OPTIMIZING AGENT EXECUTION THROUGH JUST-IN-TIME SKILL COMPILATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent skills describe procedures that an LLM agent unfolds at run time into sequences of reasoning steps, tool calls, and observations. Much of this work recurs across runs, yet writing the procedure back into a skill as text still leaves the agent to carry it out step by step. We present a just-in-time compiler for skills that treats execution traces as profiling information, compiles recurring deterministic segments into parameterized scripts, and delivers them as an ordinary skill. Our central observation is that savings depend on the handoff: a script helps only if the agent calls it where it applies and trusts what it returns. The compiler therefore records each script's assumptions and tests them on subsequent seen inputs; a runtime interface discloses these assumptions on demand and returns a checkable execution summary. We evaluate on 20 SkillLearnBench tasks and a 100-task SpreadsheetBench subset, using held-out inputs. On SkillLearnBench our system raises correctness from 60% to 80% and lowers median delivery cost by 37.7% relative to bare agents. On SpreadsheetBench, it raises correctness from 72% to 78% and reduces median delivery cost by 24.7% relative to static compilation.

1 INTRODUCTION

Agent skills are written as text, but they are executed as work. A *skill* (Anthropic, 2025; Agent Skills, 2026)—a textual interface packaging user instructions, domain knowledge, and specialized expertise—can tell an agent how to review a contract, investigate a software failure, or conduct a literature search without programming every operation. The interface is expressive and cheap to author because all it omits is left to the agent: to execute a skill, the agent unfolds the compact description online into a long sequence of *thinking* $\rightarrow$ *tool call* $\rightarrow$ *observation* steps (Yao et al., 2022), and much of that sequence is deterministic computation the agent re-derives on every run.

This unfolding is costly in two ways. First, it is refreshed for every task: an agent repeatedly discovers how a repository is organized, debugs the same data conversion, or relearns the same environment-specific constraint—exploration and online debugging that earlier runs have already paid for. Second, existing tools are fine-grained: even when a recurring procedure is well understood, the agent reasons between each file read, query, and command, and every boundary is another model invocation and another opportunity for failure.

Writing the discovered procedure back into the skill as text need not remove this work. Supplied skills mostly leave pass rates unchanged (Han et al., 2026), and self-written skills did not repay their tokens in one cost study (Peng et al., 2026): a skill that *describes* a procedure can still leave the agent to carry it out step by step, and to pay for the description. Experience that is merely remembered is re-interpreted; only experience that is *compiled* is skipped.

Recent systems work touches the execution layer from two sides but closes no loop. PASTE mines recurring tool invocations from traces, yet uses them only to schedule speculation, not as capabilities the agent could invoke as a unit (Sui et al., 2026); and holding the available actions similar, the interface alone changes step count and token use (Xu et al., 2026c), yet those interfaces are designed in advance. The discovered units are read, not kept, and the better interfaces are built, not learned—although repeated traces contain exactly what closing the loop needs: which sequences recur, which procedures have stabilized, and which runs of small calls can safely become one.

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

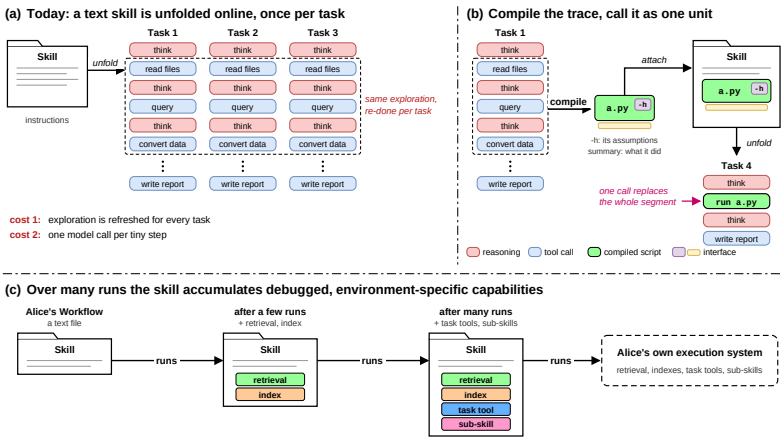


Figure 1: **Remembered experience is re-derived; compiled experience is skipped.** (a) A text skill, unfolded online once per task. (b) A script compiled from an earlier trace, with `-h` and a summary, called as one unit. (c) Capabilities the skill accumulates over many runs (Appendix H).

We introduce a just-in-time compiler for agent skills, built as two layers that close this loop. The **compile layer** treats a trace as profiling information: a compile session, itself an agent following a single compile skill, folds the deterministic segments the agent re-derived by hand into coarser parameterized scripts and records the assumptions each one makes. The **runtime layer** governs the *handoff*: the package reaches the model as an ordinary skill, an optional interface lets the model read a script's assumptions on demand and returns only a summary of its execution, and a hook at the end of every solving session starts the next fold. One observation organizes both layers: the saving is decided at the handoff, not at compilation—a compiled script pays off only when the model calls it where it applies and trusts what it returns; when it does not, compilation makes a run *more* expensive, as we observe on short tasks. The design follows from what a plain version of this loop got wrong (§2); under it, a skill evolves by acquiring an execution substrate debugged in the agent's own environment, not by appending memories to a prompt (Appendix H).

We hypothesize that agent experience should be *compiled*, not only remembered. Contributions:

- We show that agent execution recurs and that part of it is a candidate for compilation: reruns of a task yield markedly more similar action sequences than different tasks, and mechanical steps plus input inspection make up 28–54% of turn weight on SpreadsheetBench (SSB; Ma et al., 2024) and 13–26% on SkillLearnBench (SLB; Zhong et al., 2026) (§4.1).
- We show where a compiled package goes unused: the agent ignores, redoes, or re-checks scripts whenever the package's assumptions are invisible to it at the handoff, and a package compiled from one trace leaves them invisible (§2). A per-run account shows where the saving comes from and where it goes: cheaper runs hand the package the scripts the agent would otherwise write and the input it would otherwise inspect by hand; costlier runs pay for calling the package and coordinating around it, not for re-checking what it returns (§4.3, Fig. 5).
- We build a just-in-time skill compiler that closes this gap from both sides: the compile layer makes every script's assumptions explicit and re-tests them at each fold; the runtime layer discloses them, and the script's execution, at the moments the agent decides (§3, Fig. 4). On two benchmarks it turns multi-step tasks the bare agent fails into solved ones and reduces cost where the account predicts a gain (§4.2, Table 2); an interface ablation isolates the runtime layer (§4.4).

2 Observations: Where a Compiled Package Goes Unused

Whether a compiled package saves work is decided by the agent that receives it, at two moments of the *handoff*: before a call, whether the package applies to the input at hand; after it, whether the result can be taken as is. We write a package as

$$P = (S, H), \qquad H = \{(h_i, \sigma_i)\}_{i=1}^{m}, \qquad A_P(x) \;:=\; \bigwedge_i h_i(x), \tag{1}$$

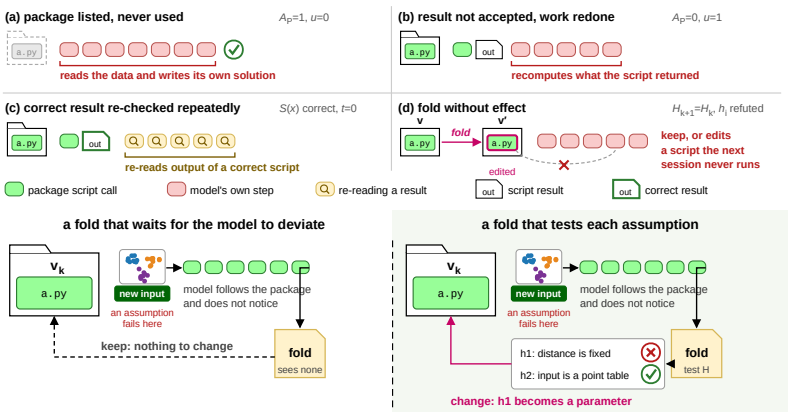


Figure 2: **Unexpected responses to a package, as mismatches at the handoff, and why a fold misses them.** Top: the four mismatches of §2 in the terms of Eq. (1). Bottom: on an input that refutes an assumption, a fold that waits for deviations versus one that tests each assumption.

where S is its scripts, each h_i an assumption on the input under which S is correct, and σ_i its source (task, seen input, compiler, or original skill). P *applies* to x when $A_P(x)$ holds.

