SKILLZIP PRO: EXECUTION-AWARE DYNAMIC COMPRESSION OF PROGRESSIVELY LOADED SKILLS FOR SELF-EVOLVING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Self-evolving agents accumulate useful procedures, but also repeat the same rules across references and subskills. Compressing these skills as flat prompts misses a crucial distinction: the agent stores a whole directory but loads only the files needed for a task. A smaller directory can therefore increase execution context or leave a required subskill inaccessible. We introduce SKILLZIP PRO, an evaluation-free compiler that compresses *content and its loading structure together*. It shares repeated instructions within the branches that use them, removes obligations already supplied by declared context, and defers explicitly guarded sections. Typed removal evidence and an independent audit of the emitted files preserve routing, interfaces, and standalone entries. The output uses ordinary files without changing the agent harness. One-Shot and Continual schedules support both rewritten bundles and source-preserving execution views. Across three task suites, SKILLZIP PRO saves 19.7% of shipped and 14.3% of mean-run tokens while preserving every declared route and meeting a five-point quality non-inferiority criterion. *On a production moderation service,* SKILLZIP PRO *shows 38.1% skill token and 10.4% end-to-end runtime token savings, with nearly no performance drop.* Code available at: https://anonymous.4open.science/r/skillzip-pro-ICLR

1 INTRODUCTION

A self-evolving agent improves by turning experience into reusable instructions. A failed tool call adds a warning; an invalid answer adds a format check; a successful attempt adds a procedure. These edits are useful individually, yet their accumulation need not be efficient. The same output rule may appear in several branches, and the same verify–repair workflow may be copied after each failure. Importantly, a warning that appears only once may encode an expensive lesson. The maintenance problem is therefore **to remove repeated representations without forgetting learned requirements**.

This problem becomes more demanding when a skill is a directory rather than one document. Production skill formats use a root file to route tasks to references, nested subskills, scripts, and schemas. Catalog metadata is visible before selection, the root is loaded at activation, and auxiliary files are read only when needed (Anthropic, 2025; OpenAI, 2025). Self-evolution grows the entire directory, whereas a specific execution only activates part of it. Thus, **storage reduction and execution savings are different objectives and must therefore be measured separately**.

Existing compression strategies miss different parts of this structure (Figure 1). Rewriting only the root leaves repetition in other files and may erase the instructions that lead to them. Flattening all files exposes that repetition but removes the boundaries that enable progressive loading. Even cross-file deduplication is not sufficient: a rule shared by rare branches becomes more expensive if moved into the always-loaded root. Nor does retaining a file guarantee usability. A directly callable subskill may still need a rule that is redundant only when the root has already been loaded.

These failures point to a single question: *where can an instruction be stored once while remaining available to every execution resource that needs it?* SKILLZIP PRO answers by treating compression as constrained optimization over an execution graph. It jointly considers the content to retain, the scope at which to load it, and the routes through which it is reached. The result remains an ordinary directory that the existing agent can use without a new execution protocol.

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.

To see why placement matters, consider two mutually exclusive branches, each used by 5% of tasks, that repeat a 300-token rule. Moving the rule to the root saves one stored copy but adds **270 expected body tokens per invocation**, before reference overhead. A shared file read only by those branches avoids that penalty. If a branch is also a public subskill, its own entry must still lead to the shared rule: an unseen root cannot supply missing context.

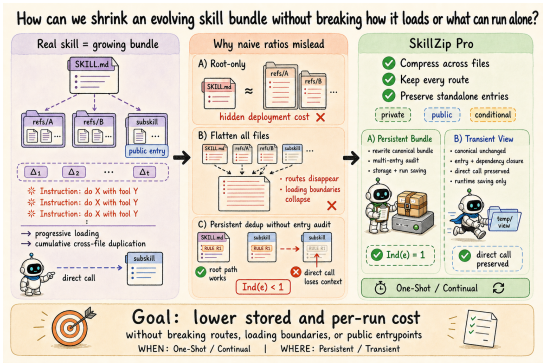

Figure 1: Why a shorter skill can be worse. Root-only rewriting misses distributed repetition; flattening loses loading boundaries; cross-file deletion can break direct calls. Pro preserves both content and access.

SKILLZIP PRO combines two mechanisms. **Activation-aware cross-file compression** finds reuse within the scope that needs it. **Cross-file routing preservation** checks that the result remains accessible. Rewrites use source structure and declared context, not task-rollout feedback. Held-out execution tests the artifact, rather than selecting candidates. SKILLZIP PRO separates *when to compress*—One-Shot rebuilding or Continual updates—from *what to publish*: a rewritten Persistent bundle or a Transient view that leaves the source unchanged.

Our contributions are:

- **Bundle-aware compression for progressively loaded skills.** Scope-aware placement removes cross-file redundancy without inflating the root or dropping rare rules.
- **Safe routing and independent entry preservation.** Audited routing and entry contracts preserve branch reachability and public-entry usability in the emitted directory.
- **Flexible compression for different update and deployment requirements.** Orthogonal modes and lifecycles support static and self-evolving skills with one shared kernel.
- **Comprehensive compression results with production evidence.** We remove content only when its requirements are already covered by the retained instructions or a declared environment contract. On a production skill, SKILLZIP PRO reduces deployed content by **38%** and end-to-end per-run tokens by **10.4%** while preserving decision quality.

2 RELATED WORK

Skill evolution and prompt compression. Voyager, Reflexion, and ACE retain programs, feedback, and playbooks (Wang et al., 2024; Shinn et al., 2023; Zhang et al., 2025); SkillOpt, SkillRevise, and SkillGrad improve skill artifacts (Yang et al., 2026; Liu et al., 2026; Wang et al., 2026). Prompt compressors select or rewrite inference context (Jiang et al., 2023; 2024; Pan et al., 2024; Li et al., 2023; Mu et al., 2023; Xu et al., 2024). For reusable skills, however, an infrequent rule can remain mandatory: *rarity justifies delayed loading, not deletion*, because future tasks may still need it.

Skill compression. SkillReducer selects rewrites using task feedback (Gao et al., 2026), adding rollout cost and dependence on the evaluation tasks. SKILLZIP (Bai et al., 2026) instead finds an evaluation-free typed cover within skill. Its single-document-level compression objective does not price progressively loading. SKILLZIP PRO addresses both limitations: it optimizes *where* shared instructions reside and checks coverage from *each declared entry*, while remaining evaluation-free.

Graph compression and execution representations. Another independently developed SkillZip (Tan et al., 2026) compresses skill library graphs into reversible macros, constructs query-specific context through graph retrieval and macro expansion, and uses execution evidence to revise those macros. This requires a graph-based context-provider layer rather than a replacement skill directory. Our SKILLZIP PRO instead compiles **ordinary files with unchanged loading interfaces**, explicitly accounting for always-loaded versus branch-local content and auditing public entry paths. Persistent mode avoids query-time macro expansion while transient mode doesn't modify disk files.

3 PROBLEM: COMPRESS CONTENT WITHOUT LOSING ACCESS

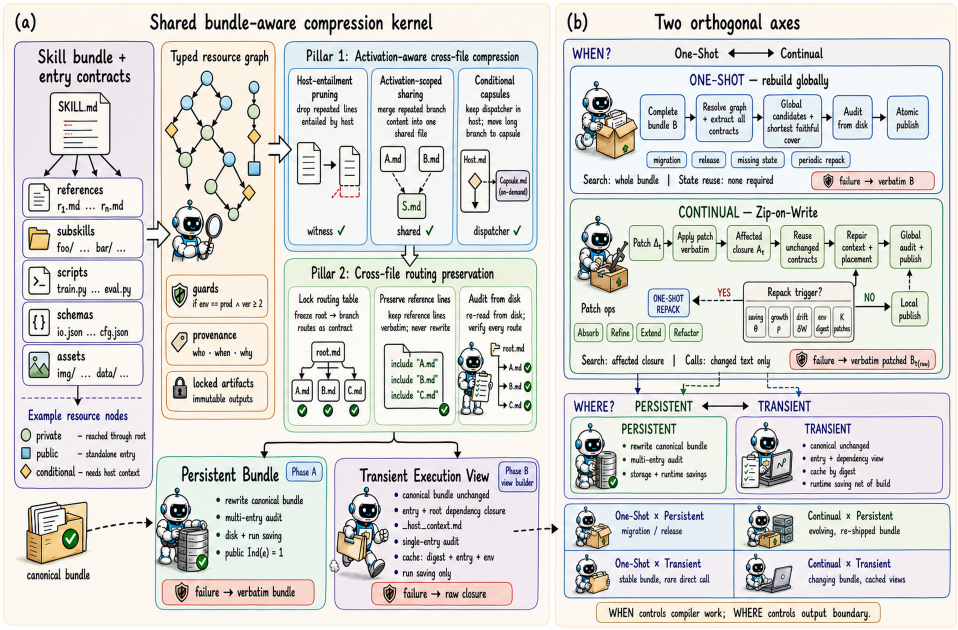

Figure 2: SKILLZIP PRO overview. The shared compiler reduces cross-file repetition and checks the emitted directory. One-Shot/Continual determines when it runs; Persistent/Transient determines whether the result replaces the bundle or forms a temporary execution view. Compiler state is not needed by the executing agent.

What the compressor can observe. SKILLZIP PRO receives the source directory, declared entry contracts, and optionally a tokenizer, loading-frequency estimates, and an explicit environment contract. Task outcomes, rewards, and behavioral verifiers do not select rewrites. *Evaluation-free* describes compression, not the subsequent tests of effectiveness.

What must remain available. We represent the bundle as $B = (V, E, r)$, rooted at `SKILL.md`. Each edge (u, v, g, s) records a local reference, its explicit guard g, and source span s. Each text file carries typed units for applicability, workflow order, tools, rules, outputs, and evidence, with source provenance and locked residuals. Coverage preserves type, polarity, guard, and scope; an example cannot replace a prohibition. Code, data, and binary artifacts retain their paths and bytes. The resolver neither fetches external links nor invents dependencies; strict mode rejects unsafe or unresolved local paths. The author of the pre-compression skill declares entries *private*, *public*, or *conditional* on supplied host context. A public entry must be discoverable at its advertised path and cover all its requirements through its own reachable files, without relying on an unseen root:

$$\mathrm{Ind}(e, B') = \mathrm{Cov}(e, B'_{\downarrow e})\, \mathbf{1}\{e \text{ is discoverable}\} = 1. \tag{1}$$

Here Cov is the fraction of covered source units; $B'_{\downarrow e}$ includes declared context only. This excludes deleting a public subskill's rule merely because the root also contains it.

What the compressor minimizes. For length function τ, catalog cost counts selection metadata, activation cost the root, deployment cost all shipped content, and path cost subsequent loaded resources. We minimize their combined cost to account for both storage and expected loading:

$$J(B') = C_{\mathrm{cat}}(B') + C_{\mathrm{act}}(B') + \mathbb{E}_{\pi\sim q} C_{\mathrm{path}}(B'; \pi) + \lambda C_{\mathrm{dep}}(B') \tag{2}$$

subject to coverage, routing, entry, and locked-artifact constraints. Initial activation is counted once; later reads are separate events. We use $\lambda = .05$ and declared traces or uniform guarded-leaf weights for q. Crucially, **weights price placement, not whether a lesson may be forgotten**. Rare rules remain covered. Actual behavior and runtime are evaluated separately. Appendix A defines each cost and specifies how repeated reads, entry context, and deployment are accounted for.

4 METHOD: REUSE WITHIN SCOPE, THEN AUDIT EVERY ENTRY

4.1 ACTIVATION-AWARE CROSS-FILE COMPRESSION

SKILLZIP PRO first resolves the resource graph and extracts source-grounded contracts. The file-level optimizer covers typed units with statements, shared rules, workflows, and exceptions (Bai et al., 2026). The following transformations add opportunities that no file can expose in isolation.

Prune content already supplied by the host. *Host-entailment pruning* removes repeated content when retained root context covers it on every applicable entry path, or a declared environment contract provides an exact type/key/value/scope match. Environment witnesses are bound to a digest; a changed environment requires re-auditing. Without such a contract, environment pruning is disabled. Public entries must satisfy their requirements through their own files and declared host context.

Share within the required loading scope. For a fragment of d_x tokens repeated in files S_x, *activation-scoped sharing* creates module h and replaces each copy with a mandatory relative read only when the shared definition and its references cost less than retaining all copies:

$$w_h d_x + \sum_{v \in S_x} w_v d_{\text{ref}} + d_{\text{overhead}} < \sum_{v \in S_x} w_v d_x. \tag{3}$$

The weights w_v include loading exposure and deployment; d_{ref} and d_{overhead} include emitted references and scaffolding. Sharing stays within its users' activation scope, so unrelated executions do not pay for branch-specific rules, while branches that need them retain explicit access.

Defer explicitly guarded branches. A *conditional capsule* moves an explicitly guarded body into an on-demand file, leaving its trigger and mandatory read at the source. For body length d_b, dispatcher length d_g, and trigger probability p_g, the move helps when

$$(1 - p_g) d_b > (1 + \lambda) d_g. \tag{4}$$

A 600-token body with a 20-token dispatcher, used on 10% of paths, saves 520 expected load tokens before storage weighting. Sharing removes copies; capsules avoid unnecessary reads. Neither invents a guard. Because costs interact, selection uses improving marginal moves and local add/drop/swap refinement, not a global optimum. Appendix B derives both tests.

4.2 CROSS-FILE ROUTING AND ENTRY PRESERVATION

A cheaper layout is useful only if the agent can follow it. SKILLZIP PRO locks routing instructions and preserves reference-bearing lines during rendering. It then **reopens the emitted files**, independently checking resolved paths, guards, scoped coverage, public-entry discoverability, and locked-file hashes. Schemas, legal labels, and worked formats must appear as complete, contiguous, verbatim sections; retaining scattered lines does not preserve the specification as one complete unit.

Every removal also records a witness: W_1 is literal containment at a retained reachable site; W_2 is deterministic coverage with protected literals and polarity; W_3 is a frozen model's logged entailment judgment. W_3 is restricted to evidence without prohibition, output, or exemption markers. Intra-root checks exclude the candidate itself and veto overlapping deletion candidates. These are auditable checks, not infallible semantic proofs: preservation is conditional on extraction and the final auditor.

4.3 KEEP THE SAME GUARANTEES ACROSS UPDATES AND LIFECYCLES

Following SKILLZIP (Bai et al., 2026), SKILLZIP PRO also has two modes. **One-Shot mode** compresses the evolved skills by offline compression. **Continual mode** first applies an authored patch verbatim, reuses digest-identical contracts, and repairs changed files plus affected references, shared supports, guards, and host witnesses. Periodic repacking revisits cross-round reuse; both schedules audit the entire publication, while the **Continual mode** is designed to be compatible with self-evolving agents. Independently, **Persistent** publication replaces the shipped bundle and audits all public entries. **Transient** publication leaves the canonical source unchanged and builds an entry-rooted view with its dependencies and declared host context. Cache keys include bundle, entry, environment, and any conditioning inputs. Failed or non-improving views use the raw closure. All outputs use ordinary relative files; Appendix C gives the transactional algorithms.

5 EXPERIMENTS

We evaluate task quality and loading cost, explain savings during evolution, and test whether compressed skills remain usable through ablations, direct calls, and production measurements.

5.1 TASK QUALITY, LOADING COST, AND RETAINED CONTENT

Setup. SkillOpt (Yang et al., 2026) evolves task-class skills; a deterministic builder places their original text in guarded references and subskills. Qwen3.6-Plus executes 21 BFCL-v4 (Patil et al., 2025; Gorilla Team, 2025), 33 Math/LiveMath (He et al., 2026), and 48 SpreadsheetBench tasks (Ma et al., 2024). Compressors receive identical source bundles and environment contracts, without held-out outcomes. Appendix G specifies the adapters and measurement boundaries.

Comparisons. Root-only and Flat-concat apply single-document SKILLZIP (Bai et al., 2026) to the root or concatenated files. SkillReducer uses its compression stage with rollout selection disabled, not its full feedback-guided pipeline. Expert Progressive shares repeated paragraphs through a root-linked file; No Skill and Human Skill are quality references, not alternative compressors.

Table 1: Quality, cost, and access on grown bundles. Savings are percentages versus Evolved. Kept/Route: reachable-line/declared-route retention; Reuse: repeated text removed. These diagnostics are not semantic certificates. [a]Compression-only adaptation; †routing-damaging control.

	Task success ↑				Token saving (%) ↑				Structural Fidelity ↑		
Method	BFCL	Math	Sheet	Macro	Ship	Root	Run	Worst	Kept	Route	Reuse
No Skill	.809	.364	.312	.495	No reusable skill; quality reference only						
Human Skill	.905	.455	.333	.564	Before evolution; quality reference only						
Evolved	.905	.364	.354	.541	0.0	0.0	0.0	0.0	1.000	1.000	.000
Root-only†	.905	.333	.271	.503	2.2	27.8	10.3	8.2	.939	.000	.002
Flat-concat†	.857	.394	.167	.473	63.8	42.9	68.9	67.3	.241	.000	1.000
SkillReducer[a]	.857	.394	.271	.507	5.0	−.4	4.1	4.4	1.000	1.000	.073
Expert Prog.	.905	.333	.312	.517	13.7	−6.5	17.9	.2	1.000	1.000	.482
SKILLZIP PRO	**.952**	.394	.333	**.560**	**19.7**	7.3	14.3	10.8	.986	**1.000**	.736

Useful savings require retained content and access. SKILLZIP PRO has the **highest macro success among compressors** and lowers all four costs. Expert saves more mean-run tokens but expands the root; Flat-concat loses content and routing. SKILLZIP PRO reclaims **73.6% of repetition**, retains .986 of source lines, and keeps every declared route. Its paired quality interval clears the reported −5-point margin (Figure 3a), not an exact-equivalence or superiority test. Figure 3(b–d) previews how the start time of compression affects cumulative savings, examined next.

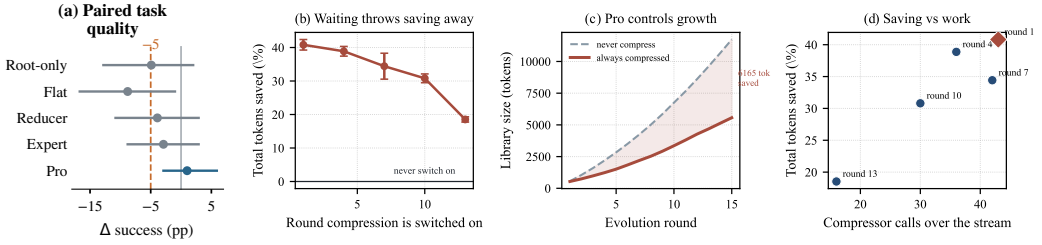

Figure 3: Task quality and the value of early compression. (a) Paired 95% quality intervals over 102 tasks. (b) Published-size saving versus the round compression starts; bars show variation across three libraries. (c) Library size with and without compression. (d) Saving versus processing invocations. Panels (b–d) are analyzed in Section 5.2 and enlarged in Figure 12. "Compressor calls" counts processed text nodes, not API calls; cumulative size is not billed runtime. It sums the sizes of all published bundles over the entire 15-round evolution stream.

