secondary sources for discovery or context, not to override a
   conflicting primary-source date.
5. In the final answer, attach each date to its event and source. If the
   official text does not specify an effective date, state that absence
   rather than substituting the publication timestamp.
```

The resulting answer was substantively different:

```
The issuing authority published the notice on [publication date]. The
notice states that the policy takes effect beginning [effective date].
The earlier date reported by the secondary article refers to the draft
announcement, not to the policy's entry into force.
```

The retained skill does not prescribe a fixed sentence template. It prescribes a small set of evidence decisions that remain meaningful when terminology, page layout, or answer format changes. This case also shows why verifier sanitization must be followed by behavioral auditing: removing all date-related instructions would produce a shorter skill, but it would also remove the distinction that enables correct temporal grounding.

E.4 Joint Compression of Analysis Instructions and Utilities

The final case concerns code-assisted analysis of a delimited data file. The task asks the agent to identify records with inconsistent category labels, normalize only unambiguous variants, and produce a cleaned file together with a short audit summary. The initial skill bundled prose with two overlapping utilities.

```
Procedure:
- Load customer_export.csv with UTF-8 encoding.
- Category values are in the segment_name column.
- Convert "enterprise ", "Enterprise", and "ENT" to "Enterprise".
- Replace missing values with "Unknown".
- Run normalize_segment.py.
- Run cleanup_labels.py afterward to catch remaining variants.
- The evaluator expects the cleaned file to contain the column
  normalized_segment.
- Print "normalization complete" after writing the file.

normalize_segment.py:
    value = value.strip().lower()
    aliases = {
        "enterprise": "Enterprise",
        "ent": "Enterprise",
        "small business": "SMB"
    }
    return aliases.get(value, value.title())

cleanup_labels.py:
    value = value.strip()
    if value.lower().startswith("ent"):
        return "Enterprise"
    if "small" in value.lower():
        return "SMB"
    return value
```

The second utility appears to increase coverage but changes the semantics of the task. For example, it maps `Entity Services` to `Enterprise` because both begin with the same letters, and maps `Smallholder Program` to `SMB` because it contains the word `small`. The two scripts also make

normalization order dependent. A value modified by the first script is silently accepted by the second, while an unfamiliar value may be changed by a broad substring rule without appearing in the audit summary.

A text-only compressor shortened the prose but left both utilities unchanged. The executor consequently produced the requested schema and completion message while still introducing unsupported category changes. This illustrates a limitation of treating code as an opaque attachment: the natural-language skill may become concise even though redundant and unsafe behavior remains in the executable component.

MSS represented the scripts and their call sites as dependent modules. It proposed removing `cleanup_labels.py` together with its invocation. Counterfactual inspection showed that exact aliases and case-or-whitespace variants remained normalized by the first utility, whereas ambiguous substring matches were no longer collapsed. The hard-coded filename, column name, output phrase, and evaluator-specific schema instruction were also removed. The retained hybrid skill was:

```
Identify the input file and target categorical field from the task
specification. Preserve the original field and write normalized values
to the requested output field.

Normalization is conservative:
- remove surrounding whitespace;
- compare labels case-insensitively;
- apply only an explicit alias mapping justified by repeated equivalent
  labels or task-provided definitions;
- do not use prefix, substring, or edit-distance matches to merge
  otherwise distinct categories;
- preserve missing values unless the task explicitly defines a
  replacement;
- leave an unfamiliar label unchanged and include it in the audit
  summary.

Use one normalization function for both transformation and reporting:

def normalize(value, aliases):
    if value is missing:
        return value, "missing"
    key = strip(value).casefold()
    if key in aliases:
        return aliases[key], "mapped"
    if key differs from value only by case or surrounding whitespace:
        return strip(value), "surface-normalized"
    return value, "unchanged"

Before writing the output, assert that the record count is unchanged
and that every changed non-missing value is explained by an explicit
alias or a surface-only normalization.
```

For the ambiguous labels, the final audit output was:

```
"enterprise "     -> "Enterprise"      [surface-normalized]
"ENT"             -> "Enterprise"      [mapped by explicit alias]
"Entity Services" -> "Entity Services" [unchanged; review suggested]
"Smallholder Program"
                  -> "Smallholder Program"
                                       [unchanged; review suggested]
```

The compressed skill is more reliable because it makes the admissible transformation boundary explicit. It also couples transformation and reporting through the same function, preventing the audit summary from describing a different rule than the one used to modify the data. The essential

procedural knowledge is not the list of labels observed during evolution, but the distinction between surface normalization, explicit semantic aliases, and ambiguous values requiring preservation.

E.5 Cross-Case Observations

Across the four cases, MSS consistently removes content that answers "what did the construction instance look like?" while retaining content that answers "what decision must the executor make?" Literal filenames, worksheet names, cell coordinates, output phrases, and verifier-preferred wording are removed when the task or environment already supplies the relevant information. In contrast, structural editing constraints, sparse-range handling, temporal event disambiguation, conservative transformation boundaries, and post-save validation remain because they determine whether an apparently successful execution preserves the intended behavior.

The cases also clarify the role of counterfactual auditing. A module is not retained merely because it sounds prudent, nor deleted merely because it is rarely activated. Its status depends on the behavior observed when that module is absent. This criterion preserves low-frequency recovery procedures, such as scanning beyond an internal blank row, while eliminating high-frequency but incidental details, such as a worksheet name repeated in every evolution example.

Finally, minimality is achieved at the level of the complete skill artifact. Compressing prose alone can leave obsolete scripts, overlapping utilities, or unsafe matching rules active. Conversely, deleting executable code without repairing its textual call sites can produce an invalid skill. By reducing instructions and utilities under shared dependency and behavioral constraints, MSS yields procedures whose remaining components have explicit operational roles rather than historical associations with individual training instances.