The agent never evaluates A_P: it decides whether to call, $u(x) \in \{0, 1\}$, from the package's description $D(P)$, and whether to trust the result, $t \in \{0, 1\}$, from the script's output O; the handoff is right when $u(x) = A_P(x)$ and $t = \mathbf{1}[S(x) \text{ is correct}]$. A package compiled from one trace leaves both decisions blind: values of the seen input are baked into S, so $H \not\subseteq D(P)$, and O omits the execution that would justify t. Transcripts of a plain version of the loop, without the assumption rules of §3.1, show four mismatches (Fig. 2, top), each answered by a part of §3:

- **Ignored** (A_P=1, u=0): the package applies, but nothing in $D(P)$ shows it. The compiler makes H explicit (§3.1); the interface discloses it on demand (§3.2).
- **Redone** (A_P=0, u=1): a script runs outside its assumptions and returns a wrong result, found only by redoing the work; the same disclosure lets the model decline the script beforehand.
- **Re-checked** ($S(x)$ correct, t=0): the result is right, but O carries no evidence of how it was produced. The interface returns the execution step by step, so t can be decided from O (§3.2).
- **Fold without effect** (H_{k+1}=H_k): the new input refutes some h_i, yet a fold that waits for the model to deviate sees no reason to change. Each fold therefore tests H item by item on the new input and derives H_{k+1} from what was refuted (§3.3).

In a clustering task from SLB, each input names a different weighted distance. Static compilation wrote the first input's Euclidean formula into the search script without saying so; the model then ran that script on a Chebyshev input without deviating from the package, and on the held-out input only the sessions that happened to read the code were correct.

A fold that waits for the model to deviate therefore cannot be relied on to find such an assumption: on the Chebyshev input the model never deviated, so the fold had nothing to act on. This is why §3 states H and tests it against every new input (Fig. 2, bottom).

2.1 Case: a distance fixed at compile time

This appendix gives the clustering case of §2 in full.

The task is a clustering task from SLB (Fig. 3). Its three inputs ask for different distances: the first seen input a weighted Euclidean distance, the second a weighted Chebyshev distance, and the held-out input a weighted Manhattan distance (matching clusters to centroids is plain Euclidean in all three). Static compilation, following the first input, fixed the distance to Euclidean in both places the package computes it; the fold on the second seen input added a distance-form parameter without stating that the task sets the metric. On the held-out input, sessions that read the package source noticed the mismatch, bypassed the package and wrote their own Manhattan search, and were

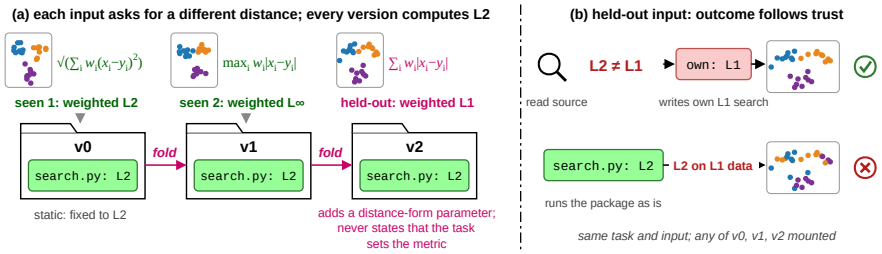


Figure 3: **On the held-out input, correctness follows trust, not the package version.** Packages of the plain loop of §2. (a) The three inputs of a clustering task ask for different weighted distances (formulas beside each input), but every version of the plain loop computes L_2. (b) Reading the source and bypassing the package succeeds; running it as is fails. Script names are illustrative.

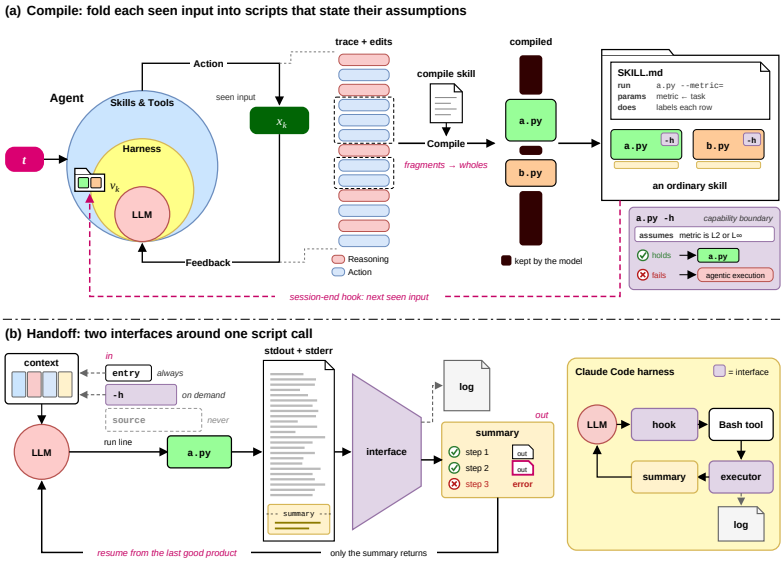


Figure 4: **The package is an ordinary skill: its scripts disclose their boundary on demand and return their process.** (a) The compile loop: scripts (coloured) and the model's judgment (dark); held-out inputs are only evaluated. (b) One call through the two interfaces, `-h` in and summary out. Right: the hook and executor that realise the second interface in an unmodified host.

correct; sessions that trusted the package and ran it were not. Whether a package passes on this input therefore reflects whether the model trusted it, not which version it was. These packages come from the plain loop, without the assumption rules of §3.1; the `-h` entry quoted in §3.2 is from a package compiled under those rules, which names both seen distances and excludes any other.

3 Just-in-Time Skill Compilation

In the terms of §2, the method builds packages whose assumptions H are explicit, disclosed at the two moments the model decides, and revised on every input the package meets. We borrow the structure of a just-in-time compiler (Kotzmann et al., 2008; Suganuma et al., 2005), as SkVM does for skills (Chen et al., 2026): a task is first solved by interpretation while its trace is recorded, the trace's deterministic segments are compiled into scripts, and because compiled code is valid only under assumptions drawn from what was profiled, those assumptions are checked at use and the code recompiled when they fail. The analogy is structural: a package's assumptions are semantic, not machine-checkable, and its caller is a model (Fig. 4). A *package* has the shape of an ordinary skill—an instruction file plus a directory of scripts—reaches an unmodified host through its own skill mechanism, and is not tied to the model that compiled it (cf. Cai et al., 2024).

3.1 COMPILING A TRACE INTO A PACKAGE

Profiled as deployed. A fold profiles the package the way it will be used: the model solves the next seen input with v_k mounted as at evaluation, and when that session ends, a hook starts a compile session—itself an agent—that turns the transcript and the model's edits into v_{k+1}. The first version v_0 is compiled without a trace, from the task statement, the seen inputs and the original skill. Each v_k is thus a pair (S_k, H_k) tested on k seen inputs; held-out inputs never reach the compiler.