5.2 Why Reuse Becomes More Valuable during Self-Evolution

We next retain all 15 rounds of each recorded run as a progressively loaded library: three libraries, 17 files each, about 20,000 tokens, and 79–84% repeated text. One-Shot SkillZip Pro saves **23.1% shipped and 17.7% mean-run tokens**, with full measured line and route retention. These all-library means include one verbatim fallback; the two accepted outputs average 34.7% shipped saving. Expert saves 25.0% across all three but expands the root by 2.9% (Table 8).

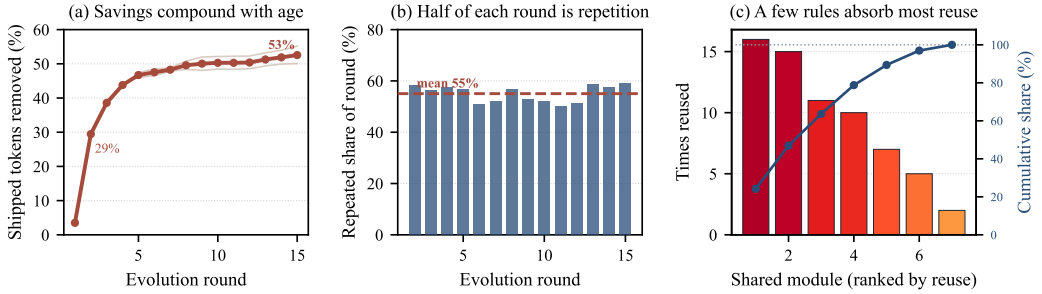

Figure 4: Self-evolution repeatedly creates opportunities for reuse. (a) Shipped saving grows from 29% to 53%. (b) The marginal compressed-size reuse proxy averages 55%±3%. (c) Reuse is concentrated: the top two of seven modules account for **47%** and the top three for **64%** of all reuse. Prefix results include the two libraries accepted at every prefix. Output formats and precision rules recur across branches, so a small shared library covers most of the redundancy while keeping each branch's distinct procedures and guards explicit.

Figure 4 shows common output formats, precision rules, and checks stored once, with local guards and procedures kept explicit. On recorded runs with identical patches, repacking raises shipped-token savings from **2.7% to 48.1%** using about one third of rebuild work. It also matches the rebuild's final objective on structured patches. **Repacking recovers repetition missed by local updates.** Table 2 gives costs and trigger sensitivity; Figure 13 shows the evolution diagnostics.

Table 2: Continual compression reduces token use and update cost. Real-library results average three 13-round runs; the structured replay is reported separately. Ship/Run denote shipped/per-run token savings (%), and R is the relative objective difference from full rebuilding. Work counts processed text nodes, not API calls.

	Real evolved libraries						Structured patch stream					
Schedule	Ship	Run	R	Work	Bytes	ms	Ship	Run	R	Work	Bytes	ms
Uncompressed	0.0	0.0	.000	0.00	3,201	0	0.0	0.0	.046	0.0	263	1
Full rebuild	35.1	18.7	.000	10.00	13,388	234	12.4	2.1	.000	8.8	6,250	170
Root-only	2.4	19.6	−.027	1.00	389	94	9.4	21.7	−.277	1.0	604	115
Pro, local	2.7	.8	.262	1.00	3,201	8	7.7	4.7	−.008	1.0	263	13
Pro + repack	**48.1**	**28.2**	−.083	**3.31**	**5,697**	**68**	**12.4**	2.1	.000	**3.3**	**1,875**	**60**

Trigger sensitivity: separate replay	Never	Every patch	Savings	Growth	Count	Combined
Repack fraction	.00	1.00	.00	.00	.25	.25
Worst relative drift	.0014	.0000	.0014	.0014	.0014	.0014
Work / patch	1.00	9.50	1.00	1.00	3.50	3.50

On recorded runs, update time falls from **234 to 68 ms** and processed bytes from 13,388 to 5,697 per round, with the same full audit. Negative drift compares a retained earlier compression with a later heuristic rebuild, not a global optimum. Only the patch-count trigger fires; the combined policy repacks 25% of updates. We also vary the start round (Figure 3b–d): starting at round 1 saves **40.8%** of summed published sizes versus 18.5% at round 13, using 43 versus 16 invocations (Table 14). This measures cumulative size, not billed runtime. Thus, **earlier compression costs more work but keeps intermediate versions smaller; later cleanup cannot recover the savings missed in earlier rounds, since the metric includes all published versions, not just the final compressed bundle.**

5.3 Which Changes Create Savings, and Which Preserve Usability?

The preceding results show savings with retained access, but do not isolate the contribution of each component. We first remove individual transformations and entry checks (Table 3), then execute the resulting bundles to test whether their instructions remain usable (Table 4).

Table 3: Reuse creates savings; entry contracts determine what may be removed. (a) Cost-component ablations: percentage savings except relative J; this ledger is separate from Table 1. (b) Full entry diagnostics, with auditing disabled or enabled and with an unchanged transient source. Run is canonical mean-run tokens; Disk is a byte ratio. Public independence combines entry-local coverage with discoverability.

(a) Cross-file cost components

Variant	J	Ship	Root	Run
SkillZip Pro	.142	**21.1**	7.3	**15.8**
No graph	.118	13.7	7.3	13.6
No host entailment	.077	15.3	2.0	8.3
No scoped sharing	.132	7.9	7.3	17.0
No capsules	.126	26.9	7.3	11.8
No disk audit	.142	21.1	7.3	15.8
No routing lock	.156	22.2	7.1	17.8
Global sharing†	−.166	17.0	−84.9	−.9

(b) Public and conditional entries

Metric	No audit†	Audited	Transient
Disk ↓	.845	.884	1.000
Run ↓	331	352	404
Routes ↑	1.000	1.000	1.000
Discoverable ↑	.500	1.000	1.000
Public content ↑	.929	1.000	1.000
Public Ind. ↑	.500	**1.000**	**1.000**
Minimum Ind. ↑	.000	1.000	1.000
Conditional Cov. ↑	.887	1.000	1.000

Removing scoped sharing cuts shipped savings from **21.1% to 7.9%**. Removing capsules increases storage savings to 26.9%, but lowers run savings from 15.8% to 11.8%: a dispatcher adds text while avoiding a body read. Thus, **sharing removes copies, whereas deferral avoids unnecessary loading**. Global sharing makes the root **84.9% larger** and leaves a file unreachable, supporting the placement tests in Eqs. (3)–(4). Varying $\lambda \in \{0, .01, .05, .1, .25, 1\}$ leaves the selected transformations unchanged on these bundles (Figure 15), indicating stability across the tested storage weights.

Placement must also respect standalone calls. Without the multi-entry audit, .929 of public content survives and every route remains valid, yet a renamed subskill loses its advertised direct-call path. Audited Pro retains all public entries with a .884 byte ratio and 352 mean-run tokens, down from 404. Single-document SkillZip remains valid but saves less (.959; 393 tokens). Pro achieves **more reuse under the same entry requirement**, not a smaller directory at the expense of callability.

Table 4: The unmodified agent must still find and use the retained instructions. (a) Actual loading on grown bundles: required-resource recall, irrelevant-load rate, and reachable files. (b) Direct-call startup and task success on the separate multi-entry bundle ($n = 3$ public tasks; $n = 8$ per conditional reference).

(a) Progressive loading

Method	Recall ↑	Irrel. ↓	Files
Evolved	.795	.171	6.3
Root-only†	.685	.290	6.3
Flat-concat†	.708	.259	6.0
SkillReducer	.688	.288	6.3
SkillZip Pro	**.755**	**.193**	10.3

(b) Independent direct calls

Method	Start	Public	Cond.
Uncompressed	1.000	.333	.375
Pro, no audit†	.000	.000	.417
Pro, audited	**1.000**	.333	.375
Pro, transient	1.000	.667	.417

Execution tests reveal what these structural checks do and do not guarantee. Pro's required-resource recall (.755) and irrelevant loading (.193) are closer to the raw bundle's than those of the listed compressors; its extra files are shared modules and capsules. Recall still falls below the raw .795, so retained routes do not ensure correct loading. All methods have zero link errors even when routing instructions disappear. Direct calls expose a different failure: the unaudited public entry cannot start, while audited persistent and transient outputs remain callable. This small sample tests startup, not broad quality superiority. Removing dependency closure reduces reachable references from 11 to 1; omitting host context lowers conditional coverage to .887 (Table 25). Together, **these results show that a smaller bundle is useful only if each public entry retains the instructions, routes, dependencies, and supplied context needed to run independently, not just through the root**.

5.4 Deployment Costs and Transfer across Executors

We next measure the cost of preparing an execution view. Persistent compression rewrites a reusable bundle; transient compression preserves the source but builds or retrieves a view. Table 5 compares entry-level savings, view construction, and maintenance under different calling patterns.

Table 5: Entry savings and workload costs must be considered together. (a) Tokens before/after view construction and latency in ms; expected latency uses a .9 hit rate. (b) Separate workload and cache sweeps, detailed in Appendix Tab. 20: P/T is persistent/transient, U edits per run, and d workload drift.

(a) Entry-specific views and preparation

Entry	Closure → view	Saved (%)	Cold / warm	Expected
Root	493→371	**24.8**	218/.34	22.1
Public subskill	548→545	.5	173/1.53	18.7
Algebra	532→529	.6	155/.23	15.7
Edge cases	513→505	1.6	153/.29	15.5
Geometry	489→489	.0	244/.54	24.9
Number theory	483→478	1.0	157/.56	16.2

(b) Reuse, updates, and cache sensitivity

Separate test and setting	Measured result
Rework, $U = 1$: P/T	273/183 ms
Saving, $d = 0$: P/T	12.9/4.7%
Saving, $d = .8$: P/T	2.6/4.7%
Cache on/off	18.7/173.4 ms
Cold build, 548→879 tokens	170→228 ms
Load, 1–50 stable calls	P < T

The root saves 122 tokens (**24.8%**), whereas other views save only 0–8 despite similar closure sizes (Table 5a). Thus, **reusable content within an entry's dependencies matters more than input length alone**. Savings use each raw closure, not the canonical mean. Geometry's 244-ms fallback also incurs work without saving tokens. Cold builds take 153–244 ms; **cached reads take .23–1.53 ms**. At a .9 hit rate, expected preparation is 15.5–24.9 ms, still dominated by cache misses. Removing the cache raises measured cost over 20 calls from 18.7 to 173.4 ms. Together, these results favor *reusing views with substantial shared content rather than rebuilding every entry indiscriminately*.

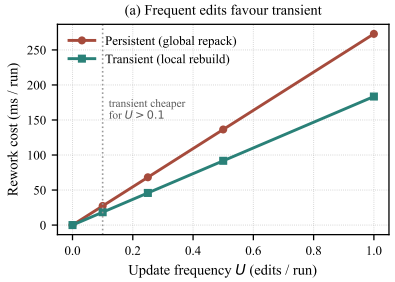

Figure 5: Update policy changes preparation cost. Local views cost less for $U > .1$ under the tested policies. Figure 16 also includes stable calls, which favor persistent files rather than repeatedly built views.

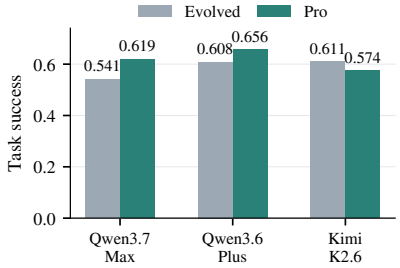

Figure 6: One bundle, three executors. The same files run without recompilation on a separate 44-task track. Both Qwen models improve, but Kimi declines. Full loading results appear in Table 26.

The workload tests extend this comparison beyond one call (Table 5b). Persistent files use fewer loaded tokens across 1–50 stable calls, but global repacking costs more than local view rebuilding when updates are frequent. Under workload drift, persistent savings fall from 12.9% to 2.6%, below the transient 4.7%. *Stable reuse favors persistent files; frequent change can favor local views*. These are policy-specific results: changing update schedules or cache-hit rates can shift the crossover.

Finally, we keep the compressed files fixed and vary the executor over 27 Math and 17 BFCL tasks. Success rises from .541 to .619 for Qwen3.7-Max and from .608 to .656 for Qwen3.6-Plus. Kimi falls from .611 to .574 despite higher required-resource recall (.435→.490), with irrelevant loading rising from .077 to .287. **One directory runs unchanged across all three executors, but its behavioral benefit remains model-dependent.** The remaining question is whether these savings extend beyond benchmark tasks to complete executions in a production service.

5.5 Production Deployment in a Content-Moderation Service

We deploy compressed files in a content-moderation service's **native production harness**, without changing its file interface. The Chinese bundle has a roughly 20,000-token root, three references, and a locked risk map. Each audit averages 13 reasoning rounds; each evaluation session retrieves 100 real business tasks at runtime, evenly split between violating and normal cases.

Across two paired sessions, full Pro removes **38.1% of shipped tokens** and **10.4% of end-to-end per-run tokens**. Accuracy is 89% in both sessions, against paired raw baselines of 92% and 91% (Table 6). These measurements show **a runtime saving in the existing service, with an observed 2–3-point quality trade-off**, rather than a saving inferred from shorter files alone.

Table 6: Compression in the service's native production harness. Savings and accuracy are percentages. (a) I/S denotes an intact/split interface. (b) Paired runtime measurements; the cross-day pilot and static estimate are separate. *Full Pro's controls appear in (b). ‡FP=9 is configuration-level, not a count for each session.

(a) Protection, compression, and decision quality

Configuration	Ship	Run	Accuracy Pro/raw	FP	I/S
Unprotected structural	71.4	—	70/88	29	—
Unprotected + model	75.8	—	62/88	35	—
W_1, W_2; C1/C2	13.8	7.1	89/88	9	S
+ reference W_3	32.7	6.8	88/88	10	S
+ C0 restoration	32.2	6.1	89/88	9	I
Full Pro	**38.1**	**10.4**	89*	9‡	I

(b) Runtime and session-level evidence

Evaluation	n	Run saved	Accuracy Pro/raw
Session 1	100	9.5	89/92
Session 2	100	11.3	89/91
Pooled	200	**10.4**	—
Pilot (days)	30	6–14	—
Static	—	∼3	—

Pooled n counts session–task observations, not necessarily unique tasks.

An English-only extractor marks just 7 of 264 Chinese units as required. Unprotected compression removes 71.4–75.8% of shipped tokens but loses 18–26 accuracy points: false positives rise from 10 to 29/35, while false negatives remain between one and three. Inspection identifies missing authorized-case rules, whitelists, and evidence thresholds. Pro therefore detects language-specific obligations, locks exemptions (C1), marks required rules (C2), and protects interfaces (C0). *These exceptions define correct business decisions; they are not optional background.*

With required rules protected, literal and deterministic coverage checks recover 13.8% deployment saving. Restricted reference-level W_3 raises this to **32.7%**, with 106 logged decisions and no exemptions or outputs eligible for deletion. Reuniting the two JSON examples and legal-label list restores an intact C0 section for only .5 percentage points of saving. **Interface integrity is verified separately from accuracy**: the one-task change does not establish a causal gain. These model-assisted checks have costs separate from the deterministic compiler timings.

Repeated root use makes its compression especially valuable: removed root tokens are associated with five to six times the runtime saving of removed reference tokens. Restricted intra-root W_3 yields the final configuration while excluding boundaries, interfaces, and self-supporting deletions. Full-session savings of 9.5% and 11.3% exceed the roughly 3% static estimate; the separate pilot varies across days (Table 6b). Although these records do not isolate changes in execution, they support **measuring production benefits end to end, alongside same-session quality checks**, rather than using static estimates as a substitute for measurements from complete multi-round service executions.

6 Limitations and Conclusion

Structural preservation depends on extraction and auditing; available instructions may still be followed incorrectly. The quality test supports a five-point tolerance, not exact equivalence. Prefix analyses condition on accepted outputs, and production retains a quality trade-off. Appendix H.4 documents the adapter, aggregation, and protocol qualifications underlying these results.

SkillZip Pro jointly compresses *content, placement, and access*. Scoped reuse reduces cost, entry audits preserve usability, and continual updates maintain both as skills evolve. Ordinary-file deployment brings these mechanisms to existing agents without a new runtime protocol.

AI USE STATEMENT

Generative AI assisted with manuscript restructuring, language editing, literature comparison, interpretation of the reported results, clarification of mathematical exposition, and LaTeX/figure-table layout. AI assistance reorganized the manuscript and visualized existing records; it did not generate experimental observations or replace the reported task evaluations. The method itself uses language models for skill evolution and task execution and, in the production configuration, for the restricted entailment checks described in Section 4.2. The distinction between deterministic structural compression and model-assisted checking is maintained in the reported costs. Responsibility for the final scientific claims, source records, and submission remains with the authors.

ETHICS STATEMENT

Compressing operational instructions can remove safety-critical exceptions even when overall text retention is high. The moderation study therefore reports false positives and protects exemption rules rather than treating aggressive reduction as an unconditional benefit. Production examples may contain sensitive content; aggregate results do not imply that underlying user data can be released. A structural audit is not a substitute for application-specific validation, privacy controls, or human oversight in consequential deployments.

REPRODUCIBILITY STATEMENT

Appendices A–C provide the representation, cost assumptions, preservation argument, and algorithms. Appendix D retains the full experimental measurements and figures; Appendices E–G specify implementation, schemas, prompts, and protocols. Appendix H.4 distinguishes completed measurements from protocol provisions and records source-level aggregation and provenance qualifications. No unreported seed results, task-level confidence intervals, or additional experiments are inferred from aggregate tables.

REFERENCES

Anthropic. Claude Code: An Agentic Coding Tool for the Terminal. https://docs.anthropic.com/en/docs/claude-code/overview, 2025. Accessed: 2026-07-25.

Xiaofan Bai, Hongqiang Lin, Chao Liu, Yantao Zhang, Xuan Jin, Xipeng Cao, and Yuhong Li. SkillZip: Evaluation-Free Skill Compression for Self-Evolving Agents by Discovering Reusable Structure, 2026. URL https://arxiv.org/abs/2608.11079.

Le Chen, Erhu Feng, Yubin Xia, and Haibo Chen. SkillRT: Compiling Skills for Efficient Execution Everywhere. *arXiv preprint arXiv:2604.03088*, 2026.

Esther Galbrun. The Minimum Description Length Principle for Pattern Mining: A Survey. *Data Mining and Knowledge Discovery*, 36(5):1679–1727, 2022.