From fragments to wholes. A classical JIT selects hot paths by execution count; our target is the deterministic segment the model re-derived by hand, which one trace reveals. The compiler merges such segments into as few and as whole scripts as it can, so that one call replaces a segment rather than a step, and exposes whatever may vary with the task as a parameter: a value that differs across seen inputs is never fixed in code. Each script checks results by value, not only format, and reports the quantities the answer is built from, so the caller can judge a result without recomputing it.

Speculation is recorded. One trace cannot tell the task from the input: the first input's distance or sheet layout looks like part of the task. The compiler therefore writes down every assumption a script makes with its source—task statement, original skill, seen input, or its own choice—so that an assumption drawn from one input is visible as such and tested when the next arrives. A value that only a seen input supports becomes a parameter or a stated assumption, never a silent constant.

3.2 THE INTERFACE: DISCLOSURE IN, PROCESS OUT

Into compiled code: progressive disclosure. The interface puts H into $D(P)$ on demand and the script's execution into O. A speculative JIT guards optimized code with executable checks and falls back to the interpreter when one fails (Kotzmann et al., 2008). A package's assumptions are semantic and cannot be enumerated from a few traces, so no guard is both sound and complete; we move the check to the caller, which can read, and disclose the boundary in tiers, as skills themselves are loaded (Agent Skills, 2026), so that it costs context only when a decision needs it. The instruction file lists each script's run line, parameters with their sources, and one sentence on what it does. The *boundary* sits behind the script's `-h`: its steps and products, and for each assumption its source, what to do when it fails, and how a misuse would show. In the clustering case, `-h` names the two distances the script implements, with sources, so an input asking for a third can be declined from the description (after two folds; translated, identifiers and formulas verbatim):

```
[run_grid_search-a3]
 assumption  The custom distance is chosen by --metric, w = shape_weight:
             scaled_euclidean  d(a,b) = sqrt((w*dx)^2+((2-w)*dy)^2)
             scaled_chebyshev  d(a,b) = max(|w*dx|,|(2-w)*dy|)
             The seen statements differ, so it is required, with no default.
 source      task statement; scaled_euclidean from seen input 1,
             scaled_chebyshev from seen input 2
 if it fails the statement's formula is neither: the script does not apply; follow the steps
 misuse      -- (a wrong formula still runs with no sign in the summary; confirm first)
```

Out of compiled code: the process, not the output. A JIT runtime trusts its translation once the guards pass; the model has no such guarantee, and when it cannot see how a result was produced it recomputes it by hand. Every script therefore prints a summary, even on failure: each key step with its product and values, or the last good step and the error. A hook routes each package call through an executor that logs the full output and returns only the summary; a short runtime note, loaded as a skill, asks the model to take the summary's values as given, resume from the last good product after a failure, and accept a success unless it shows a stated sign of misuse. Without the interface, packages are compiled without `-h` and summary and run without hook and note.

3.3 CLOSING THE LOOP: FOLDS THAT TEST THE BOUNDARY

The fold is the guard. Each fold tests every assumption in H_k against the new input, whether or not the model deviated. A JIT deoptimizes on a failed guard (Hölzle et al., 1992); nothing plays that role here: the model often follows the package even where an assumption fails (§2). In the clustering case, testing the distance assumption on the Chebyshev input shows that the task sets the metric, so the fold makes it a parameter. The fold revises H before the scripts, and extends the instruction file only where a boundary moved, so it grows with what was learned, not with every fold.

Edits locate the boundary. A fold reads the model's edits to a script as evidence about H, not as fixes to copy. An edit that repairs a general fault, such as a crash, enters the package as code; one that fits the script to the current input only, such as swapping in its column name, marks an assumption as refuted, which becomes a parameter or a new boundary (Appendix J).

4 EXPERIMENTS

Setup. We evaluate on 120 tasks from SkillLearnBench (SLB) and SpreadsheetBench (SSB) (Zhong et al., 2026; Ma et al., 2024). Packages are compiled on a task's seen inputs, and every comparison runs on a held-out input. Compilation and solving run in an unmodified Claude Code host with `claude-opus-5`. Seven conditions differ only in what the agent receives: **bare** (no skill); **static** with and without the interface (the package v_0, compiled with no trace); **native** v_1, v_2 (the package after one and two folds without the interface); and **interface** v_1, v_2 (the same folds with the interface on). Native against static isolates what traces add; interface against native isolates the runtime layer. SkVM, SkillSmith and skill-creator (Chen et al., 2026; Xu et al., 2026b; Anthropic, 2026) run from their authors' code under the same host and model. Correctness is the official grader; Table 2 reports median priced model usage over available runs so every system shares one cost definition, while paired cost ratios are reserved for within-system analyses. The analyses of §4.3 and §4.4 read mini20, a random 19-task subset, and only explain the full-set results. See Appendix C.

4.1 Q1: IS THERE ANYTHING TO COMPILE?

Over three bare gpt-6-astra runs of each of 398 SSB and 20 SLB tasks, reruns of a task yield markedly more similar action sequences than different tasks (Table 1; Appendix D). The gap appears at both granularities: on SSB, same-task similarity exceeds different-task similarity by 0.17 with coarse action classes and 0.15 when tool and argument shape are retained; on SLB, the gaps are 0.25 at both granularities. Repeated work is therefore not an artifact of masking arguments or collapsing tools into broad classes.

Table 1: **Repeated execution and the candidate set for compilation.** Entries are median LCS similarities of main-session action sequences; each cell gives coarse/fine granularity. Candidate is the lower–upper share of turn weight formed by mechanical work, with input inspection included only in the upper definition.

Benchmark	Same task	Different tasks	Coarse gap	Candidate (%)
SSB	0.833 / 0.750	0.667 / 0.600	0.166	28.31–54.26
SLB	0.690 / 0.543	0.445 / 0.295	0.245	12.94–26.15

The form of repetition differs by benchmark. On SSB, 99 coarse templates describe 1,191 measurable runs, and the five most frequent cover 74.73%; one read–script–reply template alone accounts for 591 runs. SLB has 59 templates over 60 runs, so reuse is local rather than a whole repeated trajectory. Even under this harder regime, mechanical work plus input inspection occupies 12.94–26.15% of turn weight, against 28.31–54.26% on SSB. These are candidate shares, not predicted savings; whether the compiler realizes them is what follows.

4.2 Q2: DOES COMPILATION MAKE AGENTS MORE CORRECT AND CHEAPER?

Table 2 gives the main result. On the full SLB set, folding raises correctness from 60% for bare and 70% for either static package to 80%, the highest value in the table. JIT with the interface has a median delivery cost of \$0.663, below bare (\$1.064), SkVM (\$1.146), and SkillSmith (\$1.634). Skill-creator is cheaper at \$0.540 but solves only 45% of tasks. JIT attains the highest correctness while remaining among the least expensive systems.

The SLB ablations separate this improvement from merely adding a package. Without the interface, moving from the static package to its folded JIT version raises correctness from 70% to 80% while reducing median cost from \$0.848 to \$0.762, a 10.1% reduction. With the interface, the same transition again raises correctness by ten points and reduces median cost from \$0.946 to \$0.663, a 29.9% reduction. The interface alone is therefore not uniformly cheaper: it increases cost for

the static package, but the lowest-cost JIT variant on SLB is the one that exposes the structured interface. Folding making the interface useful at delivery time, so that the solver can consume a compact operation rather than rediscovering the procedure from the package contents.

Table 2: **Main results.** Higher correctness and lower cost are better. $+I$ denote delivery with the runtime interface.

		Main experiment					External baselines		
Benchmark	Metric	Bare	Static	Static $+I$	JIT	JIT $+I$	SkVM	SkillSmith	skill-creator
SLB ($N = 20$) full set	Correct (%)	60	70	70	80	80	70	70	45
	Cost (USD)	1.064	0.848	0.946	0.762	0.663	1.146	1.634	0.540
SSB ($N = 100$) subset	Correct (%)	72	78	75	78	76	65	78	76
	Cost (USD)	0.313	0.465	0.489	0.350	0.356	0.114	0.475	0.376

The external baselines clarify the trade-off beween accuracy and cost. Neither SkVM nor SkillSmith exceeds 70% on SLB, despite median costs above bare and above both JIT variants. Skill-creator has the lowest median cost among the SLB baselines, but its 45% correctness is 35 points below JIT. Thus, on the full SLB set, both JIT variants occupy the strongest observed accuracy level, and JIT with the interface does so at lower cost except the substantially less accurate skill-creator baseline.

On the 100-task SSB subset, static and JIT without the interface reach 78%, six points above bare, at median costs of \$0.465 and \$0.350. Skill-creator reaches 76% at \$0.376, while SkillSmith matches 78% at \$0.475; SkVM is cheapest (\$0.114) but solves 65%. Unlike SLB, every compiled condition in our main experiment costs more than bare (\$0.313), consistent with handoff overhead dominating these short spreadsheet runs. The contrast motivates the per-run account in §4.3.

Folding nevertheless removes a substantial part of that overhead. For the no-interface condition, it preserves 78% correctness while lowering the static package's median cost from \$0.465 to \$0.350, a 24.7% reduction. With the interface, folding raises correctness from 75% to 76% and lowers median cost from \$0.489 to \$0.356, a 27.2% reduction. The remaining gap to the cost of bare agentic traces shows that saved solver work does not always repay the fixed cost of invoking and handing off to the compiled package on short spreadsheet tasks.

SSB therefore presents a tighter frontier than SLB rather than a single dominant system. JIT without the interface matches SkillSmith's 78% correctness at a 25.1% lower median cost. SkVM minimizes cost but trails JIT without the interface by 13 correctness points. Across the two benchmarks, the main table supports a conditional conclusion: folding consistently lowers the cost of the corresponding static package, while its end-to-end advantage depends on whether the task is long enough for the saved interaction to outweigh package handoff.

4.3 Q3: Where does the saving come from, and where does it go?

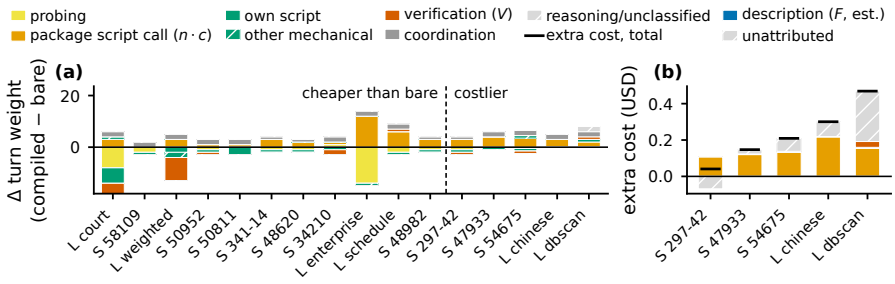


Figure 5: **Cheaper runs hand the package work the agent did by hand; costlier runs pay for calling it.** (a) Change in turn weight by turn class, compiled run (interface, v_2) minus bare, on each pair both solve, ordered by cost ratio (L: SLB, S: SSB). (b) Extra cost of each costlier pair by term of Eq. (2); hatched: unattributed; tick: the host's total (Appendix E).

We read the terms of the per-run account of Appendix I—work the scripts remove (R), re-checking (V), call overhead ($n \cdot c$), and the description (F)—off the pairs both conditions solve (Fig. 5;

Appendix E). In the cheaper runs the package's scripts take over work the bare agent did by hand, mostly the turns it spent writing its own scripts, often with input inspection; what is saved is the write–run–fix sequence a call replaces, not mechanical turns as such. The costlier runs are all short, but so are most cheaper runs on SSB, so length alone does not decide the sign. Their extra cost is neither verification nor the description, but the calls and the coordination around them, loading the skill first of all, on runs the bare agent finishes in a few turns. The sign is thus set task by task by how much of the bare run the scripts replace, against the cost of bringing the package in.

4.4 Q4: Does the runtime layer matter?

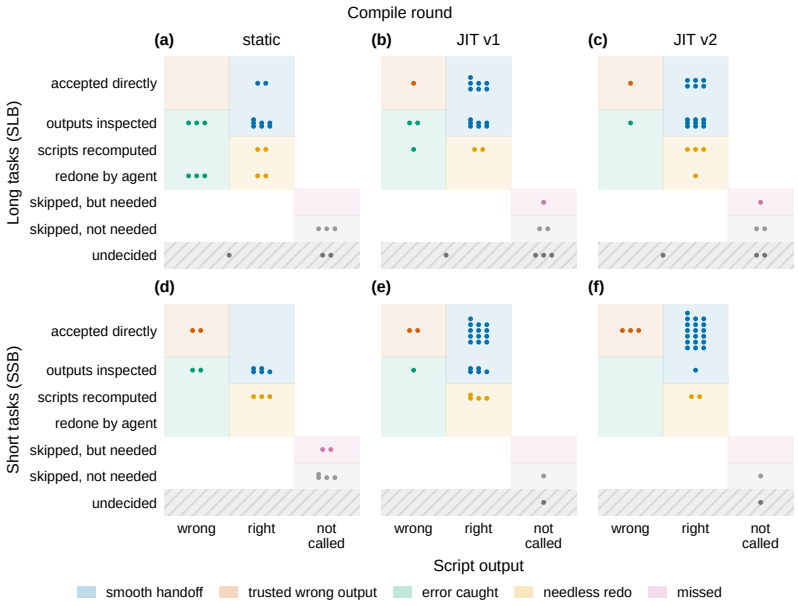


Figure 6: **Across compile rounds, the agent hands work to the package instead of redoing it.** One dot per package script per trace, interface off. Columns: the script's output; rows: what the agent did with it (labels in Appendix F); hatched row: undecided.

Beyond cost and correctness (Table 2), we ask what the runtime layer changes in the two decisions of §2, u and t. We label every package script in the 114 mini20 traces under six conditions—three compile rounds, interface off and on, one run per cell[1]—by its output and by what the agent did with it; the crossings are the mismatches of §2 (Appendix F). The unit is the script call and the trace, not the task, which is enough to describe how the agent treats a package.

Folds move work into the package. Fig. 6 holds the interface off across rounds. On short tasks the static package is barely used as delivered—no right output is accepted directly—whereas after one and two folds 13 and 19 are; per trace, package runs go from 1.6 to 3.3 and verifications before use from 1.0 to 0.3 (Fig. 7a), while wrong outputs accepted directly do not grow (2, 2, 3). On long tasks the direction is the same but the movement small, and inspecting the output remains the common response. A fold changes not only the scripts but what the agent does with them.

The interface changes what is trusted. Pooled over rounds (Fig. 7b), the interface lowers how often a wrong output is accepted directly: 0 of 10 against 2 of 12 on long tasks, and 2 of 12 against 7 of 10 on short ones. Redone right outputs and skipped needed scripts move the same way, but with independent packages and one run per cell we do not read them as differences (Appendix F). The interface targets exactly this decision: an execution summary lets the agent decide t from evidence.

[1]The solving traces use Opus 5; behavior labels are assigned by a separate read-only model, gpt-6-astra. Counts are not comparable with Table 2.