Yudong Gao, Zongjie Li, Yuanyuan Yuan, Zimo Ji, Pingchuan Ma, and Shuai Wang. SkillReducer: Optimizing LLM Agent Skills for Token Efficiency. *arXiv preprint arXiv:2603.29919*, 2026.

Gorilla Team. BFCL V4: Agentic Part 1—Web Search. Berkeley Function Calling Leaderboard Technical Blog, 2025. URL https://gorilla.cs.berkeley.edu/blogs/15_bfcl_v4_web_search.html. Accessed: 2026-07-23.

Peter D. Grünwald. *The Minimum Description Length Principle*. MIT Press, 2007.

Linyang He, Qiyao Yu, Hanze Dong, Baohao Liao, Xinxing Xu, Micah Goldblum, Jiang Bian, and Nima Mesgarani. LiveMathematicianBench: A live benchmark for mathematician-level reasoning with proof sketches. *arXiv preprint arXiv:2604.01754*, 2026.

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*, pp. 13358–13376, 2023.

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 Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1658–1677, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.91. URL https://aclanthology.org/2024.acl-long.91/.

N. Jesper Larsson and Alistair Moffat. Off-Line Dictionary-Based Compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

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*, pp. 6342–6353, 2023.

Yuxuan Liu, Zhaochen Su, Lingyun Xie, Yuhao Zhang, Qing Zong, Jiahe Guo, Zhongwei Xie, Yiyan Ji, Yauwai Yim, Hongyu Luo, Xiyu Ren, Ruan Chenyu, Haoran Li, and Yangqiu Song. SkillRevise: Improving LLM-Authored Agent Skills via Trace-Conditioned Skill Revision. *arXiv preprint arXiv:2606.01139*, 2026.

Zeyao Ma, Bohan Zhang, Jing Zhang, Jifan Yu, Xiaokang Zhang, Xiaohan Zhang, Sijia Luo, Xi Wang, and Jie Tang. SpreadsheetBench: Towards Challenging Real World Spreadsheet Manipulation. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Moonshot AI. Kimi K2.6: Advancing Open-Source Coding. Kimi Technical Blog, 2026. URL https://www.kimi.com/blog/kimi-k2-6. Accessed: 2026-07-23.

Jesse Mu, Xiang Lisa Li, and Noah Goodman. Learning to Compress Prompts with Gist Tokens. In *Advances in Neural Information Processing Systems*, volume 36, pp. 19327–19352, 2023.

Craig G. Nevill-Manning and Ian H. Witten. Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

OpenAI. Codex: An Agentic Coding Tool for the Terminal. https://github.com/openai/codex, 2025. Accessed: 2026-07-25.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, et al. LLMLingua-2: Data Distillation for Efficient and Faithful Task-Agnostic Prompt Compression. In *Findings of the Association for Computational Linguistics: ACL 2024*, 2024.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The Berkeley Function Calling Leaderboard (BFCL): From Tool Use to Agentic Evaluation of Large Language Models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 48371–48392. PMLR, 2025.

Qwen Team. Qwen3.6-Plus: Towards Real World Agents. Qwen Research Blog, April 2026a. URL https://qwen.ai/blog?id=qwen3.6. Accessed: 2026-07-23.

Qwen Team. Qwen3.7: The Agent Frontier. Qwen Research Blog, May 2026b. URL https://qwen.ai/blog?id=qwen3.7. Introduces the Qwen3.7 series, including Qwen3.7-Max. Accessed: 2026-07-23.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Xingyu Tan, Xiaoyang Wang, Qing Liu, Xiwei Xu, Xin Yuan, Liming Zhu, and Wenjie Zhang. SkillZip: Contract-Preserving Graph Compression for Scalable Agent Skill Libraries. *arXiv preprint arXiv:2608.05604*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models. *Transactions on Machine Learning Research*, 2024.

Hanyu Wang, Yifan Lan, Bochuan Cao, Lu Lin, and Jinghui Chen. SkillGrad: Optimizing Agent Skills Like Gradient Descent. *arXiv preprint arXiv:2605.27760*, 2026.

Fangyuan Xu, Weijia Shi, and Eunsol Choi. RECOMP: Improving Retrieval-Augmented LMs with Context Compression and Selective Augmentation. In *International Conference on Learning Representations*, 2024.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive Strategy for Self-Evolving Agent Skills. *arXiv preprint arXiv:2605.23904*, 2026.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. *arXiv preprint arXiv:2510.04618*, 2025.

Tianyi Zhang and Zhonghao Qi. Skill-to-LoRA: From Using Skills to Learning Behaviors for Token-Efficient LLM Agents. *arXiv preprint arXiv:2606.16769*, 2026.

Xi Zhang, Meijun Gao, Yuntian Zhao, Xinyu Tan, Yilun Yao, Feiyu Wang, Yanshu Wang, Dingsiyi, and Tong Yang. Formal Skill: Programmable Runtime Skills for Efficient and Accurate LLM Agents. *arXiv preprint arXiv:2605.19604*, 2026.

SUPPLEMENTARY MATERIAL

Reading guide. Appendix A expands the representation and cost ledger; Appendix B gives placement analysis and conditional preservation; Appendix C supplies all three algorithms. Appendix D retains the detailed experimental measurements and original visual evidence. Implementation, schemas, and protocols are in Appendices E–G; Appendix H.4 makes the source record's measurement boundaries explicit. Main tables consolidate related results; no missing experimental outcome has been estimated to fill a cell.

A FULL PROBLEM FORMULATION AND COST LEDGER

A.1 A SKILL IS A PROGRESSIVELY LOADED BUNDLE

We model a skill as a rooted directory $B = (V, E, r)$. Each resource $v \in V$ has a canonical path p_v, bytes b_v, media type m_v, and loading class ℓ_v. The root r is normally `SKILL.md`. An edge $e = (u, v, g, s)$ records a reference from u to v, its guard g, and source span s; a guard may be unconditional or author-written, such as "for CSV export."

We distinguish four loading layers:

1. **Catalog:** the name, description, and entry metadata visible before activation;
2. **Activation:** the root instructions loaded whenever the skill is selected;
3. **Path:** the transitive resources loaded for a particular execution branch;
4. **Deployment:** all bytes distributed with the skill, including resources rarely or never read.

An edit can improve one layer while harming another. Moving a rare 500-token branch into the root may reduce **deployment** length after deduplication yet add 500 tokens to every activation. We therefore measure each layer separately.

Definition A.1 (Safe resource graph)**.** A graph is *safe* if every resolved target is a regular file inside the canonical bundle root, every recorded internal reference has an existing target, and every edge retains its source span and guard. External URLs are recorded as external edges but are never fetched. Symbolic-link escape, path traversal, missing targets, and ambiguous dynamic paths are rejected in strict mode.

The resolver is intentionally conservative. Markdown links, explicit path literals, and declared subskill entries are recognized; an unrecognized file remains in the deployment bundle but contributes no inferred dependency. This prevents the compressor from inventing loading semantics.

A.2 Entry Contracts: Which Resources Must Stand Alone

Agents usually reach resources from the root, but they may also select a nested subskill or reference directly. The compressor must know this before deleting content: text covered by the root may still be essential to a direct call. An *entry contract* η_v therefore assigns each node one of three roles:

- **Private (internal):** reached only through the root, with no standalone-use requirement.
- **Public (standalone):** directly selectable and therefore fully usable without the root.
- **Conditional:** directly usable only with a declared host context or dependency closure.

The author or catalog declares the contract; the compressor never infers it. For a public or conditional entry e, define independence as

$$\mathrm{Ind}(e, B') = \mathrm{Cov}(e, B'_{\downarrow e})\, \mathbb{I}[e \text{ discoverable}], \tag{5}$$

where Cov is the fraction of source contract units covered by the closure $B'_{\downarrow e}$ reachable from e, including any declared host context. Discoverability requires an executable entry at the advertised path. Thus $\mathrm{Ind} = 1$ means that the entry remains independently usable; missing content lowers coverage, while a renamed or hidden entry has independence zero.

A.3 Typed Resource Contracts

For each textual node v, SkillZip Pro extracts a contract

$$\mathcal{C}_v = \langle I_v, W_v, T_v, R_v, O_v, E_v, P_v, L_v \rangle, \tag{6}$$

where I contains applicability and interface conditions; W workflow states and ordering edges; T tool or resource requirements; R rules and prohibitions; O output fields and formats; E evidence or verification obligations; P provenance to source spans; and L locked residuals that must remain verbatim. Each unit $a \in \mathcal{C}_v$ has a semantic type, normalized payload, scope, guard, and provenance.

Executable code, structured data, images, and other non-instructional artifacts are *locked nodes*. The harness-neutral compiler may change neither their path nor their bytes. Their inbound references can be rewritten only when the referring Markdown is rewritten and the canonical target remains identical.

For a candidate bundle B', let $\mathrm{cover}(a, B')$ mean that a compatible statement, reference, or locked artifact in B' entails unit a on every path where it originally applied. Coverage is type constrained: an example cannot cover a prohibition, generic advice cannot cover a required output field, and an unrelated branch cannot cover a guarded rule.

Definition A.2 (Bundle faithfulness)**.** B' is faithful to B under environment contract H if (i) every source unit is covered at a compatible scope, (ii) every locked node is byte-identical, (iii) all internal references in B' resolve safely, and (iv) any unit removed by host entailment has an exact typed witness in H bound to the audited environment digest.

This structural guarantee does not assert behavioral equivalence for every language model. It protects the bundle's explicit contract, not every latent reading.

A.4 Four Costs, One Constrained Objective

Let $\tau(x)$ be the deployment tokenizer or another declared length function. For a bundle B', define

$$C_{\mathrm{cat}}(B') = \tau(\text{name, description, entry}), \tag{7}$$

$$C_{\mathrm{act}}(B') = \tau(b'_r), \tag{8}$$

$$C_{\mathrm{dep}}(B') = \sum_{v \in V'} \tau(b'_v), \tag{9}$$

$$C_{\mathrm{path}}(B'; \pi) = \sum_{v \in \mathrm{load}(B', \pi)} \tau(b'_v). \tag{10}$$

For the additive ledger, load excludes the initial root activation already charged by C_{act}; later rereads are separate charged events. The displayed mean-run measurements already include activation and must not have the root column added again. Here π denotes an execution path or task class and load follows the unchanged agent's progressive-loading behavior. Given an empirical path distribution $q(\pi)$, our primary optimization target is

$$J(B') = C_{\text{cat}}(B') + C_{\text{act}}(B') + \mathbb{E}_{\pi \sim q} C_{\text{path}}(B'; \pi) + \lambda C_{\text{dep}}(B'), \tag{11}$$

subject to bundle faithfulness. We use $\lambda = 0.05$ as a transparent default so that storage matters without dominating runtime exposure. If no trace distribution is available, explicit branch guards induce a uniform distribution over reachable leaf paths; every reported result must identify which estimator was used.

We report all four costs even when optimizing Eq. (2). A single "compression ratio" is insufficient: it can conceal a regression in the always-loaded root or in the tail of **path** cost. For metric x, the reduction is $1 - C_x(B')/C_x(B)$; negative values are preserved rather than clipped.

A.5 Two Deployment Lifecycles and Their Costs

The four costs above describe one artifact, but deployment has two distinct lifecycles. **Persistent** compression rewrites the canonical bundle; its one-time work reduces storage and future runs. **Transient** compression leaves the canonical bundle unchanged and builds a task-specific execution view $\widehat{B}_{e,\pi}$ before a run through entry e. Its benefit applies only to that run and must be reported after build and cache cost.

Because the two change different things, we separate seven quantities and never average across them:

1. **Canonical storage** $C_{\text{disk}}(B')$: bytes of the on-disk bundle. Persistent shrinks it; transient keeps it equal to the source.
2. **Shipped/deployment** $C_{\text{dep}}(B')$ (Eq. (2)): distributed content under the declared length function.
3. **Per-entrypoint activation** $C_{\text{act}}(B'; e)$: entry tokens loaded when entry e is selected.
4. **Per-run execution-view tokens** $C_{\text{view}}(\widehat{B}_{e,\pi})$: context loaded for one run through e on task π.
5. **Transient build latency** $\Lambda_{\text{build}}(e, \pi)$: wall-clock time and any model calls to construct $\widehat{B}_{e,\pi}$.
6. **Cache and invalidation** C_{cache}: storing views keyed by (bundle digest, entry, environment version) and rebuilding them when any key changes.
7. **Public-entrypoint independence** $\text{Ind}(e, B') = 1$ for every public e: a mandatory deployment constraint.

Persistent results report quantities 1–3 and per-run load. Transient results report quantities 4–6, including build overhead, but never a disk ratio because disk is unchanged. Both lifecycles must satisfy constraint 7 for every public entry. They use the same kernel but differ in deletion scope, publication, auditing, and fallback (Section C.2).

Units and measurement boundaries. Token and byte ledgers are distinct. Text uses the declared tokenizer; non-text artifacts are locked and measured in bytes, with any contribution to a token objective requiring an explicit callback. External audit state is compressor storage, not runtime context. Provider billing, generated outputs, repeated processing of conversation history, and prefix-cache discounts require a separate measured ledger. The static objective is a placement proxy, not a universal price formula. Under a unique-load structural estimate a resource contributes once per path; trace accounting retains repeated load events. Public-entry constraints apply in addition to the root-mediated objective.

B EXECUTION-AWARE COMPRESSION THEORY

B.1 FROM REPETITION TO SCOPED REUSE

Within one document, SKILLZIP selects a shortest cover of typed contract units using primitive statements, shared rules, parameterized procedures, and explicit exceptions. For candidate representation z, let $d(z)$ be its token cost and $\Gamma(z)$ the units it covers. The file-level problem is a weighted set cover with hard coverage:

$$\min_{Z \subseteq \mathcal{Z}} \sum_{z \in Z} d(z) \quad \text{s.t.} \quad \bigcup_{z \in Z} \Gamma(z) \supseteq \mathcal{C}_v. \tag{12}$$

SKILLZIP PRO retains this optimizer but changes candidate cost according to where the representation is placed in the bundle.

Suppose exact fragment x of length d_x occurs in files $S_x \subseteq V$. Keeping all copies incurs $\sum_{v \in S_x} w_v d_x$, where w_v is the effective load weight induced by Eq. (2). Factoring x into shared module h with reference cost d_{ref} costs

$$w_h d_x + \sum_{v \in S_x} w_v d_{\text{ref}} + d_{\text{audit}}, \tag{13}$$

where w_h is the probability-weighted scope of h, and d_{audit} covers import instructions and provenance. Factoring is allowed only if Eq. (13) is smaller.

Proposition B.1 (No sparse-root promotion)**.** *Let x occur only in branches with total access probability $p < 1$. Placing x in the always-loaded root adds $(1-p)d_x$ expected tokens relative to keeping one copy within the affected **activation scope**, before reference overhead. More precisely, let D be the deployment-token saving of root placement relative to scoped placement and let Δ_{ref} be its signed change in expected reference/dispatcher cost. Root placement is worse when $(1-p)d_x + \Delta_{\text{ref}} > \lambda D$.*

Proof. In Eq. (2), root placement gives load weight one, whereas branch-scoped placement gives weight p. Their path-cost difference is $(1-p)d_x$; deployment can offset it only through the λC_{dep} term. The full difference is $(1-p)d_x + \Delta_{\text{ref}} - \lambda D$. Reference overhead need not change in the same direction for both layouts, so it is kept signed rather than assumed nonnegative. This gives the stated condition. □

The proposition captures a common failure: global deduplication can reduce storage while making common requests more expensive. SKILLZIP PRO instead places a shared module within the activation scope of the branches that need it. The root imports it only when every reachable path requires it.

B.2 CONDITIONAL CAPSULES

A guarded section with an explicit trigger can be moved from an always-loaded file into an on-demand capsule. Let the section body cost d_b, dispatcher cost d_g, and trigger probability p_g. Keeping the body inline costs d_b per load; a capsule costs $d_g + p_g d_b$ in path expectation, plus λd_g deployment overhead.

Proposition B.2 (Capsule threshold)**.** *Moving a guarded section to a capsule decreases Eq. (2) if*

$$(1-p_g)d_b > (1+\lambda)d_g. \tag{14}$$

The condition favors long, infrequent branches with short and unambiguous dispatchers. SKILLZIP PRO never infers a new guard: a capsule candidate must originate from an explicit heading or conditional clause, and the dispatcher must preserve the original trigger and relative path.

Capsules and sharing address different costs. A capsule delays an infrequent branch; a shared module removes exact repetition across branches. When both apply, the shared module remains within the union of those branches rather than moving to the root.

B.3 Host Entailment Under a Closed Contract

Some skill text restates guarantees already enforced by the deployment environment: an available tool, an immutable output schema, or a mandatory safety policy. Removing such text can yield a high activation reduction, but only if the guarantee is explicit and stable. We represent the environment as a signed typed contract

$$H = \{(t, k, v, \sigma, \delta)\}, \tag{15}$$

where t is unit type, k a canonical key, v the exact value, σ its scope, and δ the environment digest. Removal requires an exact type/key/value/scope match; semantic similarity only flags manual review.

Assumption B.3 (Environment stability)**.** The audited environment digest remains unchanged between compression and deployment. If it changes, the bundle is re-audited or the entailment transformation is disabled.

Absent a supplied contract, host entailment is a no-op. This default avoids treating model knowledge, tool documentation, or informal conventions as guarantees.

B.4 Interface Contracts and the Witness Hierarchy

Typed coverage and host entailment do not fully protect two structures that matter in deployment: interface contracts and evidence for deletion.

Definition B.4 (Interface contract)**.** An *interface contract* is a maximal source section that specifies the skill's inputs or outputs as a machine-checked format: an output schema with worked examples, a whitelist of legal labels, or a field-level validation rule. Interface contracts are *atomic*: the specification is read as one unit by the consuming model, so its coverage obligation is discharged only when the whole section is emitted contiguously and verbatim. A rewriting that preserves every line but scatters the section across several synthetic headings does *not* cover the unit.

The second gap concerns *how* a removal is justified. Coverage by itself is a claim; a witness is the evidence attached to it. Every removal SkillZip Pro commits carries exactly one, recorded in the audit state:

$$W_1 \succ W_2 \succ W_3, \tag{16}$$

where W_1 is *literal containment*: the removed text survives byte-for-byte at another reachable location, so the removal is reversible by construction; W_2 is a *cover witness*: a merged or shared representation passes the deterministic content-word coverage gate while retaining every protected literal and every negation polarity; and W_3 is an *entailment witness*: a frozen checker model (temperature 0, fixed prompt) certifies that the decision-relevant content of the removed span—conditions, verdicts, thresholds, whitelists, exemptions—is already fully expressed elsewhere. W_3 verdicts are accepted only for evidence-class units that carry no prohibition, output, or exemption marker, and each accepted verdict is logged with the checker's stated basis, so every W_3 removal is individually auditable after publication. Any deletion that cannot attach a witness is refused, whatever the predicted saving. Two further rules keep W_3 honest when it is applied inside a single document: the entailment base excludes the candidate segment itself, and two candidates may not justify each other's removal—a deletion whose only witness is another deletion is vetoed, because each verdict may have relied on text that is itself about to disappear. The implemented overlapping-candidate veto is a conservative heuristic against this failure, not a general proof of semantic independence. Where placement costs differ by layer, W_3 should additionally be spent where the executor charges most: a removal from the always-loaded root saves its weight once per reasoning round, while a removal from an on-demand reference saves it about once.

These witness classes are auditable evidence, not an ordering of unconditional formal guarantees. The hierarchy makes the intended safety boundary testable: *witness strength, not deletion count, sets the safe compression ceiling*. Section 5.5 compares the same bundle under W_1, W_2 alone and with restricted W_3 witnesses.

B.5 Bundle Faithfulness Invariant

Let Φ be the ordered transformation sequence: resolve, extract, drop entailed units, factor exact structure, form capsules, apply file-level shortest cover, materialize, and audit.

Proposition B.5 (Conditional bundle preservation)**.** *Under a safe source graph, complete extraction with exact provenance, Assumption B.3, interface-contract atomicity, and a sound final auditor, a committed output $B' = \Phi(B)$ is faithful according to Definition A.2 and can be executed by any harness that already supports the original bundle's ordinary relative-file loading semantics.*

Proof sketch. Host deletion retains a digest-bound witness. Scoped sharing replaces covered units with a reachable reference in the same activation scope. Capsule extraction preserves the explicit guard in a dispatcher, and file-level compression obeys Eq. (12). Interface contracts remain contiguous, while locked nodes are copied byte-for-byte. The disk audit rejects uncovered units, unsafe or dangling paths, scope expansion, scattered contracts, and changed locked bytes. Every committed candidate thus meets Definition A.2 and runs through the source model's ordinary files and relative paths, with no added runtime mechanism. □

As with any static natural-language verifier, the guarantee depends on the extractor and auditor. We reduce this trusted surface by retaining source spans, preferring exact transformations, auditing emitted files rather than an in-memory plan, and falling back to a byte-identical copy on uncertainty.

B.6 Continual Compression

For an incoming evolution patch Δ_t, Continual Bundle Compression identifies an affected closure A_t: changed nodes, reference ancestors, newly or formerly referenced descendants, shared modules whose support set intersects the patch, capsules whose guards changed, and host witnesses invalidated by a new environment digest. Only this closure is re-extracted and locally re-optimized; unchanged contracts are reused by source digest. The graph-update cost is

$$O(|A_t| + |E(A_t)|) + \mathrm{LM}(T_t), \tag{17}$$

where $T_t \subseteq A_t$ are changed textual nodes requiring extraction. A global audit remains $O(|V_t| + |E_t|)$ in the conservative implementation, but uses hashing and cached contracts rather than new language-model calls.

Local decisions may drift from the one-shot reference as support sets and path frequencies evolve. Define relative continual drift at checkpoint t as

$$R_t = \frac{J(B_t^{\mathrm{cont}}) - J(B_t^{\mathrm{one}})}{\max(1, J(B_t^{\mathrm{one}}))}, \tag{18}$$

where B_t^{one} is a full one-shot recompression of the same verbatim authored state. A triggered global repack resets placement debt. The trigger is conservative: recoverable saving above θ_{repack}, relative bundle growth above ρ, workload drift above δ_W, an environment-digest change, or a maximum of K patches. This gives a tunable continuum between full recompression after every write and cheap local updates.

Faithfulness does not depend on the schedule: both modes materialize and audit a complete directory. If continual optimization fails, the system publishes the verbatim patched bundle, not the previous compressed version. Compression may lose savings, but it cannot discard the patch.

What the preservation result does not prove. The theorem is conditional on the soundness of coverage judgments and on the completeness of the extracted contract. A deterministic lexical coverage gate or a model-based entailment verdict can violate these assumptions. The theorem therefore does not turn W_2 or W_3 into a proof of behavioral equivalence. Nor does it imply that a model will execute a reachable route correctly. Empirical line retention, routing retention, task quality, and public-entry startup test different consequences of compression.

C Compiler Details and Complete Algorithms

Figure 2 summarizes the compiler. Like SkillZip, it is evaluation-free and seeks the shortest faithful representation. The difference is scope: SkillZip compiles one `SKILL.md`, whereas SkillZip Pro compiles a progressively loaded directory as a typed resource graph. It supports both full One-Shot optimization and state-reusing Continual updates.

The compiler removes linked-file content already supplied by the root or environment, then verifies every route. One-Shot or Continual determines when optimization runs; Persistent or Transient determines whether output replaces the canonical bundle or forms a per-run view.

C.1 The Two Pillars

The extension has two goals. **Pillar 1 (compress across files)** chooses what to remove and where to place what remains. **Pillar 2 (preserve routing)** ensures that these rewrites do not sever the links used for progressive loading. The first creates savings; the second makes them safe to deploy.

Pillar 1 — activation-aware cross-file compression. Three graph-level transformations, none available to a single-document compressor, act on top of the per-file optimizer. (i) *Host-entailment pruning*: a unit whose obligation is exactly implied by the root or by a declared environment contract is removed, with a digest witness recorded, because the agent already receives it. (ii) *Activation-scoped sharing*: text repeated across branches is factored into one shared module, but placed *only within the activation scope that loads it*—never promoted into the always-loaded root, which would raise per-run cost. (iii) *Conditional capsules*: a long guarded branch moves into an on-demand file, leaving a one-line dispatcher. Together, these transformations remove redundancy between the root and dynamically loaded files.

Pillar 2 — cross-file routing preservation. The same rewriting is dangerous precisely because a routing line—"when the task matches X, read `[X](refs/x.md)`"—looks like ordinary prose but is the skill's **navigation table**. SkillZip Pro protects it with three layers of defense. (i) The routing table is locked as a first-class unit, so the optimizer may neither reword nor merge it. (ii) Every reference-bearing source line is preserved through rendering, so a link cannot be dropped as a side effect of compressing the text around it. (iii) Before publishing, an independent pass re-reads the materialized directory and verifies that *every* file—including shared modules and capsules the compressor itself created—is still **reachable** from the root by following resolved links; if any branch became unreachable, the candidate is rejected and the verbatim bundle is shipped instead. Section 5 measures the result: SkillZip Pro keeps 1.000 routing fidelity while root-only and flat compressors drop to 0.000.

C.2 Two Deployment Lifecycles for the Kernel Output

Together, the pillars form a compression kernel that rewrites files while preserving their routes. A separate lifecycle choice determines where the result lives. This choice is independent of the One-Shot/Continual schedule in Sections C.5–C.6: the schedule controls when the kernel runs, while the lifecycle controls whether its output replaces the shipped bundle.

Persistent Bundle Compression runs the kernel on the canonical bundle and ships the smaller bundle in its place. It suits private references and subskills that are only ever reached from the root: their repeated text is covered elsewhere, so it can be removed once and for all, cutting both the shipped size and every future run that would have loaded it. Because the shipped files change, the kernel may touch a public entry only after a *multi-entry audit* re-reads the published directory from *every* declared public entry and confirms that each still routes correctly and keeps $\mathrm{Ind} = 1$ on its own; if any check fails, the verbatim bundle is shipped. This is the lifecycle of every earlier section, and we label it as such from here on.

Transient Execution-View Compression leaves the canonical bundle byte-for-byte unchanged and builds a throwaway *execution view* $\widehat{B}_{e,\pi}$ before a run. For selected entry e, the view combines the closures of the root, e, and their shared dependencies, compresses them with the same kernel, and re-roots the result at e. The original root remains linked as `_host_context.md`, so host obligations survive. The view is discarded or cached by (bundle digest, e, environment digest) until a key changes. A view that additionally specializes on task content must include that conditioning in its cache key. It lowers run context, not disk use. Algorithm 1 summarizes the transient compression. If compression does not improve the view or its single-entry audit fails, the system loads the uncompressed closure.

Because *when* to compress and *whether to keep the result* are independent, they combine into four production modes (Section 4.3). One-shot persistent is the initial migration of an evolved library into

Algorithm 1 Transient Execution-View Construction

Require: canonical bundle B with root r, chosen entry e, optional environment H, cache flag
Ensure: execution view rooted at e; canonical B unchanged

$k \leftarrow \textsc{ViewKey}(\mathrm{digest}(B), e, \mathrm{digest}(H))$
if cache **and** $\textsc{Hit}(k)$ **then**
  return $\textsc{LoadCached}(k)$ ▷ no kernel call
$\Omega \leftarrow \textsc{Reach}(B, e) \cup \textsc{Reach}(B, r)$ ▷ entry and root closures
$R \leftarrow \textsc{MaterializeClosure}(B, \Omega, e)$ ▷ promote e to root; link r as host context
$\widehat{B} \leftarrow \textsc{Kernel}(R)$ ▷ scoped rewrites and entry checks
if $\widehat{B}$ unusable **or** $\tau(\widehat{B}) \geq \tau(R)$ **or** single-entry audit fails **then**
  $\widehat{B} \leftarrow R$ ▷ safe fallback: uncompressed closure
return $\textsc{Cache}(k, \widehat{B})$ ▷ verify $\mathrm{digest}(B)$ unchanged

a shipped bundle; continual persistent keeps that bundle small as it is edited and re-shipped; one-shot transient serves a stable bundle whose entries are called directly and rarely; and continual transient serves high-frequency direct calls against a bundle that keeps changing, where digest-keyed caching amortizes the build over many runs.

C.3 Shared Inputs and Persistent Compiler State

The required input is a bundle root and its entry file. Optional inputs are a deployment tokenizer or cost callback, a workload ledger of task-class frequencies and observed resource loads, and a typed, digest-bound environment contract. Without traces, explicit branch structure estimates path weights. Without an environment contract, no host-entailment deletion is attempted. Both modes share a persistent compressor state

$$M_t = \langle G_t, \mathcal{C}_t, W_t, H_t, \mathcal{I}_t, \mathcal{P}_t, \mathcal{A}_t \rangle, \tag{19}$$

where G_t is the safe resource graph; $\mathcal{C}_t$ the per-node contract store; W_t the path-weight ledger; H_t the environment digest and witnesses; $\mathcal{I}_t$ candidate, support, and reverse-reference indices; $\mathcal{P}_t$ provenance; and $\mathcal{A}_t$ prior audit results. The sidecar accelerates future compression but is never required at execution. If it is absent or its digest is stale, the tool safely falls back to one-shot reconstruction.

C.4 Shared Bundle Compiler

Safe graph resolution. The scanner walks the directory without following escaping symbolic links and classifies each node as instructional Markdown, subskill entry, code, structured data, or opaque asset. It recognizes local Markdown links, explicit path literals with known extensions, and declarative subskill references. Every internal edge records its exact source span and nearest explicit guard. External URLs are preserved but never fetched; missing, ambiguous, or escaping local targets fail strict mode. Unreferenced files remain in deployment cost and are never silently discarded.

Typed contract extraction. Each instructional node is segmented by headings and source spans. One structured extraction recovers Eq. (6), while every unit retains provenance. Relations such as `same_rule`, `same_workflow`, and `exception_of` are accepted only after deterministic type, polarity, guard, and payload checks. Low-confidence units are locked and preserved verbatim. The original SkillZip file-level scanner, type-compatible reuse, minimum-cost cover, fixed-template render, and span-restoration audit operate here unchanged for each eligible file.

Bundle-level candidate generation. Three transformations expose savings unavailable inside a single file. (1) Exact contract units entailed by H_t may be removed only with a scope-compatible, digest-bound witness. (2) Repeated exact rules or workflow fragments may move to `.skillzip_shared/<digest>.md` at their lowest safe activation scope; affected branches receive a mandatory relative loading instruction. (3) A long section with an explicit guard may move to `capsules/<slug>.md`; a dispatcher retaining the trigger remains at the original site. Each transformation is evaluated together with per-file covers under Eq. (2).

Global selection and materialization. Because a capsule changes path weights and a shared module may serve several capsules, marginal selection is followed by local add/drop/swap improvement. Every move preserves modeled coverage. Materialization writes an ordinary relative-path directory; code, data, schemas, and assets remain byte-identical. Interface contracts (Definition B.4) are restored as one contiguous verbatim span: surrounding prose may shrink, but the optimizer cannot split, reorder, or paraphrase the contract. A post-pass removes scattered render fragments and restores the source span. Every removal records its witness class from Eq. (16); W_3 removals also store the verdict and rationale.

C.5 Mode I: One-Shot Bundle Compression

One-shot mode is intended for initial migration, release packaging, a missing or invalid state sidecar, or a periodic global re-optimization after workload drift. It reconstructs M_t from the complete source bundle and searches globally. Algorithm 2 never publishes a candidate whose materialized objective is worse than the source.

Algorithm 2 SkillZip Pro: One-Shot Bundle Compression

Require: source bundle B, entry r, optional H, Q
Ensure: audited bundle $B^\star$ and state M, or verbatim B

$G \leftarrow \textsc{ResolveSafeGraph}(B, r)$
$\mathcal{C} \leftarrow \textsc{ExtractTypedContracts}(G)$
$W \leftarrow \textsc{EstimateLoadWeights}(G, Q)$
$Z \leftarrow \textsc{FileCoverCandidates}(\mathcal{C})$
$Z \leftarrow Z \cup \textsc{HostEntailments}(\mathcal{C}, H)$
$Z \leftarrow Z \cup \textsc{ScopedSharing}(G, \mathcal{C}, W)$
$Z \leftarrow Z \cup \textsc{GuardedCapsules}(G, \mathcal{C}, W)$
$P \leftarrow \textsc{ConstrainedSelect}(Z, J, \text{cover})$
$D \leftarrow \textsc{MaterializeTemporary}(B, P)$
if $J(D) \le J(B)$ **and** $\textsc{AuditFromDisk}(B, D, H)$ **then**
    return $\textsc{AtomicCommit}(D), \textsc{SaveState}(D)$
else
    return $\textsc{VerbatimCopy}(B), \textsc{FailureState}(B)$

C.6 Mode II: Continual Bundle Compression

Continual mode targets self-evolving skills that receive frequent small patches. Let Δ_t contain added, modified, moved, and deleted paths. The tool first applies Δ_t without paraphrase to the current authored source, producing the semantic fallback B_t^{raw}. It then updates reference edges and constructs an invalidation closure A_t containing changed nodes, their reference ancestors, newly or formerly referenced descendants, shared modules whose support changed, capsules whose guard changed, and any host witness invalidated by an environment digest change.

Only changed textual nodes are re-extracted; digest-identical contracts are reused. Patch units retain the four operations of SkillZip: Absorb removes an exact duplicate already covered; Refine updates an existing rule, guard, argument, or exception; Extend adds genuinely new behavior or a new resource; and Refactor changes a representation when several edits make another cover shorter. Pro extends Refactor to placement: affected shared modules, capsules, and host witnesses are re-priced whenever their support set or path weight changes.

Local repair cannot accumulate unbounded debt. A one-shot global repack is triggered when recoverable objective saving exceeds θ_{repack}, bundle growth exceeds ρ, the workload distribution drifts beyond δ_W, the environment digest changes, or K patches have elapsed. Repacking reads the current bundle and compact state, never the full patch history.

The fallback distinction is essential. A failed one-shot run may return its input. A failed continual run cannot reactivate B_{t-1} after a valid patch, which would discard learned behavior; it publishes B_t^{raw} and records the failure for retry.

Algorithm 3 SkillZip Pro: Continual Bundle Compression

Require: authored source S_{t-1}, audited state M_{t-1}, patch Δ_t
Ensure: current faithful bundle B_t and state M_t

$S_t, B_t^{\text{raw}} \leftarrow \textsc{ApplyPatchVerbatim}(S_{t-1}, \Delta_t)$
$G_t, A_t \leftarrow \textsc{UpdateGraphAndClosure}(M_{t-1}.G, S_t, \Delta_t)$
$\mathcal{C}_t \leftarrow \textsc{ReuseAndReextract}(M_{t-1}.\mathcal{C}, A_t)$
$U_t \leftarrow \textsc{Classify}(\textsc{Absorb}, \textsc{Refine}, \textsc{Extend}, \textsc{Refactor})$
$Z_t \leftarrow \textsc{RefreshAffectedCandidates}(A_t, U_t, M_{t-1})$
if $\textsc{RepackTriggered}(M_{t-1}, S_t, Z_t)$ **then**
    return $\textsc{OneShot}(S_t)$
$P_t \leftarrow \textsc{RepairLocalCoverAndPlacement}(Z_t, J)$
$D_t \leftarrow \textsc{MaterializeTemporary}(B_t^{\text{raw}}, P_t)$
if $J(D_t) \leq J(B_t^{\text{raw}})$ **and** $\textsc{AuditFromDisk}(S_t, D_t, H_t)$ **then**
    return $\textsc{AtomicCommit}(D_t), \textsc{SaveState}(D_t)$
else
    return $\textsc{AtomicCommit}(B_t^{\text{raw}}), \textsc{RebuildState}(S_t)$

C.7 Cross-File Audit and Mode Selection

The auditor ignores in-memory coverage claims and rereads the emitted directory. It checks (i) graph closure and path confinement; (ii) typed coverage at a compatible scope or by an exact host witness; (iii) mandatory reachability and guard preservation for every capsule and shared module; (iv) path and SHA-256 identity for locked artifacts; (v) that every interface contract appears as one contiguous span whose normalized lines equal the source section, with no scattered fragments elsewhere; and (vi) all four materialized costs. Atomic publication occurs only after audit succeeds.

C.8 Harness-Agnostic Deployability

Both modes remain agent harness-agnostic. The compressor may persist state, observe offline traces, and run on every write, but the published bundle does not depend on it. The agent selects the same skill, loads the same root entry, follows ordinary relative links, and executes byte-identical artifacts.

D Complete Experimental Evidence

Table 1 joins task success, cost, and access in the main paper. Tables 7 and 8 below retain the full paired quality tests and the two distinct library regimes. This appendix supplies the complementary absolute costs, retention diagnostics, execution traces, workload sweeps, entry-level results, and visualizations. A superscript a consistently denotes the compression-only SkillReducer adaptation; † or a cross identifies a structurally damaged control, not a valid compression winner. All source measurements are retained, including negative savings and verbatim fallbacks.

D.1 Paired quality and the two bundle regimes

D.2 Absolute costs, actual loading, and retained content

The four cost measures in Table 9 are not interchangeable. Human Skill has a small shipped artifact but a longer always-loaded root than Evolved Bundle. Expert Progressive shortens deployment and the mean run while expanding activation. Pro reduces every displayed cost measure, including the worst-path column by 10.8%, without the route loss of the root-only or flat controls. The token counts and reduction percentages are aggregated quantities, so recomputing each percentage from the rounded means can yield a different number.