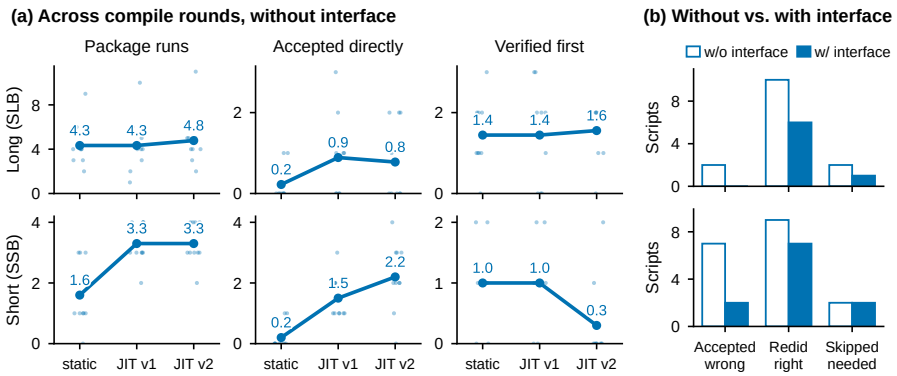


Figure 7: **As folds go on the agent runs the package more and checks it less; with the interface it accepts fewer wrong outputs.** (a) Per-trace counts, interface off, by compile round; one dot per trace, line at the mean. (b) Outcome rates pooled over rounds, interface off vs. on.

5 Related Work

Agent execution: the loop is optimized as given. Speculative tool execution (Sui et al., 2026) and TokenPilot (Xu et al., 2026a) overlap or cheapen turns, a better interface cuts the turns a fixed action set takes (Xu et al., 2026c), and SoL-Pi (Liu et al., 2026) searches traces for harness rules. All are fixed once for every task; we compile a task's recurring segment into one call.

Self-improving agents: experience is stored, not executed. Reflection, memory, and self-evolving harnesses re-inject feedback, insights, workflows, or trajectories, or adapt prompts and tools (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2024c; Kang et al., 2026), or, like skill-creator (Anthropic, 2026), write them into a skill; most treat useful experience as *knowledge* the agent acts on once reminded; we compile its procedural part into code the agent calls.

Skill compilation: the boundary is fixed, and left implicit. SkVM (Chen et al., 2026), Trace-Compiler (Yadouni & Li, 2026) and SkillSmith (Xu et al., 2026b) turn skills or traces into code but fix the boundary between model and code at compile time; Skill-DisCo (Guo et al., 2026) lets the model pick a distilled skill, then invokes it as a single call (Table 3, Appendix A). We leave every call to the model and give it what the decision needs; starting from a trace, not a skill file, compilation also applies where no skill exists, as on SpreadsheetBench.

6 Discussion and Limitations

We claim that compiled scripts remove deterministic work only if the handoff goes right; that explicit assumptions, visible at the two decisions, let the two layers realize a saving on most tasks that both conditions solve; and that a per-run account shows where it comes from and where it goes—not a large saving on every task, monotone gains across folds, or transfer to any input.

Limitations are discussed in Appendix G.

7 Conclusion

Compiled scripts save work only if the model calls them where they apply and trusts what they return, and both fail when the package's assumptions are invisible. The compiler makes them explicit and keeps testing them; the interface discloses them where the model decides. We regard the handoff, not the compiler, as the open problem for agents that learn from their own execution.

AI USE STATEMENT

The authors conducted the research and designed the experiments. An LLM coding agent implemented much of scripts from author-written specifications. The authors reviewed and checked script-derived results. An LLM assistant helped search the literature and polish the paper; the authors checked every citation against its source. The authors reviewed all AI-assisted output and take responsibility for the final text, claims, and artifacts. Generative AI is also the object of study: we evaluate an LLM-driven coding agent and model-generated artifacts; §4 reports the host and models.

ETHICS STATEMENT

This work evaluates coding agents on public benchmarks. It involves no human subjects and no personal data. The agents execute model-generated code as part of every run.

REPRODUCIBILITY STATEMENT

Appendix C gives the benchmarks, the task-selection rule, the assignment of seen and held-out inputs, the host and model versions, and the price table used to convert usage into cost. Appendix D gives the fixed rule set behind the turn classification of §4.1; Appendix E states how each term of Eq. (2) is read off a trace. The source code and experimental artifacts will be released upon publication.

REFERENCES

Agent Skills. Agent Skills specification, 2026. URL https://agentskills.io/specification. Accessed 2026-09-26.

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. EvoSkill: Automated skill discovery for multi-agent systems. *arXiv preprint arXiv:2603.02766*, 2026.

Anthropic. Equipping agents for the real world with Agent Skills, 2025. URL https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills. Engineering blog post, 16 October 2025.

Anthropic. skill-creator, 2026. URL https://github.com/anthropics/skills/tree/main/skills/skill-creator.

John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *International Conference on Learning Representations*, 2024.

Le Chen, Erhu Feng, Yubin Xia, and Haibo Chen. SkVM: Revisiting language VM for skills across heterogenous LLMs and harnesses. *arXiv preprint arXiv:2604.03088*, 2026.

Zhongxin Guo, Danrui Qi, Hanwen Gu, Peng Cheng, and Yongqiang Xiong. SKILL-DISCO: Distilling and compiling agent traces into reusable procedural skills. *arXiv preprint arXiv:2606.26669*, 2026.

Tingxu Han, Yi Zhang, Wei Song, Chunrong Fang, Zhenyu Chen, Youcheng Sun, and Lijie Hu. SWE-Skills-Bench: Do agent skills actually help in real-world software engineering? *arXiv preprint arXiv:2603.15401*, 2026.

Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pp. 32–43, 1992.

Borui Kang, Jinrui Gu, Junhan Lv, Wenbin Li, Lei Wang, and Yang Gao. Harness continual learning: Continual adaptation beyond model parameters. *arXiv preprint arXiv:2608.19013*, 2026.

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):1–32, 2008.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, et al. SkillsBench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026.

Haozhe Liu, Tian Ye, Sensen Gao, Qihang Cao, Yitong Li, Mingchen Zhuge, Duomin Wang, Ruihua Zhang, Ping Luo, Jiawang Bian, et al. SoL-Pi: Recursively scaling auto-research loops for efficient agent harness. *arXiv preprint arXiv:2609.20519*, 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. *Advances in Neural Information Processing Systems*, 37:94871–94908, 2024.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2Skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026.

Zhiyuan Peng, Xin Yin, Chenhao Ying, Zhe Cui, Zixiang Ding, Zhenhua Liu, Jiang Wu, and Yuan Luo. EvoClawBench: Can agents learn reusable skills from their own runs? *arXiv preprint arXiv:2607.09711*, 2026.

Yifei Shen, Bo Li, and Xinjie Zhang. SkillOpt-Lite: Better and faster agent self-evolution via one line of vibe. *arXiv preprint arXiv:2607.03451*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):732–785, 2005.

Yifan Sui, Han Zhao, Rui Ma, Zhiyuan He, Hao Wang, Jianxun Li, Kaiqiang Xu, Kai Chen, and Yuqing Yang. Parallelizing tool execution and LLM generation for low-latency agent serving. *arXiv preprint arXiv:2603.18897*, 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. *arXiv preprint arXiv:2305.16291*, 2023.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. *arXiv preprint arXiv:2402.01030*, 2024a.

Zhiruo Wang, Daniel Fried, and Graham Neubig. TroVE: Inducing verifiable and efficient toolboxes for solving programmatic tasks. *arXiv preprint arXiv:2401.12869*, 2024b.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024c.