Required-resource recall asks whether the executor opens resources identified as necessary for its task. Irrelevant load counts resources opened unnecessarily; dispatch records branch selection. The source data give Pro .755 required recall and .193 irrelevant load, closer to the uncompressed .795/.171 than the other reported compressors. The increase from 6.3 to 10.3 reachable files reflects generated shared modules and capsules. It does not mean all those files enter every execution context.

Table 7: Task quality and paired testing. Qwen3.6-Plus on grown bundles; best compression macro score in bold. Macro weights benchmarks equally; pooled scores use all 102 tasks. CI is the paired 95% interval versus Evolved; NI uses the reported -0.05 margin.

	Benchmark task success ↑				Paired quality assessment		
Method	BFCL	Math	Sheet	Macro	Pooled	Difference [95% CI]	NI
No Skill	.809	.364	.312	.495	.431	$-.039$ $[-.108, +.029]$	✗
Human Skill	.905	.455	.333	.564	.490	$+.020$ $[-.039, +.078]$	✓
Evolved Bundle	.905	.364	.354	.541	—	reference	—
Root-only†	.905	.333	.271	.503	.422	$-.049$ $[-.128, +.020]$	✗
Flat-concat†	.857	.394	.167	.473	.382	$-.088$ $[-.167, -.010]$	✗
SkillReducer[a]	.857	.394	.271	.507	.431	$-.039$ $[-.108, +.029]$	✗
Expert Progressive	.905	.333	.312	.517	.441	$-.029$ $[-.088, +.029]$	✗
SKILLZIP PRO	**.952**	.394	.333	**.560**	**.480**	$+.010$ $[-.029, +.059]$	✓

Table 8: Cost and fidelity across two regimes. Ship/Root/Run are percentage reductions; Kept is reachable source-line retention and Route is declared-route retention. Real means include Pro's fallback (2/3 published). [a]Compression-only adaptation; †routing-damaging control.

	Grown task-suite bundles					Real evolved libraries				
Method	Ship	Root	Run	Kept	Route	Ship	Root	Run	Kept	Route
Evolved	0.0	0.0	0.0	1.000	1.000	0.0	0.0	0.0	1.000	1.000
Root-only†	2.2	27.8	10.3	.939	.000	1.6	69.2	12.5	.951	.000
Flat-concat†	63.8	42.9	68.9	.241	.000	64.4	82.4	90.2	.002	.000
SkillReducer[a]	5.0	-0.4	4.1	1.000	1.000	6.3	63.0	30.9	.954	.125
Expert Prog.	13.7	-6.5	17.9	1.000	1.000	25.0	-2.9	26.0	1.000	1.000
SKILLZIP PRO	**19.7**	7.3	14.3	.986	**1.000**	23.1	1.7	17.7	**1.000**	**1.000**

Table 9: Absolute cost and reduction by loading layer. Each cell gives tokens/reduction relative to Evolved; reductions are macro-aggregated and need not equal ratios of displayed mean token counts. Human Skill is a reference condition, not a compression of Evolved. The source table reports worst-path cost, distinct from the P95 label in Figure 7.

Method	Shipped ↑	Root ↑	Mean run ↑	Worst run ↑
Human Skill	328/+87.7%	328/-51.4%	328/+43.3%	328/+54.3%
Evolved Bundle	2649/0%	212/0%	588/0%	755/0%
Root-only†	2589/+2.2%	153/+27.8%	529/+10.3%	696/+8.2%
Flat-concat†	954/+63.8%	120/+42.9%	183/+68.9%	249/+67.3%
SkillReducer[a]	2518/+5.0%	213/-0.4%	564/+4.1%	722/+4.4%
Expert Progressive	2285/+13.7%	226/-6.5%	482/+17.9%	754/+0.2%
SKILLZIP PRO	2112/+19.7%	196/+7.3%	500/+14.3%	669/+10.8%

Table 10: Executed loading on grown bundles. Required recall and dispatch measure executor behavior; route retention in Table 8 is a separate static property. Missing navigation can produce no dangling-link error, so zero link errors alone does not establish faithful access.

Method	Recall ↑	Irrel. ↓	Dispatch ↑	Links ↓	Fallback ↓	Files
Evolved Bundle	0.795	0.171	0.795	0	–	6.3
Root-only†	0.685	0.290	0.685	0	0.000	6.3
Flat-concat†	0.708	0.259	0.708	0	0.000	6.0
SkillReducer[a]	0.688	0.288	0.688	0	0.000	6.3
SKILLZIP PRO	0.755	0.193	0.755	**0**	0.000	10.3

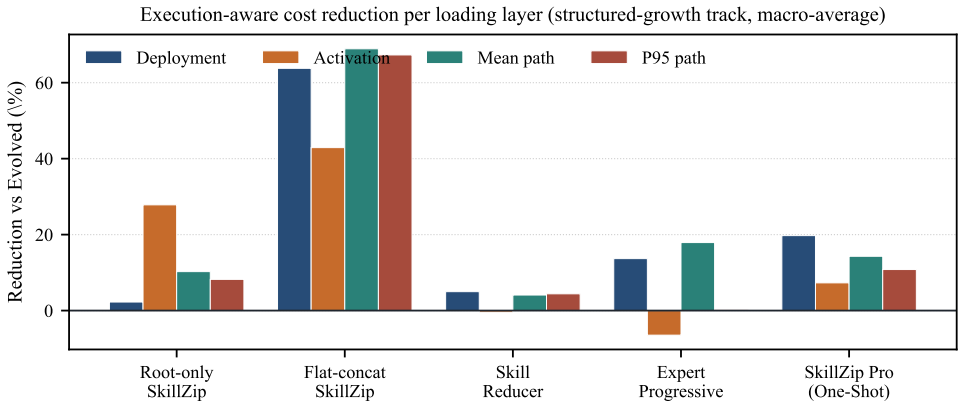

Figure 7: Per-layer cost visualization on grown bundles. Negative bars denote regressions. The figure's tail series is labeled P95 in the supplied asset; Table 9 reports a worst-path statistic. These labels are retained separately rather than silently treated as the same measurement.

Static routing is different. A removed routing line need not produce a broken link: the edge can simply disappear. This explains why the root-only and flat controls have zero link errors but zero retained routing pairs. The audit must therefore compare the original and emitted navigation, not only test whether surviving links resolve.

> **Access is a separate invariant.** Preserving bytes on disk does not preserve a skill if the agent can no longer discover the required branch. Cost, content retention, declared routing, and actual loading must all remain visible.

Table 11: Additional retention measurements on grown bundles. Kept is reachable source-line retention; lost lines are the source record's unwitnessed-loss count. Repetition removed and shipped savings are not substitutes for coverage. Environment-witnessed removals are tracked separately.

Method	Kept ↑	Lost lines ↓	Repetition removed ↑	Ship saving ↑	Route ↑
Evolved Bundle	1.000	0.0	0.000	+0.0%	1.000
Root-only†	0.939	5.3	0.002	+2.2%	0.000
Flat-concat†	0.241	59.0	1.000	+63.8%	0.000
SkillReducer[a]	1.000	0.0	0.073	+5.0%	1.000
Expert Progressive	1.000	0.0	0.482	+13.7%	1.000
SkillZip Pro	0.986	1.0	0.736	+19.7%	1.000

Table 11 separates genuine reuse from text disappearance. Pro removes .736 of measured repetition with .986 line retention, whereas Flat-concat removes all measured repetition but retains only .241 of source lines. These are operational line-matching diagnostics, not a formal semantic-equivalence score. In particular, the reported 1.0 lost-line mean is not silently relabeled as complete retention. Source removals supported by environment witnesses are recorded separately.

D.3 Worked example: keep the branch, share the repeated rules

The source example is `round_02` from the recorded `qwen3.6-plus` math library. Its 15 strategy branches repeat several output and precision rules, with one rule occurring at 16 bundle locations. Pro reduces the selected branch from 527 to 160 tokens while retaining all 15 strategy branches. Flat-concat retains only `round_00` and `round_10`; the selected branch cannot be reached. Figure 10 preserves the original example's text, highlights, counts, and branch comparison in a more readable two-column arrangement.

A shared module contains the repeated precision and formatting rules once and supplies explicit read references to its consumers. The 315-token module has 16 source occurrences; replacing the other 15 copies avoids $15 \times 315 = 4{,}725$ repeated body tokens before accounting for the new references. The

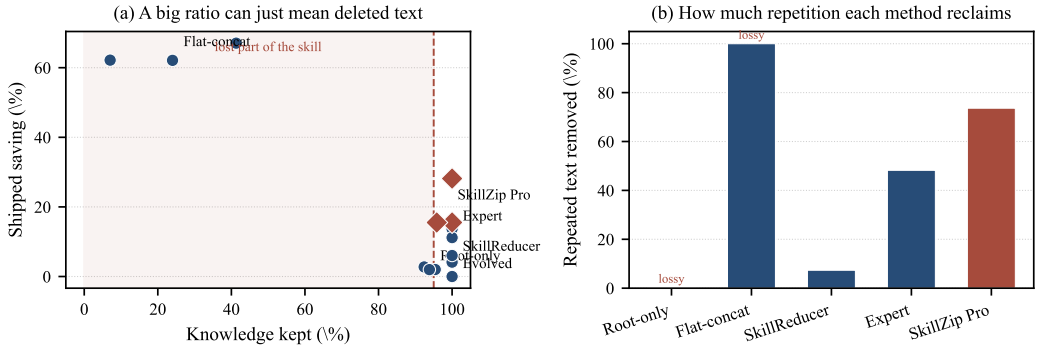

Figure 8: Cost–retention and repetition-reuse plots. The source labels visually distinguish methods losing content; Table 11 gives the exact line-level measurements. A large saving alone cannot distinguish compression from deletion.

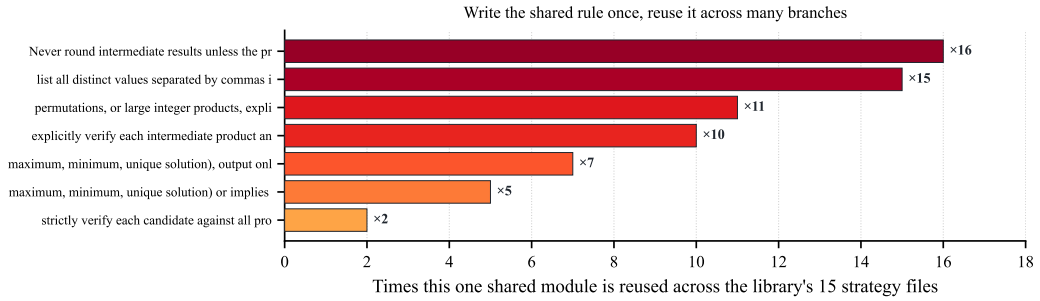

Figure 9: Reuse counts from the worked library. Seven shared modules support the 15 strategy files; the most widely reused module serves 16 bundle locations including the root. Label snippets are truncated in the original asset. The two most-used modules account for 47% of reuse and the top three for 64%.

branch-specific workflow and verification obligations remain represented. This example illustrates the mechanism rather than a claim that every generated module has the same size or payoff.

D.4 Repeated use and increasing library size

Table 12: Cumulative token accounting for one deployment and N executions, averaged over benchmarks. "Kept" applies the recorded 95% line-retention criterion, not universal semantic preservation. Each workload is compared against its uncompressed reference.

Method	Kept	N=1 ↑	N=10 ↑	N=50 ↑	N=200 ↑	N=1000 ↑
Root-only	no†	+5.1%	+11.5%	+13.8%	+14.3%	+14.5%
Flat-concat	no†	+63.6%	+63.1%	+62.9%	+62.9%	+62.9%
SkillReducer^{a}	yes	+5.1%	+5.4%	+5.4%	+5.5%	+5.5%
Expert Progressive	yes	+13.1%	+11.7%	+11.3%	+11.1%	+11.1%
SkillZip Pro	yes	+17.9%	+13.9%	+12.5%	+12.2%	+12.1%

The workload ledger adds one deployment to N task executions. Among methods satisfying its 95% line-retention criterion, Pro has the highest reported saving at all five measured workload sizes, from 17.9% at $N = 1$ to 12.1% at $N = 1000$. Expert Progressive approaches 11.1% and SkillReducer 5.5% in the same ledger. This comparison is confined to that workload aggregation: it should not be substituted for the grown-track run column of Table 8, which uses a different reported aggregate.

As the library grows, a root containing explicit branch dispatchers still incurs navigation overhead. Pro's shipped reduction increases from 21.8% to 25.5%, while its per-run saving falls from 17.6% to 5.3%. Expert's corresponding large-library per-run saving is 10.4%. These values constrain the

Original branch: 527 tokens

```
## Purpose
Solve competition-style math
## Approach
3.  Notation for all
unknowns...
4.  Solve step by step...
## Rules
- Never round intermediate...
- Reduce fractions...
## Output
- \boxed{...} final answer
- multi-answer:  comma list
- single:  output only that
## Verification
- Substitute back
- Sanity-check units
```

Pro branch: 160 tokens

```
## Workflow
1.  Read [rounding rule]
  [multi-answer]
  [single]
2.  Notation for unknowns...
3.  Ordered vs. unordered...
4.  If ambiguous,
  state assumptions
5.  \boxed{...}
## Verification
- Substitute back
```

Rules and outputs use linked shared modules. The branch-specific procedure, guarded instruction, verification, and routing remain represented.

Flat-concat control: the selected branch disappears. Only `round_00` and `round_10` remain (2/15 branches); the selected branch's rules, output instructions, verification, routing link, and approach are unavailable. The root becomes a flat list containing `--name: ...`, `read [round_05](...)`, and `read [specialist](sub/...)`. The .2% content-retention result is the real-library aggregate in Table 8.

One shared module (315 tokens; 16 source occurrences).

```
- Never round intermediate results unless the problem explicitly asks
for a decimal.
- Reduce fractions to lowest terms and rationalize denominators where
standard.
- For counting:  ordered vs. unordered, with vs. without replacement.
- Give the answer in simplest exact form.
```

Reuse ledger. Precision/output rules occur at 16 bundle sites; the multi-answer instruction at 15, and the single-answer instruction at 7. Pro creates 7 shared modules, keeps 15/15 branches, and reduces the selected branch by about 70%. Replacing 15 copies of this module avoids 4,725 body tokens before reference overhead.

Figure 10: A branch before and after cross-file reuse, with the flat-concat failure shown separately. Yellow marks repeated source sections; green marks shared references; blue marks an explicit guard. Ellipses remain excerpt markers, not claims that omitted source obligations were deleted.

Table 13: Library-size sweep: smallest and largest tested bundles. Values are benchmark means; Kept and Route use the largest size. These data do not imply constant per-run savings or dominance on every cost measure.

	Shipped saving		Per-run saving		At largest size	
Method	small ↑	large ↑	small ↑	large ↑	Kept ↑	Routing ↑
Root-only	+2.0%	+5.2%	+8.7%	+3.5%	0.896	0.000
SkillReducer[a]	+5.4%	+4.3%	+4.7%	+2.1%	1.000	1.000
Expert Progressive	+12.4%	+22.6%	+18.4%	+10.4%	1.000	1.000
SkillZip Pro	+21.8%	+25.5%	+17.6%	+5.3%	0.984	1.000

interpretation of the source curves: Pro retains every route with .984 source-line retention at the largest size, but it does not have constant run savings or the best large-library run reduction in this table. A proposed additional on-demand grouping adds a hop and is rejected by the non-inflation check.

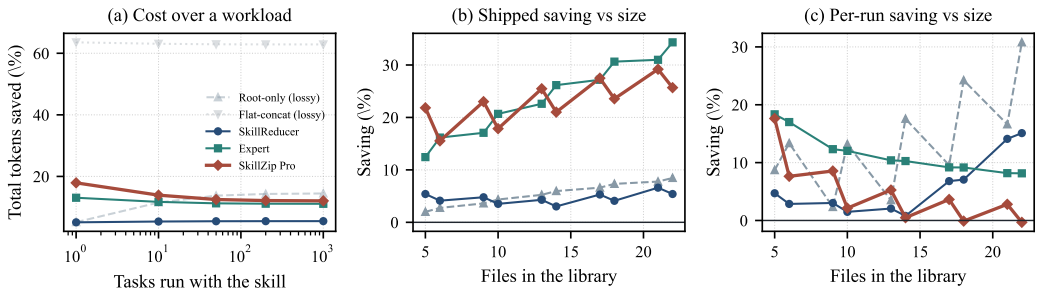

Figure 11: Workload and library-size sweeps. (a) Cumulative saving over repeated use; (b) shipped saving and (c) per-run saving over library size. Read the plotted trends with Tables 12–13; declining per-run reduction remains visible.

D.5 Real evolution streams, start time, and processing cost

The start-time experiment asks how many published versions benefit from compression; the schedule comparison asks how much work each update requires. We separate these questions below. Figure 12 provides enlarged start-time plots for the comparison in Figure 3(b–d).

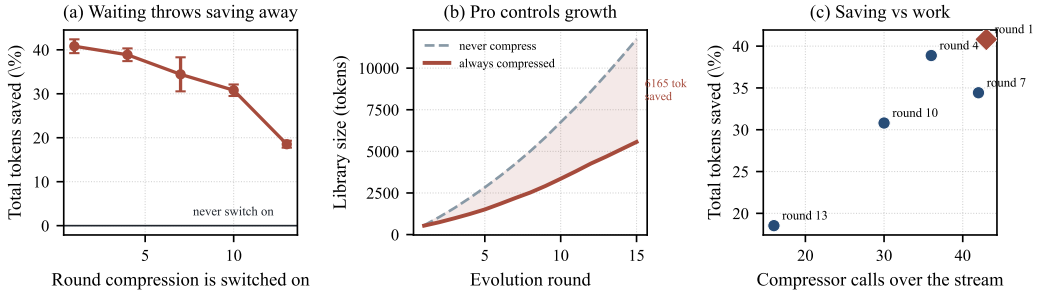

Figure 12: Activation-time ablation, enlarged from Figure 3(b–d). (a) Published-size saving versus start round, with variation across three libraries; (b) library size over evolution; (c) saving versus processing invocations. These measurements sum published bundle sizes, not provider-billed runtime.

Table 14: Start-time ablation over 15 rounds. Work counts node-processing invocations.

Start	Carried	Saved	Work
Round 1	48,055	40.8%	43
Round 4	49,627	38.9%	36
Round 7	53,234	34.4%	42
Round 10	56,179	30.8%	30
Round 13	66,139	18.5%	16
Never	81,176	0.0%	0

Table 15: Compression overhead. Every row uses 0 model calls, 0 model tokens, and 0 rollouts.

Method	Time (s)	Peak (GB)
Root-only	.11	.02
Flat-concat	.14	.02
SkillReducer[a]	.12	.02
SkillZip Pro	.21	.02

Enabling compression at round 1 rather than 13 lowers cumulative published size from 66,139 to 48,055 tokens. The corresponding savings are 18.5% and 40.8%. Early activation uses more processing work (43 rather than 16 invocations), and the invocation counts are not monotone in start time because repack schedules differ. The difference between round-1 and round-7 saving is 6.4 percentage points, from 40.8% to 34.4%. Later compression can shrink the current bundle but cannot change versions already published.

We next examine the underlying library and update results. Table 8 includes all three real libraries: Pro publishes two candidates and returns the third source byte-for-byte because its candidate does not improve the materialized objective. The accepted-output saving of 34.7% therefore differs from

the 23.1% mean including fallback. Figure 13 retains the original point groups, trajectories, and work–distance plot; fallbacks remain in the aggregate tables.

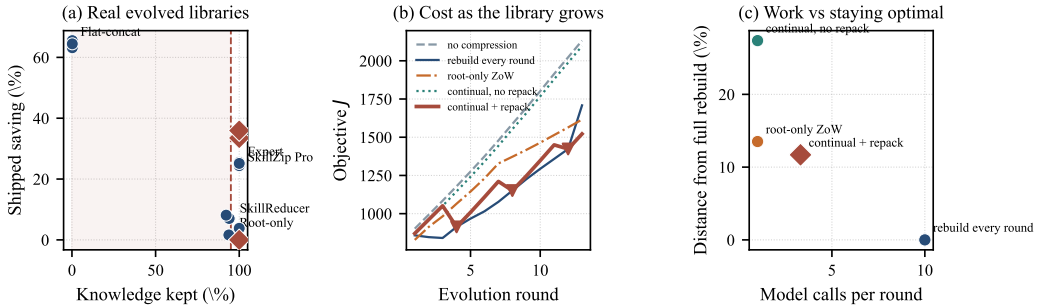

Figure 13: Real-library and evolution diagnostics. (a) Saving versus retained content; (b) published objective with repack markers; (c) update work versus distance from rebuilding. "Model calls" counts processed text nodes, not API calls. The plotted distance is distinct from the signed three-library aggregate drift in Table 2.

Table 16: Full per-round work ledger on the real 13-round streams. Work is text-node processing. Savings are percentages; drift is a relative difference from the heuristic full rebuild.

Schedule	Ship	Run	Drift	Work	Bytes	ms
No compression	0.0	0.0	.000	0.00	3,201	0
Full rebuild	35.1	18.7	.000	10.00	13,388	234
Root-only Zip-on-Write	2.4	19.6	−.027	1.00	389	94
Pro, local only	2.7	.8	.262	1.00	3,201	8
Pro + repack	48.1	28.2	−.083	3.31	5,697	68

The continual study replays 13 rounds for each of three libraries. An earlier accepted repack can remain smaller than a later heuristic rebuild whose candidate is rejected. Negative aggregate drift therefore does not imply improvement over a global optimum. Without repacking, local repair misses repetition whose common structure becomes apparent only across several updates. Table 16 preserves all per-round processing measurements.

The prefix analysis in Figure 4 addresses a separate question: how reusable representation grows with the library. For raw prefix size $U(k)$ and compressed prefix size $C(k)$, its marginal repetition proxy is

$$1 - \frac{C(k) - C(k-1)}{U(k) - U(k-1)}. \tag{20}$$

This proxy can reflect representation changes as well as duplicated prose; it is not independent semantic annotation. The prefix analysis includes only the two libraries accepted at every prefix. The third remains included in the three-library One-Shot and continual results. Figure 4 retains the original 29% to 53%, 55%±3%, and reuse-concentration measurements under this condition.

D.6 Structural patch replay and repack policies

Table 17: Structured patch-stream results after the final patch. Savings are relative to publishing without compression; drift compares a full rebuild. Time, bytes, and processing invocations are per patch. This is a different stream from Table 2.

Schedule	Ship saving ↑	Run saving ↑	Drift ↓	Update s ↓	Bytes ↓	Work/patch ↓	ΔQuality
Append Only	+0.0%	+0.0%	+4.6%	0.001	263	0.0	0
Pro full rebuild	+12.4%	+2.1%	+0.0%	0.170	6250	8.8	0
Root-only update	+9.4%	+21.7%	-27.7%	0.115	604	1.0	0
Pro local only	+7.7%	+4.7%	-0.8%	0.013	263	1.0	0
Pro continual + repack	+12.4%	+2.1%	+0.0%	0.060	1875	3.3	0

The structured stream differs from the real 13-round replay. Here continual repacking matches the full rebuild's final objective with 3.3 rather than 8.8 processing invocations and 1,875 rather than 6,250 touched bytes per patch. The source table records zero quality change in every row; it does not include uncertainty estimates for those deltas. A more negative drift for the root-only control is not evidence of better faithful compression, because it may rewrite navigation that Pro protects.

Table 18: Repack policies on the structured patch stream. Rate is the fraction of patches rebuilt globally. Worst drift is the maximum recorded relative drift, not a guarantee for arbitrary future updates.

Rule	Repack rate	Worst drift ↓	Work/patch ↓	Update s ↓
Never repack	0.00	0.0014	1.00	0.000
Every patch	1.00	0	9.50	0.413
Savings only (θ)	0.00	0.0014	1.00	0.000
Growth only (ρ)	0.00	0.0014	1.00	0.000
Patch count only (K)	0.25	0.0014	3.50	0.119
Combined policy	0.25	0.0014	3.50	0.106

On this stream, neither the savings-only nor growth-only trigger fires. The patch-count and combined policies repack one quarter of updates; all non-every-patch policies have a recorded worst drift of .0014. The experiment therefore identifies the patch counter as the active trigger at this scale, not an empirical benefit for every component of the combined trigger. Table 19 reports all original

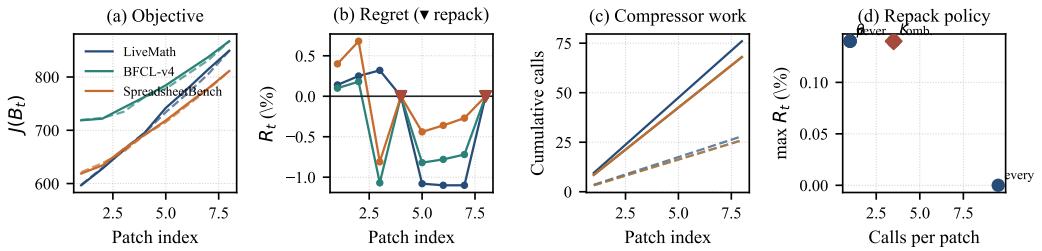

Figure 14: Structured continual diagnostics: objective, relative drift with repack markers, cumulative processing work, and policy sensitivity. Original "calls" labels use the replay's node-count convention. The curves are not a measurement of model API calls.

Table 19: Complete component ablations. Reductions are percentages except J. Damage counts unreachable files plus broken links. Only the full method has reported task success (.560) and required-resource recall (.755); no missing variant outcome is inferred.

Variant	J saving	Shipped	Root	Run	Damage
SKILLZIP PRO	.142	21.1	7.3	15.8	0
No resource graph	.118	13.7	7.3	13.6	0
No host entailment	.077	15.3	2.0	8.3	0
No scoped sharing	.132	7.9	7.3	17.0	0
No capsules	.126	26.9	7.3	11.8	0
No cross-file audit	.142	21.1	7.3	15.8	0
No routing lock	.156	22.2	7.1	17.8	0
Global sharing†	−.166	17.0	−84.9	−.9	1

component-cost rows. Only the complete method has measured quality columns in that source ablation table (task success .560; required recall .755); dashes for other variants mean unreported values, not zero. Removing the audit alone causes no damage on these tested examples because the other checks remain. The entry-ablation experiment below supplies a concrete failure when an independent entry is not checked.

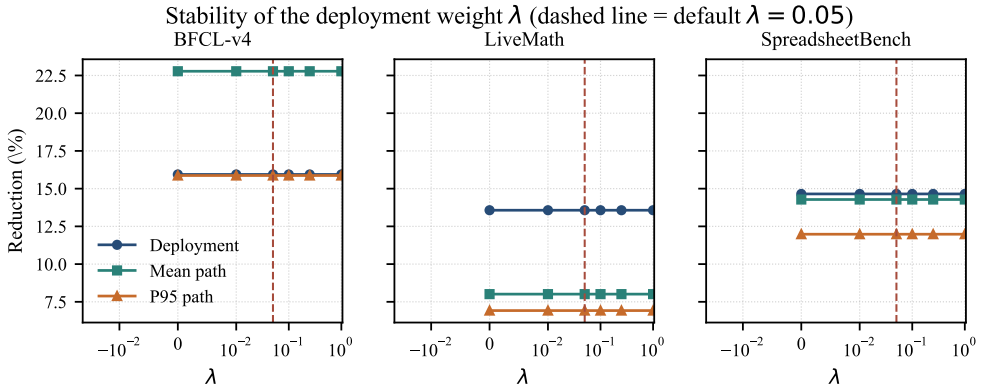

Figure 15: Storage-weight sensitivity. The tested λ values are $0, .01, .05, .1, .25, 1$; the selected transformations and reported costs do not change. This is empirical stability on these bundles, not a guarantee for arbitrary workloads.

D.7 Independent entry content and direct-call execution

Table 20: Entry-preserving compression. Disk: canonical-byte ratio; Run/P95: tokens. The Transient row describes the unchanged canonical bundle; Table 24 measures compressed views.

	Cost ↓			Structural fidelity ↑		
Method	Disk	Run	P95	Route	Public mean	Public min
Uncompressed	1.000	404	607	1.000	1.000	1.000
SkillZip	.959	393	587	1.000	1.000	1.000
Root-only†	.972	242	537	.000	.625	.250
Flat-concat†	.379	74	190	.000	.062	.000
Pro, no entry audit†	.845	331	565	1.000	.500	.000
Pro, audited persistent	**.884**	**352**	607	**1.000**	**1.000**	**1.000**
Pro, transient (source)	1.000	404	607	1.000	1.000	1.000

Table 21: Decomposing multi-entry independence. Discoverability and content are averaged separately; Effective is the average of their per-entry product, not necessarily the product of the displayed means. Cond. denotes conditional-entry content.

Method	Discovery ↑	Content ↑	Effective ↑	Cond. ↑
Uncompressed	1.000	1.000	1.000	1.000
SkillZip	1.000	1.000	1.000	1.000
Root-only ✗	1.000	0.625	0.625	1.000
Flat-concat ✗	1.000	0.062	0.062	0.150
Pro, no entry audit ✗	0.500	0.929	0.500	0.887
Pro, audited persistent	**1.000**	1.000	**1.000**	1.000
Pro, transient	**1.000**	1.000	**1.000**	1.000

The controlled math bundle has a root, a catalogued public subskill at `sub/SKILL.md`, and four conditional references. Content can remain in a file that is no longer exposed under its advertised path. That is why the no-audit variant has .929 average public content but .500 average effective independence. The latter is the mean of per-entry coverage–discoverability products, not the product of two independently averaged columns.

Table 22: Direct-call rollout: public startup and task success by entry type. Sample sizes are $n = 3$ for the public entry and $n = 8$ per reference. Missing root-mediated cells are not filled by inference.

Method	Discovery ↑	Pub. ↑	Cond. ↑	Root ↑
Uncompressed	1.000	0.333	0.375	0.364
Pro, no entry audit ✗	0.000	0.000	0.417	0.394
Pro, audited persistent	**1.000**	0.333	0.375	–
Pro, transient	**1.000**	0.667	0.417	–

The live direct-call test cannot start the renamed public subskill under the no-audit variant. Audited persistent and transient outputs preserve startup. The transient .667 versus .333 public success is based on only three examples; it is not evidence of broad accuracy superiority. Conditional and root-mediated results remain separate, and absent cells are left absent.

D.8 Lifecycle boundaries, per-entry view cost, and safeguard ablations

Table 23: Lifecycle sensitivity under the tested policies and workload. The reported crossover compares global persistent repacking to local transient rebuilding; it is not a universal threshold.

Sweep	Range	Measured effect	Verdict
Public-node fraction	0.10→1.00	standalone keeps 0.10→1.00 of nodes	more public ⇒ less persistent deletion room
Direct-call frequency N	1→50 calls	persistent load < transient load at all N	persistent-standalone has lower load
Cross-file redundancy	0.35→0.41 dup.	persistent deploy saving up to 28.1%	more redundancy ⇒ larger permanent saving
Bundle size (tokens)	548→879	cold view build 170→228 ms	build cost is per-run for transient, one-time for persistent
Update frequency U	0→1 edits/run	repack 273 ms vs rebuild 183 ms at U=1	transient lower rework once $U > 0.1$
Workload drift d	0→0.8	persistent saving 12.9%→2.6% vs transient 4.7% (flat)	transient higher saving for $d \geq 0.8$

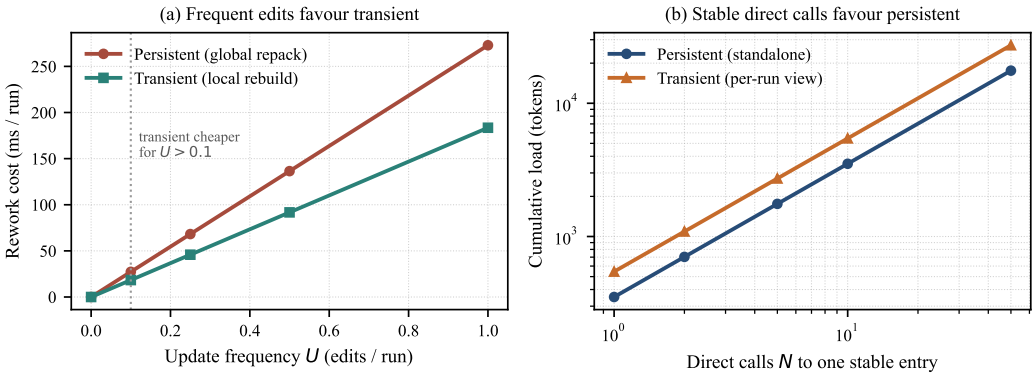

Figure 16: Full lifecycle sweep. Local transient rebuilding and global persistent repacking have different rework costs; repeated stable direct calls favor the persistent bundle in the measured policy. These boundaries are not universal across update policies or cache hit rates.

Figure 16 visualizes both measured comparisons; the update-rate panel is highlighted in Figure 5. Persistent compression has the lower cumulative loaded-token count for stable direct calls in the tested range. Frequent edits favor local transient rebuilding over the particular global persistent repack policy. Different cache behavior or a different persistent update policy could change that boundary; the crossover is not an intrinsic threshold of all persistent and transient systems.

Table 24: Transient-view construction by entry. Closure/View are token counts; cold, warm, and amortized costs are milliseconds (0.9 cache-hit rate). Root and sub denote `SKILL.md` and `sub/SKILL.md`; remaining names are `references/<name>.md`. Built marks a strictly smaller view, and Canon. confirms unchanged source bytes.

Entry	Kind	Closure	View	Saving	Cold	Warm	Amort.	Built	Canon.
`root`	publ	493	371	**24.8%**	218	0.34	22.1	✓	✓
`sub`	publ	548	545	0.5%	173	1.53	18.7	✓	✓
`algebra`	cond	532	529	0.6%	155	0.23	15.7	✓	✓
`edge_cases`	cond	513	505	1.6%	153	0.29	15.5	✓	✓
`geometry`	cond	489	489	0.0%	244	0.54	24.9	✗	✓
`number_theory`	cond	483	478	1.0%	157	0.56	16.2	✓	✓

A view must be measured on its own closure, not on the unchanged canonical bundle. Thus the 493-to-371 root-view result and the canonical 404 mean-run count in Table 20 have different denominators. The geometry view does not improve and correctly falls back. Cold build latency, cache-hit latency, and amortized latency are distinct from token saving and cannot be added without specifying the workload and its unit costs. The labels, closure, host-context supplement, and multi-entry audit

Table 25: Lifecycle safeguards and concrete failures. The intact/ablated values refer to the same row-specific diagnostic. The .845 disk ratio belongs to this source ablation ledger, not to the standalone-preserving .884 configuration.

Removed safeguard	Metric	Intact	Ablated	What breaks
− Entry-type labels	mean public independence (math/structured)	1.000	0.500	public entries lose content the root happens to cover
− Entry audit	all public entries discoverable (0/1)	1.000	0.000	a public sub-entry is renamed and no longer callable
− Dep. closure	dependency files reachable from the entry in its view	11	1	the view omits every reference the entry routes to
− Host context	mean conditional-entry independence (math/structured)	1.000	0.887	content assumed covered by the host is deleted, then truly lost
− View cache	mean build ms/call over 20 calls	18.7	173.4	every call rebuilds the same view from scratch
− Global repack	deploy token ratio, lower is better (math/structured)	0.845	0.959	cross-file duplication remains; only per-file gains

protect different correctness obligations. The cache and repack rules instead protect efficiency. Keeping those diagnostics separate explains why a configuration can preserve routing yet lose a public entry, or retain all entry content but pay to rebuild the same view on every call.

D.9 Production configuration and session-level evidence

The production record has three relevant uncompressed session baselines: 88% for the initial paired comparisons, and 92%/91% for the two final paired runs. The standalone official evaluation reports 90.91%; it is not substituted for any same-session baseline. Table 6 separates the source's unprotected structural and model-assisted conditions, which remove 71.4% and 75.8% respectively. The original compact table combines them under one row with a 71.4% deployment label and 70%/62% accuracies; the two reductions are explicitly stated in its surrounding text.

The unprotected rows produce 29 and 35 false positives versus 10 in their reference, with false negatives between one and three. Inspection identifies authorized-seller exceptions, whitelist clauses, and evidence thresholds as missing decision-boundary knowledge. Language-specific modality signals restore Chinese required-unit detection before applying witnesses. Exemptions remain locked even if an entailment checker proposes their removal.

W_3 accepts evidence-class candidates only; it excludes prohibited/output/exemption markers and logs the decision basis. Intra-root checking also excludes YAML frontmatter and every interface-contract span, removes the candidate from its own evidence base, and retains overlapping approved candidates

when normalized bigram overlap exceeds .6. That veto reduces one source of circular justification but is not a complete semantic-dependency analysis.

The renderer initially separates two worked JSON examples and the label whitelist by about sixty lines. C0 restoration removes scattered fragments and reinstates the source section contiguously. This restores an independently auditable interface property even though the one-task accuracy difference is too small to attribute a causal quality improvement. The source's unprotected-table interface label and its narrative are not fully configuration-aligned; no separate unprotected interface measurement is inferred.