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

Tianxin Wei, Zhan Shi, Minhua Lin, Bing He, Zewen Liu, Yisi Sang, Yuanchen Bei, Xuying Ning, Jiaru Zou, Ting-Wei Li, et al. Evo-Harness: Context-to-harness skill compilation for self-evolving agents. *arXiv preprint arXiv:2608.15071*, 2026.

Buqiang Xu, Zirui Xue, Dianmou Chen, Chenyang Fu, Chiyu Wu, Caiying Huang, Chen Jiang, Jizhan Fang, Xinle Deng, Yijun Chen, et al. TokenPilot: Cache-efficient context management for LLM agents. *arXiv preprint arXiv:2606.17016*, 2026a.

Duling Xu, Zheng Chen, Zaifeng Pan, Jiawei Guan, Dong Dong, Jialin Li, and Bangzheng Pu. SkillSmith: Compiling agent skills into boundary-guided runtime interfaces. *arXiv preprint arXiv:2605.15215*, 2026b.

Xiangzhe Xu, Hamidreza Saghir, Qianhui Wu, Marc-Alexandre Côté, Tong Wang, Kiran Lakkaraju, Kexin Pei, and Xiangyu Zhang. The devil is in the interface: Evaluating how tool architecture shapes coding agent behavior. *arXiv preprint arXiv:2608.11386*, 2026c.

Salma El Yadouni and Guanyi Li. TraceCompiler: Skill-guided mining and compilation of LLM agent traces into mostly deterministic workflows. *arXiv preprint arXiv:2608.02680*, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. SkillOpt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Boqin Yuan, Yue Su, Renchu Song, Sen Yang, and Jing Qin. ClawTrace: Cost-aware tracing for LLM agent skill distillation. *arXiv preprint arXiv:2604.23853*, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Shanshan Zhong, Yi Lu, Jingjie Ning, Yibing Wan, Lihan Feng, Yuyi Ao, Leonardo FR Ribeiro, Markus Dreyer, Sean Ammirati, and Chenyan Xiong. SkillLearnBench: Benchmarking continual learning methods for agent skill generation on real-world tasks. *arXiv preprint arXiv:2604.20087*, 2026.

Deyao Zhu, Xin Zhou, Shengling Qin, Xuekai Zhu, Hangliang Ding, Shu Zhong, Zixin Wen, Zhonglin Xie, Chenhui Gou, Linxuan Ren, et al. EdgeBench: Unveiling scaling laws of learning from real-world environments. *arXiv preprint arXiv:2607.05155*, 2026a.

Kan Zhu, Mathew Jacob, Chenxi Ma, Yi Pan, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. TraceLab: Characterizing coding agent workloads for LLM serving. *arXiv preprint arXiv:2606.30560*, 2026b.

A EXTENDED RELATED WORK

We group prior work by what each line assumes and this paper drops. Numbers quoted from other papers are those papers' own claims.

Agent execution and tool granularity: the loop is the target. Agent execution is commonly a serialized loop in which generation alternates with tool execution. PASTE attacks the serving-level latency of this loop by predicting future tool calls from recurring patterns and executing them speculatively while generation continues (Sui et al., 2026). Its results establish that repeated structure in trajectories supports execution-time optimization, and its mined patterns (an edit followed by a test run, say) point to an implicit granularity above individual calls, which we call emergent macro-actions; they are never presented to the agent as such. That interface design shapes agent behavior was shown by SWE-agent's agent–computer interface (Yang et al., 2024); Xu et al. hold available information and actions similar while varying how they are organized and exposed, and show that a CodeAct-style interface (Wang et al., 2024a) reaches similar task performance to a bash-only interface with 41.6% fewer steps and 56.3% lower token usage (Xu et al., 2026c). These two results are the two opportunities of the introduction: the compile layer materializes the granularity PASTE's patterns point to, and the runtime layer treats the interface of a compiled package as the variable Xu et al. show it to be. A second group works on what each turn carries. TokenPilot manages context with the prompt cache in mind (Xu et al., 2026a); TraceLab characterizes coding-agent workloads, including how often context is compacted (Zhu et al., 2026b); SoL-Pi runs automated research loops over execution traces to find harness rules, including one that fuses two actions into a single round trip, and reports a harness that moves fewer tokens than the Pi harness, at a score it reports as comparable, on EdgeBench tasks kept outside its search loop (Liu et al., 2026; Zhu et al., 2026a); ClawTrace attributes cost to each step of a trace, and its companion, CostCraft, uses this to emit preserve, prune, and repair patches to a distilled skill (Yuan et al., 2026). These systems optimize the loop, or fix harness rules once for all tasks, and compose with compilation, which removes a task's own recurring segment of the loop altogether. Their savings come from the context and harness layers, so we never place their numbers beside ours.

Self-improving agents: experience is stored, not executed. Reflection and memory systems retain verbal feedback, insights, induced workflows, or past trajectories and re-inject them into later prompts (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2024c), or retrieve verified code skills into them (Wang et al., 2023); self-evolving harnesses adapt prompts, tools, or policies from accumulated runs (Kang et al., 2026). Most of these systems assume that the useful part of experience is *knowledge*, and that the agent can act on it once reminded. We treat the deterministic majority of a trace differently: its useful part is a *procedure*, which we compile into code the agent calls rather than describe for the agent to re-derive. Two lessons from this line shape our design. Voyager's library is ever-growing: each verified program is added, and the paper describes no mechanism that merges or consolidates programs; TroVE periodically trims rarely used functions from its toolbox (Wang et al., 2024b); a fold instead rewrites the package, merging fragments into wholes. And a program verified only by the model that wrote it inherits that model's blind spots; our packages pass mechanical checks that sit outside the compile session. Self-evolving harnesses are complementary: they evolve the harness around the model, whereas we leave the host unmodified and change only what it is given, and we discuss them rather than evaluate them. Closest among them, Evo-Harness distills single executions into reusable skill harnesses, written as natural-language guidance, around a frozen model (Wei et al., 2026); we instead compile the executed procedure into scripts delivered as an ordinary skill, and measure what they remove from the next run.

Skill learning and evolution. A fast-growing line produces or improves skills as text. Skill benchmarks measure what such skills buy. SkillLearnBench (Zhong et al., 2026) finds that nearly all generated skills are pure instruction, whereas a third of human-written ones carry scripts. SkillsBench (Li et al., 2026) pairs runs with and without curated skills across domains; SkillSmith's headline token saving on it is measured against an agent using the raw skill; its no-skill run is reported but set aside because it solves one task fewer. SWE-Skills-Bench (Han et al., 2026) finds that most supplied skills leave the pass rate unchanged, and EvoClawBench (Peng et al., 2026) that, on a focused cost subset, the token cost of self-authored skills does not pay back. Skill optimizers edit skill text against a validation score: SkillOpt proposes bounded edits and keeps those that improve it

(Yang et al., 2026); SkillOpt-Lite, a different work by different authors, strips this down and reports that a coding agent simply exploring raw trajectory files is already a strong optimizer (Shen et al., 2026); EvoSkill refines skills from failure analysis (Alzubi et al., 2026). Anthropic's skill-creator writes and revises a skill from test runs (Anthropic, 2026), and we run it as a baseline for writing experience down as a skill. Trace2Skill (Ni et al., 2026) distills many trajectories of a domain into one transferable skill and reports only accuracy in its main results; on SpreadsheetBench it evolves the skill on one half and tests on the other, loading the skill directly into context without retrieval. It answers a different question—what to know across a domain—whereas we ask how to execute recurring work on one task more cheaply, and its skills could serve as the original skill of our static compilation. Closest in spirit, agent skill induction (Wang et al., 2025) has web agents induce skills as programs verified by execution. We share the program form; what we add is that compiled work is delivered through the host's own skill mechanism, refolded from traces produced under that delivery, and judged by what it removes from the next run. Using a coding agent as the compiler is therefore not our contribution—SkillOpt-Lite shows it is a strong default—but the compile skill that governs it, the form of the package, and the measurement of the handoff are.