The final root-focused configuration reports 38.1% deployment and 10.4% pooled task-paired runtime saving. The two 100-task runs yield 9.5% and 11.3%; a separate 30-task pilot ranges from 6% to 14% across days. Both final compressed runs have 89% accuracy, versus 92% and 91% in their same-session controls. The source reports FP=9 for this configuration but does not provide a separate FP count for each of the two sessions; Table 6 therefore does not duplicate that count. These observations do not support a lossless-production claim. The record reports roughly five to six times as much measured token saving for a removed root token as for one removed from an on-demand reference, consistent with repeated root exposure in about 13 reasoning rounds, but it does not causally decompose the execution change.

D.10 Cross-model execution and overhead

Table 26: Cross-executor behavior on the two-suite structured track. The same compressed bundle is executed without file changes. There are 27 Math and 17 BFCL tasks; cells are benchmark macro averages, separate from the main 102-task suite.

	Task success ↑		Required recall ↑		Irrelevant load ↓	
Executor	Raw	Pro	Raw	Pro	Raw	Pro
Qwen3.7-Max	.541	.619	.679	.690	.321	.300
Qwen3.6-Plus	.608	.656	.738	.727	.224	.223
Kimi K2.6	.611	.574	.435	.490	.077	.287

Table 26 preserves all six raw/compressed metrics for each of the three executors. It uses a separate two-suite structured track (27 Math and 17 BFCL tasks), with Qwen3.7-Max as the evolution model. It therefore should not be numerically pooled with the 102-task grown suite. The recorded executor identifiers are Qwen3.7-Max, Qwen3.6-Plus, and Kimi K2.6 (Qwen Team, 2026b;a; Moonshot AI, 2026). One set of files runs without modification across them, but task success and irrelevant loading remain model-dependent.

The structural compression overhead table reports no model usage and no task rollouts for all four conditions. Its .21-second Pro cost includes graph resolution, cross-file transformation, and a disk-level audit; peak memory is .02 GB. These timings do not describe production W_3 checking. The replay count of one processing invocation per changed text node is a work accounting convention, not a contradicting model-call measurement.

E Implementation Details

This appendix fixes the implementation contract needed to reproduce the harness-neutral compiler. It specifies bundle discovery, reference resolution, candidate layout, the cost ledger, auditing, atomic publication, and Zip-on-Write invalidation. Appendix F gives machine-readable schemas and prompt contracts; Appendix G gives the run matrix; Appendix H expands theory, limitations, and failure analysis.

E.1 Reference Directory Layout

The implementation treats the authoritative source directory as immutable during a transaction and creates all intermediate and persistent compiler state outside the published runtime bundle. A successful publication uses the following split layout:

```
<published-bundle>/
  SKILL.md # unchanged entry path
  references/...
  subskills/.../SUBSKILL.md
  capsules/<guard-slug>.md # generated on-demand text
  .skillzip_shared/<hash>.md # generated scoped modules
  scripts/... # byte-identical
  data/... # byte-identical
  assets/... # byte-identical

<compiler-cache>/<bundle-id>/
  state.json # graph, indices, workload ledger
  manifest.json # provenance and audit results
  contracts/<digest>.json # reusable typed contracts
  authored.snapshot # optional transactional source snapshot
```

Generated runtime directories are namespaced to avoid collisions. If either namespace already exists in the source, the tool selects a digest-suffixed namespace and records it in the external manifest. Nested source `SKILL.md` files are treated as subskill entries internally; the published filename is changed only when the original harness accepts `SUBSKILL.md`. The default is path preservation. Compiler state is not distributed to the agent and is therefore not charged to runtime deployment cost; a reproducibility package that includes it reports payload bytes and total archive bytes separately.

E.2 Discovery and Media Classification

Directory traversal is deterministic: paths are normalized to UTF-8 NFC, sorted bytewise, and visited without following directory symlinks. Content signatures and extensions classify regular files. Declared text types support contract extraction; code, notebooks, structured data, images, archives, and unknown binaries remain locked.

Every node stores:

- canonical path relative to the bundle root;
- SHA-256 of source bytes and media classification;
- token and byte counts;
- loading class and inbound/outbound edges;
- whether rewrite, relocation, or only verbatim copy is permitted.

The default text-size guard is 1 MiB per file. Larger text remains locked unless the user explicitly raises the limit. Archive members are never expanded implicitly.

E.3 Reference Resolution

The Phase-A resolver recognizes only syntax that the host agent can already follow:

1. Markdown links and images with relative local targets;
2. explicit code-formatted paths ending in a known extension;
3. front-matter fields declared by the selected skill format;
4. imperative loading clauses such as "read `references/csv.md`."

Fragments and query strings are separated before filesystem resolution and restored in the emitted link. Percent decoding is performed once; double decoding is forbidden. The canonical target must have the canonical root as a path-component prefix, not merely a string prefix.

The nearest enclosing heading and conditional clause define the initial guard. If guard extraction is ambiguous, the edge is marked `unknown`; it may be preserved but cannot justify capsule creation or path-specific sharing. External `http`, `https`, and `mailto` links are preserved as opaque strings and excluded from local closure checks.

E.4 Contract Extraction and Normalization

Files are chunked only at heading boundaries, with a 256-token overlap carrying the parent heading and active guard. Each chunk is extracted once. Units are merged by source span and normalized key; disagreements retain the stricter wording as a locked residual. The implementation never resolves a disagreement by majority vote.

Normalization is type specific. Tool calls normalize tool name, required arguments, and order. Output obligations normalize field name, type, and cardinality. Prohibitions preserve polarity. Workflow units form a directed graph whose edges encode `before`, `after`, `retry`, or `fallback`. Examples are tagged as non-normative unless the source explicitly declares them required.

E.5 Protection Classes, Witnesses, and Non-Latin Bundles

The production configuration in Section 5.5 adds three implementation requirements.

Script-aware lexical signals. Modality and requirement detection are lexical: obligation, prohibition, and recommendation markers drive which units are required. The reference implementation ships with English marker sets, which on a Chinese bundle demotes almost every obligation to optional (7 of 264 units required on the production skill). The fix is script-aware signal sets—obligation and prohibition markers per writing system—applied in the same deterministic extractor, so required-unit detection works before any witness is consulted. The English path is unchanged byte for byte.

Boundary classes and the exemption lock. Units whose payload carries an exemption or negative-verdict marker (authorized cases, whitelisted exceptions, "judge normal" clauses) are classified as boundary units. Boundary units are required and additionally locked against wording compression and against W_3 entailment removal: they may only be moved verbatim. Markers are a configurable lexicon per language; the audit counts them so a release reports boundary retention explicitly.

Interface-contract restoration. After materialization, a post-pass locates each source section matching the interface-contract pattern (output-format headings with fenced schema examples and whitelist constraints), strips any of its normalized lines that the fixed-template render scattered elsewhere in the emitted root, removes render-scaffolding sentences, and re-inserts the whole source span verbatim at one site. The audit then verifies contiguity: the section must appear once, whole, with no fragments outside it.

Entailment witness protocol (W_3). The checker prompt presents the emitted root document S and one candidate reference segment P, and asks whether the decision-relevant content of P—conditions, verdicts, thresholds, whitelists, exemptions—is fully expressed in S, explicitly instructing that concrete case facts need not reappear but the rule they demonstrate must. The checker runs at temperature 0 with a fixed prompt; the first response line must be the verdict and the second a one-sentence basis. A verdict is accepted only for evidence-class segments containing no boundary marker, and the verdict, basis, segment coordinates, and prompt digest are written to the witness log, making every W_3 removal individually auditable after publication.

Intra-root witnesses and their safeguards. W_3 may also be applied inside the root document, where each candidate is checked against the root *minus itself*. Three safeguards apply. (i) The YAML frontmatter and every interface-contract span are exempt from W_3 regardless of verdict: they carry no decision knowledge a checker can weigh, yet they are the loading and output contract. (ii) Boundary segments remain exempt. (iii) A mutual-witness veto: after verdicts are collected, any two approved segments whose normalized bigram sets overlap above 0.6 are both retained, since each may have received its verdict only because the other was still present in the base. Because the root is injected into every reasoning round, intra-root removals carry the largest per-run multiplier, and the veto plus the exemptions are what keep that lever safe.

E.6 Candidate Enumeration

File-level candidates are primitive statements, shared rules, named procedures, guarded exceptions, and locked residual spans. Bundle-level candidates are exact host witnesses, shared modules, and capsules.

Cross-file sharing uses a two-stage test. A hash over canonical typed units finds exact candidates; a deterministic structural comparator then verifies identical type, payload, guard parameters, and normative strength. Parameterized sharing is allowed only when differing values correspond to explicit source variables and the resulting call sites preserve them. Pure embedding similarity never creates an automatic shared module.

For each support set, the algorithm considers the lowest common activation scopes in the resource graph rather than only the directory ancestor. A scope candidate is discarded if it would be loaded by a path that previously loaded none of the occurrences, unless the added path cost is compensated under Eq. (2) and the rule is proven universally applicable. The production default disables this exception and requires no scope expansion.

Capsule candidates require all of the following: an explicit guard, a body above the minimum length, no unguarded inbound dependency from adjacent prose, and a dispatcher whose mandatory loading instruction fits within the configured budget. Headings such as "Background" or "Notes" do not constitute guards.

E.7 Selection and Cost Ledger

Candidate deltas use tokenized emitted text, including links, dispatcher verbs, generated headings, and manifest-excluded runtime content. Selection starts with the best coverage-preserving improvement, then tests removals and one-for-one or one-for-two swaps until J cannot improve by one token.

The ledger stores both estimated and materialized values:

```
{
  "tokenizer": {"name": "...", "revision": "..."},
  "lambda_deployment": 0.05,
  "path_weights": {"source": "trace|guards", "digest": "..."},
  "source": {"catalog": 0, "activation": 0,
          "deployment": 0, "mean_path": 0, "p95_path": 0},
  "candidate": {"catalog": 0, "activation": 0,
            "deployment": 0, "mean_path": 0, "p95_path": 0},
  "objective_delta": 0
}
```

Estimated and materialized costs must agree exactly for bytes and within tokenizer determinism for tokens. A discrepancy rejects the candidate.

E.8 Disk-Level Audit

The audit runs in a new process with only the source path, candidate path, and manifest as inputs. This prevents it from trusting optimizer objects accidentally. It rebuilds both graphs, rehashes every node, and verifies:

1. source and candidate roots are distinct and canonical;
2. all internal candidate targets exist, remain inside the root, and preserve fragments;
3. every source contract unit maps to a compatible candidate unit or exact host witness;
4. every generated shared module or capsule is reachable through a mandatory dispatcher under a compatible guard;
5. locked-file path and SHA-256 pairs match, except for explicitly user-approved path migrations;
6. no source file has been omitted merely because it was unreferenced;
7. every interface contract appears as one contiguous span equal to its source section, with no scattered fragments;
8. materialized costs do not exceed the source objective.

Strict mode treats any warning as failure. Diagnostic mode emits the candidate for inspection but never publishes it as a successful compression.

E.9 Atomic Publication and Recovery

The tool creates a temporary sibling directory so that final rename remains on one filesystem. It fsyncs files, the manifest, and containing directories before publication. If the destination exists, the default command fails; explicit `-replace` first renames the old output to a timestamped backup. The source is never overwritten.

On one-shot extraction, selection, materialization, or audit failure, the official output is a verbatim copy of the source with an external failure manifest. On a continual optimization failure after a patch has been applied successfully, the official output is the verbatim patched bundle. Only a failure to apply or validate the patch transaction itself leaves the previous publication active. A temporary directory may be retained for debugging only under an explicit flag. This behavior makes failure visible while preserving the current authored semantics.

E.10 Dual-Mode Driver and Bundle-Aware Zip-on-Write

The command exposes `-mode one-shot` and `-mode continual`. One-shot requires only the source and entry path; it rebuilds all compiler state. Continual additionally requires the previous bundle identifier, state digest, and an incoming patch. A missing, corrupt, or source-mismatched state never causes an unsafe partial update: the driver applies the patch verbatim and switches to one-shot compression of that current authored state.

An incoming patch identifies added, modified, moved, and deleted paths. Before optimization, the tool creates a raw patched snapshot and validates patch preconditions and expected hashes. The invalidation closure contains changed nodes, their reference ancestors, newly or formerly referenced descendants, shared modules whose support sets changed, capsules whose guard spans changed, and environment witnesses whose digest changed. Extraction and candidate generation rerun only on this closure. Global reference, coverage, and hash checks still scan the full candidate, while unchanged file contracts are reused by source digest.

Each patch unit is classified as ABSORB, REFINE, EXTEND, or REFACTOR. The first three update semantic content. REFACTOR may also change placement when a support set, guard, or path weight crosses the sharing or capsule threshold. The driver estimates recoverable saving from stale candidates and invokes one-shot repacking when any configured trigger is reached: θ_{repack} saving, ρ growth, δ_W workload drift, environment-digest change, or K patches. Trigger values and causes are logged in the state transition.

Deletion is especially conservative. A shared module is deleted only after all inbound dispatchers are removed and the recomputed coverage graph shows no remaining source unit depends on it. If compression of a valid raw patch fails, the raw patched snapshot is atomically published and a clean state is rebuilt from it. Thus a patch can reduce compression temporarily but cannot disappear because the optimizer rejected its rewrite.

F Schemas and Model Contracts

This appendix records the structured interfaces used by the extractor and rewriter. Prompts are templates; implementations should pin the system message, model snapshot, JSON validator, and retry policy.

F.1 Resource-Graph Record

```
{
  "bundle_digest": "sha256:...", "entry": "SKILL.md",
  "nodes": [{"id": "n17", "path": "references/csv.md",
    "sha256": "...", "media_type": "text/markdown",
    "kind": "reference", "locked": false, "token_count": 412}],
  "edges": [{"source": "n1", "target": "n17",
    "source_span": [88, 126], "syntax": "markdown_link",
    "guard": {"text": "when exporting CSV", "status": "explicit"}}],
  "external_edges": [], "unresolved": []
}
```

Line/byte offsets are measured against immutable source bytes. A graph containing an unresolved local reference cannot enter strict compression.

F.2 Typed Contract Record

```
{
  "resource_id": "n17",
  "units": [{"unit_id": "n17:u4", "type": "output_obligation",
    "key": "csv.encoding",
    "payload": {"value": "UTF-8", "strength": "must"},
    "scope": {"resource": "n17", "guard": "export CSV"},
    "source_spans": [[241, 274]], "confidence": "high", "locked": false}],
  "relations": [{"kind": "before", "source_unit": "n17:u2",
    "target_unit": "n17:u4", "source_spans": [[178, 274]]}]
}
```

Allowed unit types are `interface`, `workflow`, `tool`, `rule`, `prohibition`, `output_obligation`, `evidence`, and `locked_residual`. The validator rejects unknown types and units without source spans.

F.3 Environment Contract and Witness

```
{
  "environment_digest": "sha256:...",
  "guarantees": [{"type": "output_obligation", "key": "csv.encoding",
    "value": "UTF-8", "scope": "all_spreadsheet_exports",
    "enforced_by": "serializer/v3", "evidence_digest": "sha256:..."}]
}
```

A removal witness stores the source unit, guarantee index, exact comparison result, and environment digest. Natural-language descriptions of the host are not accepted as contracts.

F.4 Extraction Prompt Contract

Contract extraction

```
You are extracting an auditable behavioral contract from ONE resource
in an agent-skill bundle.
Inputs:  canonical resource path; ancestor headings; explicit loading
guard; immutable source text with byte offsets; resource-graph
neighbors.
Return JSON matching the supplied schema.  Extract every applicability
condition, ordered workflow step, tool/resource requirement,
normative rule, prohibition, output obligation, verification/evidence
requirement, and ambiguous residual.  Attach exact source spans.
Preserve polarity, strength, and guard.  Mark uncertain or
non-decomposable text as locked_residual.
Do not rewrite the resource.  Do not infer requirements from world
knowledge.  Do not treat an example as normative unless the source
explicitly does.  Do not invent references or guards.  Output JSON
only.
```

F.5 Candidate Rewrite Prompt Contract

File rewrite

```
Rewrite the supplied Markdown resource using ONLY the selected
representation plan and mapped source units.
Hard requirements:  (1) preserve every selected typed unit at the
specified scope and normative strength; (2) preserve every locked
residual verbatim; (3) emit each provided relative path exactly; (4)
for a capsule/shared module, keep the supplied explicit guard and
```

File rewrite (continued)

```
mandatory read instruction; (5) do not add tools, claims, examples,
branches, or host assumptions; (6) return one Markdown file with no
commentary.
If the plan is inconsistent or omits a mapped unit, return
PLAN_REJECTED with the unit IDs.
```

The implementation does not rely on the rewriter to calculate coverage. It re-extracts and audits the emitted text.

F.6 Audit Prompt Contract

Most audit checks are deterministic. A model is used only for semantic coverage pairs not discharged by exact normalization.

Semantic coverage adjudication

```
Given one source unit and one candidate unit, decide whether the
candidate entails the full source requirement under the same or
narrower compatible scope.
Return:  COVERED, NOT_COVERED, or UNCERTAIN; a type match; a
polarity/strength match; a scope match; and the minimal text spans
supporting the decision.
Rules:  examples do not cover obligations; recommendations do not
cover MUST; broader applicability cannot be assumed from a narrower
branch; external knowledge and model plausibility are forbidden.
UNCERTAIN is treated as NOT_COVERED by strict mode.  Output JSON only.
```

F.7 Transformation Manifest

```
{
  "format_version": "skillzip-pro/1", "phase": "A",
  "source_digest": "sha256:...", "output_digest": "sha256:...",
  "models": {"extractor": "...", "rewriter": "...", "auditor": "..."},
  "graph": {"nodes": 0, "edges": 0, "unresolved": 0},
  "transformations": [
    {"kind": "host_entailment", "units": [], "witness": "..."},
    {"kind": "scoped_share", "units": [], "scope": [], "path": "..."},
    {"kind": "capsule", "units": [], "guard": "...", "path": "..."},
    {"kind": "file_cover", "resource": "...", "units": []}
  ],
  "cost_ledger": {},
  "audit": {"passed": true, "checks": [], "fallback": false}
}
```

The manifest is not required at agent runtime. It exists for provenance, regression testing, and exact reconstruction of experimental measurements.

G Detailed Experimental Protocols

This appendix preserves the experimental protocol and the reported execution settings. The full factorial matrix, annotation-agreement checks, stochastic repetitions, and multiple-testing provisions are protocol specifications; their presence does not establish that all cells were executed. Populated measurements are in Section 5 and Appendix D. Appendix H.4 records differences between the general protocol and the results actually reported.

G.1 Leakage-Control Sequence

The evaluation follows a fixed order to prevent test leakage:

1. freeze benchmark versions, evolution/validation/test IDs, and task-template groups;
2. construct native and structured-growth source bundles using only evolution tasks;
3. freeze the bundle builder, path annotations, environment contracts, and source digests;
4. tune only declared thresholds on validation tasks;
5. freeze compressor code, prompts, model snapshots, tokenizers, and non-inferiority margin;
6. run every compressor on the same source digests;
7. audit outputs before any held-out execution;
8. execute all valid or fallback outputs on paired held-out tasks;
9. generate tables directly from immutable raw records.