Skill compilation and trace solidification. Four systems compile skills or traces into code; Table 3 compares their mechanisms. None is a pure hard switch, so we compare them by *who decides the handoff, and when*. SkVM (Chen et al., 2026) compiles a skill ahead of time into variants for a target model and harness, together with a code signature and a parameterized template, and at run time solidifies code: once the code the agent generates matches that signature on multiple consecutive invocations, later invocations bypass the model and call the instantiated function with parameters extracted from the task context. Its evaluation shows solidification on two weather and two PDF-extraction tasks; since parameters come from the task context rather than from tool output, we read the mechanism as suited to short, single-operation tasks. TraceCompiler (Yadouni & Li, 2026) clusters traces by intent and compiles a cluster only once it clears a support floor of traces (five in its experiments), into a typed workflow in which choices the traces leave open remain model nodes; a similarity router dispatches a matching request to the workflow, and by its own account it has no general executor for its intermediate representation. SkillSmith (Xu et al., 2026b) must start from an existing skill package, which it compiles without any trace into a boundary contract that hands any step it cannot execute back to the agent as guidance; the agent selects it and keeps driving. In all three the boundary between model and code is fixed at compile time and does not change at run time. Skill-DisCo (Guo et al., 2026) induces skills across hundreds of tasks, verifies them by execution on held-out tasks, and lets the model select a skill at run time; a selected skill is then invoked as a single call. We hand off step by step at run time: a script that cannot proceed returns to the model, which completes the step and may hand back to the next script, and the boundary is drawn by the profile of the package's own delivery. A single trace suffices for a fold, and the package needs no original skill, which SpreadsheetBench does not ship. Compiling without any trace is our *static* control. SkVM and SkillSmith are end-to-end baselines in our experiments; we found no public code for Skill-DisCo or TraceCompiler, so we compare them at the level of mechanism rather than end to end. Following the field's practice we compare *which steps are moved* across systems, not token counts reported under other harnesses and models.

Classical JIT compilation, and what we borrow. Our design is inspired by JIT compilers (Aycock, 2003; Kotzmann et al., 2008; Suganuma et al., 2005): the compile layer plays the JIT compiler, profiling execution and compiling what recurs, and the runtime layer plays the transitions between interpreted and compiled code. Four departures are deliberate: the check on an assumption is made by the model reading the package, not by an executable guard whose failure deoptimizes the code (Hölzle et al., 1992); the compilation target is the step the model re-derived by hand, not the most frequently executed path; the compiler is itself a model following a skill; and acceptance of a compiled result is not guaranteed by the machine, so the package must earn the model's trust—a problem a JIT runtime never has to solve, and the one §2 observes.

B MECHANISM COMPARISON WITH TRACE-SOLIDIFICATION SYSTEMS

Table 3 compares our design with the closest systems on where compilation starts, who decides the handoff and when, what happens to the model once an artifact exists, and how artifacts are validated

and updated. Entries follow each paper's own description; "not stated" means the paper does not describe that mechanism.

Table 3: **Mechanism comparison.** Entries for prior systems are taken from the cited papers, except the public-code column, which records what we could locate as of September 2026. †The paper cites a released artifact but gives no URL.

	Starts from	Handoff decided by, and when	Model once an artifact exists	Validated before use by	Updated when	Public code
SkVM (Chen et al., 2026)	a compiled signature that generated code matches on consecutive invocations	compile time; matched code generation is replaced	bypassed on the solidified invocation	signature match on consecutive invocations	failure re-enables model-side generation	yes
TraceCompiler (Yadouni & Li, 2026)	an intent cluster above a support floor of traces	compile time; open choices stay model nodes	replaced for the matched request	schema and structural checks, bounded repair	no workflow versioning (stated as a limitation)	no URL†
Skill-DisCo (Guo et al., 2026)	successful traces across hundreds of tasks	run time, by the model; a selected skill is invoked as a single call	remains the outer controller	execution on held-out tasks; postconditions	not stated	none found
SkillSmith (Xu et al., 2026b)	an existing skill package; no trace	compile time; steps outside the contract return to the agent as guidance	keeps driving; the runtime returns blocked, guidance, or execute	structural checks; a recorded evidence level	the source package changes	yes
Ours	task statement and seen inputs (and the original skill if any), then one trace per fold	run time, step by step; a script that cannot proceed returns to the model	reasons between and after script calls	mechanical checks	every fold, on a trace produced under the current package	

C BENCHMARKS, TASK SELECTION, AND HOSTS

This appendix states what a replication needs: which tasks were used, how their inputs were split, what ran the agent, and how cost was computed.

Benchmarks and task selection. We use SkillLearnBench (SLB) and SpreadsheetBench (SSB). SLB comprises 20 tasks across 15 sub-domains, 100 instances in all, each task with a human-authored skill (Zhong et al., 2026); SSB comprises 912 instructions and 2,729 test cases, about three per instruction (Ma et al., 2024). The SSB evaluation set contains 100 task identifiers drawn from the 400 identifiers in `spreadsheetbench_verified_400/dataset.json`; after lexicographic sorting, we apply `random.Random(20260925).sample(..., 100)`. The mini20 diagnostic subset randomly samples 10 SLB and 10 SSB tasks.

Seen and held-out inputs. Every task mounted in the 20-task subset has exactly three inputs: two seen inputs and one held-out input. The frozen SSB-100 manifest likewise has exactly three inputs per task. For SLB, instances are first deduplicated in their persisted order; the first two are seen, the third is held out, and any remainder is recorded as unused. The deduplicated SLB pool ranges from three to six inputs. The identifiers need not be consecutive: for `weighted-gdp-calculation`, inputs 1 and 3 are seen and input 4 is held out. For SSB, numeric case identifiers are sorted, the last is held out, and all preceding cases are seen; in the frozen sample there are exactly three. Packages are compiled only on seen inputs, and every comparison in §4 is evaluated on the held-out input.

Conditions. Seven diagnostic columns differ only in what the agent receives. **bare**: no skill. **static**: the package v_0, compiled with no trace from the task statement, the seen inputs and the original skill. **native** v_1**,** v_2: the package after one and two folds, delivered through the host's skill mechanism alone; against static—same compiler, same delivery, no profile—they isolate what traces add. **static interface** and **interface** v_1**,** v_2: the corresponding package with the interface on; against native, the folded interface columns isolate the runtime layer.

Hosts, models, and settings. Runs use Claude Code version `2.1.258` with `claude-opus-5`. Launch records set `CLAUDE_CODE_MAX_CONTEXT_TOKENS=1000000`, and solving transcripts record `high` reasoning effort.

External baselines. SkVM (Chen et al., 2026), SkillSmith (Xu et al., 2026b), and skill-creator (Anthropic, 2026) run under the same Claude Code host with `claude-opus-5`. SkillSmith's paper runs its boundary runtime under the authors' internal harness (and Codex and OpenCode); we run it under the host above, an adaptation rather than the authors' configuration. SkVM and SkillSmith run from their authors' code; skill-creator runs as a skill inside the host, turns the seen-input runs into a skill, and is then evaluated on the held-out input. We found no public code for SkillDisCo (Guo et al., 2026) or TraceCompiler (Yadouni & Li, 2026); we compare them by mechanism only (Appendix B) rather than re-implement them from their papers.

Why EvoClawBench is not evaluated. EvoClawBench (Peng et al., 2026) re-runs each task on the same fixtures, so a task offers no held-out input, whereas every comparison in §4 is made on an input the compiler never saw. Its tasks do consist of several related sub-problems, which could in principle be split into seen and held-out parts; doing so would require scoring each sub-problem on its own, which we have not validated against the released grader. We therefore cite its findings but do not evaluate on it.