Any post-freeze repair creates a new experiment revision and reruns all affected baselines.

G.2 Bundle Construction Checks

The structured-growth builder moves contiguous source spans into references or subskills and replaces them with explicit guarded links. Before compression, three checks establish that this transformation is not itself responsible for performance changes: (i) concatenating files in provenance order recovers all normative source spans; (ii) every moved span has exactly one reachable dispatcher; and (iii) the evolved flat skill and the structured bundle are compared on validation tasks under the same executor. A task-score difference above one percentage point triggers manual review and rebuilding.

Each benchmark includes at least four branch types: a common path, a rare format/tool path, a verification path, and a recovery path. Path-frequency scenarios include observed validation traces, uniform leaf paths, and a skewed stress distribution with 80% mass on the common path. This separates gains from bundle structure from gains tied to one traffic assumption.

G.3 Gold Resource Sets

For each held-out task, annotators identify the minimal set of source resources required to satisfy the explicit task and benchmark rubric. They may inspect the task, source bundle, tool schema, and official expected behavior, but not any compressed output. A second annotator independently labels a stratified 20% sample covering all branches. Disagreements are adjudicated before execution; The protocol calls for Cohen's κ and raw agreement; those statistics are not populated in the supplied results.

A resource is required if omitting it removes a unique normative unit needed by the task. Redundant references are marked acceptable alternatives rather than jointly required. Scripts invoked by a required workflow count as required even if their source code is not injected into the model context.

G.4 Run Matrix

The full factorial matrix is summarized in Table 27. Budget-matched and native-cost comparisons are separate run groups. The no-skill condition is executed but not compressed.

Table 27: Protocol run matrix before stochastic seeds.

Dimension	Levels
Benchmarks	3
Bundle tracks	2
Compression methods	7
Compressor models	3
Executor models	3
Traffic scenarios	3
Seeds, if stochastic	3

Scope of this matrix. This is the source protocol's factorial specification, not a count of populated experimental cells. The main results fix the compressor and traffic estimator; the cross-executor study is a separate two-suite track. Deterministic runs do not require stochastic repeats. Neither missing seeds nor annotation-agreement statistics are reconstructed from the aggregate tables.

For the main table, compressor model and traffic estimator are fixed in advance, reducing the execution grid to all methods, benchmarks, tracks, executors, tasks, and seeds. The full compressor–executor cross product is used only for the portability study.

G.5 Harness Instrumentation

Instrumentation records timestamp, task ID, selected skill, read path, read result, byte count, model-token count, caller action, and current branch label. It does not expose gold paths to the agent. Catalog and root injection are recorded as synthetic events emitted by the existing harness; auxiliary loads are recorded at the ordinary file-reading boundary.

Two validation tests are mandatory. The *visibility test* verifies that the agent cannot read the trace buffer. The *equivalence test* replays a fixed set of actions with instrumentation on and off and compares all task-visible observations byte-for-byte. Hashes of the harness executable and configuration accompany each trace.

G.6 Task Execution and Failure Policy

A compression-time exception, audit rejection, or timeout produces the original bundle as the executable fallback. Its task score and uncompressed costs remain in the method's aggregate; excluding failed cases would reward unsafe aggressiveness. An execution-time missing file, invalid path, or unavailable generated module counts as both task failure (when it prevents completion) and a loading error.

Task timeouts and transient provider failures are distinguished. A provider failure is retried under the same seed up to the predeclared limit; a reproducible model or agent failure is not retried. Every exclusion is listed by task ID and reason.

G.7 Metric Computation

Catalog and activation cost are deterministic functions of published files. Deployment cost includes all files distributed to the agent except the audit manifest when the production package explicitly excludes it; both inclusive and runtime-package counts are recorded. Path cost is calculated from actual load events, including duplicate loads. Mean and P95 are paired over identical tasks.

For a task t, required-file recall is defined as one when R_t is empty. Irrelevant-load rate is zero when L_t is empty. Generated shared modules inherit the union of source-unit annotations they cover, allowing them to count as a valid substitute for original resources. A capsule is relevant only when its guard is active for the task.

Quality macro-averages first within benchmark categories, then across the three benchmarks, preventing a large benchmark from dominating. Compression ratios are computed from summed token costs before averaging; we additionally report median per-bundle reduction to reveal size effects.

G.8 Confidence Intervals and Tests

The task is the resampling unit. Paired bootstrap samples preserve all method outcomes, path costs, and seeds for a task. The primary comparison is SkillZip Pro versus Evolved Bundle for quality and versus Root-only SkillZip for mean-path reduction. Secondary comparisons cover other baselines and cost layers.

The general protocol specifies a -0.01 lower-bound criterion. The populated paired-result table instead uses -0.05; its reported conclusion is evaluated under that margin, not the stricter protocol value (Appendix H.4). Only after non-inferiority is established do we test compression superiority. McNemar's test operates on paired binary outcomes; multi-level official scores use paired bootstrap or permutation tests. Holm correction is performed separately within RQ2–RQ7.

G.9 Table Population Contract

The source table-population protocol specifies that every cell in Section 5 is emitted by the table script from a row-oriented result file; nothing is transcribed by hand. The pipeline writes one JSONL

record per (benchmark, track, method) with the following shape, and the script renders each table body from those records:

```
{
  "benchmark": "...", "track": "native|structured",
  "method": "pro|root_only|flat_concat|skillreducer|expert|...",
  "cost": {"catalog_tokens": 0, "activation_tokens": 0,
         "deployment_text_tokens": 0, "path_mean_tokens": 0.0,
         "path_max_tokens": 0, "path_count": 0},
  "J": 0.0,
  "reductions": {"catalog": 0.0, "activation": 0.0,
              "deployment": 0.0, "mean_path": 0.0,
              "max_path": 0.0, "J": 0.0},
  "loading": {"orphaned_modules": 0, "ref_errors": 0,
           "dangling": 0, "unsafe": 0, "reachable_files": 0},
  "routing": {"pairs_before": 0, "pairs_exact_kept": 0,
           "routing_fidelity": 0.0},
  "efficiency": {"time_s": 0.0, "compress_calls": 0,
              "rollouts": 0, "peak_gb": 0.0},
  "report": {"selected_verbatim": false, "audit_ok": true,
          "promotions_count": 0, "capsules_count": 0,
          "env_drops_count": 0}
}
```

Quality and behavioral-loading records are stored separately, keyed by the same (benchmark, track, method) triple plus an executor field, and carry `accuracy`, `required_recall`, `irrelevant_load`, `dispatch`, and the per-task read traces. Continual records add the per-patch trajectory with $J(B_t^{\text{one}})$, $J(B_t^{\text{cont}})$, the regret of Eq. (18), and an explicit repack flag.

The generated artifacts are: `costs.jsonl` (method costs and loading structure), `ablations.jsonl`, `quality.jsonl`, `transfer.jsonl`, `continual.jsonl`, `lambda_sweep.jsonl`, and `repack_sensitivity.jsonl`. A failed compression remains in the denominator and is represented by the cost of its verbatim fallback, which the record marks with `selected_verbatim`.

G.10 Zip-on-Write Replay

Each evolution patch is a transaction with timestamp, added/modified/moved/deleted paths, inserted source spans, and expected source digest. All schedules see patches in identical order and operate on the same verbatim authored checkpoint: append-only, Pro One-Shot after every patch, root-only SkillZip Zip-on-Write, Pro Continual without repacking, and Pro Continual with the combined repack policy. We checkpoint after every patch and measure changed contract units, invalidation-closure size, reused-contract fraction, rewritten bytes, model calls, update latency, four costs, objective regret, audit outcome, fallback type, and validation quality. Held-out test quality is measured only at predeclared checkpoints to avoid adaptive tuning to the test set.

The growth plot uses patch index on the horizontal axis and reports absolute tokens, not only normalized reduction. Additional panels report objective regret, cumulative compression cost, and closure fraction so that frequent small updates are not favored by hiding total overhead. Repack events and their trigger causes are overlaid. Every rejected compression is checked automatically to ensure that the published fallback digest equals the verbatim patched snapshot, not the previous version.

G.11 Exact Setup of the Reported Runs

This subsection records the supplied settings of the populated runs. Exact task identifiers, provider snapshots, and some adapter details require the accompanying experiment records; aggregate tables alone do not determine them.

Bundle tracks. Three tracks are used. (i) *Plain*: the evolved class skills assembled into a root with a routing list, one reference per class, one class as a nested subskill, and one locked helper script. (ii)

Grown: the same skills with evolution patches replayed into named branches, which re-append the output rules, the verification checklist and the accumulated pitfall list into every branch, plus one file of long guarded edge cases. A deterministic builder records the source span and patch identifier of every moved section and never rewords text. (iii) *Real evolved libraries*: three self-evolution runs stored with the project, each keeping all of its rounds as separate files, giving 17 files, 15 rounds, roughly 20,000 tokens, and 79–84% repeated text. Main cost tables use the grown track; Sections 5.2–D.5 use the real libraries.

Held-out task counts. Math 33, BFCL 21, SpreadsheetBench 48, giving 102 tasks pooled. The same tasks are used for every method within a benchmark, which is what makes the paired test valid. SpreadsheetBench is sampled at twice the density of the other two because its verdict is the strictest (a task passes only if every one of its test cases reproduces the gold range), so single-task flips dominate at small n; at $n = 24$ the ordering between the compressed and uncompressed bundles turned on one task, and doubling the sample removed that artefact without changing any method's configuration. BFCL runs with its offline search tool enabled; without the tool the skill provides no measurable benefit on that benchmark (no skill and the uncompressed library score identically), so that configuration cannot separate compression methods and is not used.

Compressor configuration. Structural extraction and structural audit (no model calls), capsule minimum 40 tokens, shortest repeated statement worth factoring 8 tokens, storage weight $\lambda = 0.05$, navigation locked, literal blocks locked, and pre-existing source reference defects tolerated but never introduced. Every method receives the identical source bundle and the identical environment contract.

Environment contract. Guarantees are only those the benchmark harness itself puts in front of the agent on every task, quoted from the harness: for math, "Solve the PROBLEM and end with the final result on its own line as `\boxed{...}`"; for BFCL, "Decompose it into ordered hops and resolve each one" and "end with a line EXACTLY in the form 'ANSWER: <short answer>' "; for SpreadsheetBench, "Read the workbook at path IN and write the modified workbook to path OUT", "Respond with ONE `run_python` block ... the variables IN and OUT are predefined", and "write general logic (do not hard-code values you can compute)". A statement the harness does not supply is never listed as a guarantee, because removing it would take away a rule the agent still needs.

Knowledge kept. A content unit is one behaviour-bearing line of the source (a bullet or sentence of at least a few words, excluding headings and fenced code). A unit counts as kept if its wording still appears, verbatim or lightly reworded, in a file the agent can reach by following links from the root; matching accepts either high local edit similarity or high local coverage of the unit's content words inside one window, so moving a qualifier is not scored as a loss while scattering words across unrelated files is. Units removed under the environment contract are reported separately as witnessed removals.

Continual replay. Rounds are appended one at a time in evolution order. All five schedules start from the identical state and see the identical rounds. Repacking runs every fourth round after compression is switched on. Drift is the relative difference in objective against rebuilding the whole library at that same round. Node-processing invocations (called "model calls" in the source replay record) are counted as one per changed textual node for a local update and one per textual node for a rebuild.

Start-time ablation. Compression is switched on at round $k \in \{1, 4, 7, 10, 13\}$, and for comparison never. Everything else is held fixed. The source start-time table defines this total as the sum of published library sizes. The earlier protocol calls it per-run cost; these are distinct ledgers, so the reported total is not treated as provider-billed runtime saving (Appendix H.4).

G.12 Compounding Analysis (Section 5.2)

The three findings behind the compounding effect are computed from the same full-resolution real evolved libraries used for the continual study, with no new model calls. For each round k we build the prefix holding the first k rounds, measure its raw shipped size $U(k)$, compress it one-shot with SkillZip Pro to size $C(k)$, and record both. The compression ratio at round k is $1 - C(k)/U(k)$ (Figure 4a). The repeated share of round k is $1 - \big(C(k) - C(k-1)\big)/\big(U(k) - U(k-1)\big)$: the numerator is the genuinely new text the compressor kept, the denominator is all text the round added, so their complement is the fraction of the round that was already present (Figure 4b). Reuse concentration counts, for each shared module SkillZip Pro created, how many strategy files link to

it, sorted descending (Figure 4c). We report the two libraries whose one-shot compression cleared the never-inflate check at every prefix; the third is excluded because at some prefixes its candidate did not beat the source and was republished verbatim, which would report a 0% ratio that reflects the safety guard rather than the redundancy structure under study. All numbers are averaged across the included libraries and are monotone in the same direction for each one individually.

H Additional Analysis

H.1 Why a Deployment-Only Objective Is Insufficient

Consider a bundle with a 200-token root, two rare references of 800 tokens each, and an exact 300-token fragment shared by those references. Each rare branch is used by 5% of tasks. Global factoring into the root reduces deployment by nearly 300 tokens (minus references), yet increases expected runtime exposure by approximately $0.9 \times 300 = 270$ tokens per task. Factoring into a shared on-demand module retains the deployment saving while charging the module only to the 10% affected paths. This example illustrates why deduplication scope is part of the representation, not an implementation detail.

The same reasoning applies to capsules. Moving a 600-token export guide behind a 20-token dispatcher at 10% frequency saves roughly $600 - (20 + 60) = 520$ expected path tokens before deployment weighting. At 95% frequency, the saving falls to 10 tokens and may disappear after reference overhead. The path distribution changes the best layout even when the underlying prose does not.

H.2 Sensitivity to Unknown Traffic

When traces are unavailable, uniform leaf-path weights can overvalue rare branches in a deep tree or undervalue a broad common branch. SkillZip Pro therefore exposes three policies: `uniform-leaf`, `guard-prior`, and `minimax`. The minimax policy accepts a transformation only when it improves J for every distribution in a declared uncertainty set. It is more conservative but avoids large regressions under traffic shift.

We recommend reporting both the optimization distribution and counterfactual distributions. A transformation that wins only under a narrow prior should be labeled workload-specific. Host-entailment witnesses are independent of traffic, while capsule and sharing placement are not.

H.3 Faithfulness Is Structural, Not Universal Semantics

Typed coverage protects explicit instructions and their scope, but natural language can carry implicature, tone, and redundancy that help a particular model. Removing redundant emphasis may affect behavior even if normative content is retained. This is why the empirical protocol measures non-inferiority across executor families and why a strict structural audit does not replace held-out evaluation in scientific validation. "Evaluation-free" describes the compression algorithm, not the evidence standard for publishing its effectiveness.

The audit can also share errors with extraction. Independent prompts, source-span provenance, deterministic type checks, and verbatim locking reduce correlated failure but cannot eliminate it. High-stakes skills should use human review or stronger formal specifications for critical obligations.

H.4 Source-record qualifications

The following qualifications distinguish reported measurements from protocol provisions and claims not supported by the available aggregate records. They identify which conclusions the present evidence supports and which details require the underlying experiment artifacts.

Benchmark provenance and adapters. The source names the mathematics suite LiveMathematicianBench (He et al., 2026), but describes a boxed-answer/symbolic-equality adapter; its task IDs and transformation details are not specified in the aggregate tables. This submission retains the recorded "Math/LiveMath" designation without claiming evaluation on an unchanged complete

public benchmark. The BFCL record similarly uses an offline-search/answer adapter and is not a full-leaderboard score. Exact split lists, model snapshot revisions, and tokenizer revisions must be recovered from the experiment artifacts rather than inferred from these aggregates.

Quality margin and denominators. The populated paired table uses -0.05, whereas the general protocol specifies -0.01. Pro's interval $[-.029, +.059]$ satisfies the former but not the latter. Macro scores average benchmarks; pooled scores average tasks and therefore have a different denominator. Failing a non-inferiority test does not establish statistically significant inferiority. The source records expanding the spreadsheet sample from 24 to 48 after observing sensitivity to a one-task ordering change; that history does not support calling every analysis preregistered.

Distinct cost ledgers are not reconciled by assumption. The grown main-cost aggregate (19.7% shipped, 14.3% mean run) differs from the full-method ablation row (21.1%, 15.8%). Their original values are retained separately. Likewise, the source cost plot labels its tail series P95 while the corresponding table labels a worst-path statistic. No unprovided trajectory is used to reconcile them. The workload table and the main run table retain their own aggregation definitions. The start-time table defines carried tokens as cumulative published library size, although a protocol passage calls it per-run cost; we report the artifact-exposure quantity, not a provider bill. The source's stated 9.7-point round-1 versus round-7 difference is an arithmetic mismatch; the populated values 40.8% and 34.4% differ by 6.4 points.

Conditional diagnostics and work accounting. The real-library One-Shot mean includes all three libraries, with one verbatim fallback; the prefix-compounding diagnostic includes only two that pass every prefix. The 55%±3% repetition estimate is a marginal compressed-size proxy, not semantic annotation. Replay "calls" count text-node processing; actual model calls in the structural efficiency table are zero. Production W_3 entails additional model work and is not assigned that zero-call cost.

Entries and views. The ordinary SKILLZIP row in the multi-entry study is also structurally valid; Pro's advantage is its larger saving at preserved entry fidelity, not exclusive validity. The audited .884 canonical-byte ratio and the no-audit .845 ratio refer to different entry guarantees. Transient canonical cost is unchanged, whereas per-entry compressed-view costs use their own closure denominators. Discoverability, average content, and average effective independence are distinct statistics.

Production comparison. The final 89% compressed accuracy must be paired with its 92% and 91% same-session controls, not the earlier 88% baseline. The resulting 2–3-point observed decrease is retained, and no significance claim is inferred without task-level outcomes. The pooled $n = 200$ counts session–task observations and need not mean 200 distinct examples. The source's combined unprotected row does not separately resolve interface layout for its structural and model-assisted cases; the production table leaves that field unspecified rather than inventing it.

Theory, implementation, and evidence. The placement derivation retains signed reference costs and assumes complete extraction and sound auditing. A passing lexical or model check is not itself a proof. The run matrix, multiple-testing rules, and seed counts are protocols, not additional results; this revision adds no experiments, annotation measurements, or uncertainty estimates.

Relation to alternative execution representations. Formal Skill, SkillRT, and Skill-to-LoRA pursue runtime or parameterized representations (Zhang et al., 2026; Chen et al., 2026; Zhang & Qi, 2026). Pro instead targets ordinary file-based deployment. MDL and grammar compression motivate reuse (Grünwald, 2007; Galbrun, 2022; Nevill-Manning & Witten, 1997; Larsson & Moffat, 2000); progressive loading additionally constrains where that reuse may be placed and which entry paths must retain access.