Cost accounting. Cost is the host's accumulated model usage including sub-agent sessions, priced at list rates (Appendix E). Top-level usage is reported only in passing, because under delegation it understates the bill by counting only the main session. Turn counts come from the per-turn ledger of Appendix E, not from the host's own turn counter, which reports a single turn for a delegating run.

Repeats and missing values. The diagnostic subset contains $20 \times 7 = 140$ task–condition cells, one primary run per cell. All completed except the safety-remediation task, whose execution was blocked by host permissions. Thus the diagnostic cell count is $n = 1$. Table 2 reports medians over tasks.

The mini20 subset. Correctness and cost are claimed from the full task set; the two mechanism analyses (§4.3, §4.4) read mini20 and serve only to explain those results. Mini20 contains 10 SLB and 10 SSB tasks; the behavior analysis excludes the safety-remediation task whose interface execution was blocked by host permissions, leaving 19 tasks.

D TURN CLASSIFICATION AND REPETITION MEASURES

§4.1 and Fig. 5 rest on assigning every turn of a recorded trace to a class. This appendix states the data, the unit of counting, the classes, and the repetition measures. The rules were frozen and committed before the full run and were not changed after any reading; the second version splits one class and reproduces every other class of the first byte for byte.

Data. SSB: 398 tasks, three runs each (1,194 runs; the three runs of one task consist only of API errors and are excluded from the denominators). SLB: 20 tasks, three runs each (60 runs), with the host's built-in skills disabled. All are bare-agent runs from scratch with gpt-6-astra, recorded before this analysis; none was rerun for it.

Unit. The unit is the turn, with weight one. A turn with several tool calls splits its weight equally among them; a turn with no tool call counts wholly as reasoning, so reasoning stays in the denominator and parallel calls are not inflated. Shares are pooled over all turns of a condition, not medians of per-run shares. Condition A counts main and sub-agent sessions; condition B counts the main session only. SSB runs have no sub-agent sessions, so A and B coincide there.

Classes. Five classes: probing, mechanical, verification, reasoning, and delegation and coordination. Mechanical has three sub-classes: running an existing entry point, writing an ad-hoc script, and writing a product directly or operating on files. The second version splits probing only: a probing call that touches an input location declared by the task is *inspecting inputs*, any other probing call is

inspecting the environment. Input locations are read from the task: for SSB the workbook path in the statement, for SLB the data directories declared in the statement and in the container's initialisation record; a task that declares no local material has an empty input set. A call that reaches an input only through a variable loaded earlier in the session would also count as inspecting inputs; none occurs in these traces.

Candidates. The lower bound of the candidate set is the three mechanical sub-classes; the upper bound adds inspecting inputs; inspecting the environment is excluded. The bounds are two definitions of the same set, not an interval estimate, and not a saving (Table 4).

Table 4: **Probing split and candidate bounds (second version), as shares of turn weight.** A: main and sub-agent sessions; B: main session only. The 100-task SSB subset is the one used in the main table; its probing split is not reported here.

Benchmark, condition, subset	Runs	Inputs	Environment	Lower	Upper
SSB, A, all 398 tasks	1,194	25.95%	0.85%	28.31%	54.26%
SSB, A, officially passed	809	25.51%	0.82%	28.21%	53.72%
SSB, 100-task subset, all	300	–	–	28.49%	55.11%
SSB, 100-task subset, passed	224	–	–	29.09%	55.15%
SLB, A, all 20 tasks	60	13.20%	19.32%	12.94%	26.15%
SLB, A, officially passed	35	12.08%	19.13%	13.20%	25.28%
SLB, B, all	60	7.34%	15.02%	15.92%	23.26%
SLB, B, officially passed	35	6.93%	14.44%	15.50%	22.43%

In the first version, condition A, mechanical steps are 28.31% of turn weight on SSB (running an existing entry point 0, ad-hoc scripts 28.29%, direct writes and file operations 0.02%) and 12.94% on SLB (0.48%, 6.34%, 6.12%); on SLB, probing is 32.52%, delegation and coordination 36.63%, and 1.89% is unclassified. Turns with more than one tool call are 20.95% of SLB turns.

Repetition. Each tool call in the main session becomes one action token and each text-only turn one reasoning token. The similarity of two sequences is the length of their longest common subsequence divided by the length of the longer one. At coarse granularity a token is its action class; at fine granularity it is the triple of class, tool name and argument shape, with paths, numbers and quoted strings masked, so similar order does not imply identical arguments. Same-task pairs are the three pairwise combinations of a task's runs; different-task pairs are 300 random pairs of runs of different tasks within one benchmark (seed 20260926), not filtered by success or matched on length. The differences quoted in §4.1 (Table 5) are differences of medians, not medians of paired differences; we report no significance test. A template is the full coarse sequence of a run: SSB has 99 distinct templates over 1,191 measurable runs, the five most frequent cover 74.73%, and the most frequent (inspect input, ad-hoc script, text reply) covers 591 runs; SLB has 59 distinct templates over 60 runs.

Table 5: **Repetition: median LCS similarity of main-session action sequences.**

Benchmark	Pairs	Number	Coarse	Fine
SSB	same task, all	1,191	0.833	0.750
SSB	same task, all three runs passed	789	0.857	0.750
SSB	different tasks	300	0.667	0.600
SSB	different tasks within the 100-task subset	300	0.667	0.667
SLB	same task, all	60	0.690	0.543
SLB	same task, all three runs passed	33	0.727	0.594
SLB	different tasks	300	0.445	0.295

E PER-TURN LEDGER

Eq. (2) is an accounting model of a run, not a description of the fields a trace records. This appendix states how Fig. 5 was read off the traces, which terms are measured and which are estimated, and what is left unattributed.

Runs and pairing. The ledger covers the 19 mini20 tasks (SLB 9, SSB 10), one held-out input per task, one run per cell, for bare and the v_2 package with the interface. A pair enters the figure only if both conditions solve the task. Grades are the official grader's except on one SLB task, where our manual review counts both runs as solved though the grader rejects them; graded by the official verdicts alone, that pair drops out and every other pair is unchanged. A task that the compiled run solves and bare does not has no cost ratio and is left out. The cost ratio is the compiled run's total cost over bare's, both from the host's own account. No cell in this batch spawned a sub-agent, so the main-session transcript is the whole run.

Turns, classes, and weights. Turns are cut and classified by the frozen rule set of Appendix D, unchanged; a turn with several tool calls splits its weight of one evenly across them. Two labels are added on top of the frozen ones, by matching the command against the scripts of the mounted package: a call that executes a package script counts as a package call, and a call that reads a script's `-h` counts as a help call. Panel (a) shows, per pair, the difference in class weight between the compiled run and bare; probing merges input and environment inspection, and reasoning is merged with the turns the rule set leaves unclassified.

Prices and per-message cost. Each model reply is priced once, from its recorded usage, at Anthropic's list rates for `claude-opus-5` (input 5, output 25, cache read 0.50, five-minute cache write 6.25 USD per million tokens)[2]; a reply with several tool calls splits its cost evenly across them. The terms of panel (b) follow. $n \cdot c$ is the cost of the replies that make package and help calls. V is the cost of verification turns after the first package call; a package's own checking script counts as a package call, not as V. F is an estimate: the instruction file's length in characters over four, times the number of turns after the skill was loaded, at the cache-read rate; it omits the meta-skill and the first cache write. R is counterfactual and is not separated: it is the difference in mechanical and probing weight of panel (a), not a dollar figure.

What is not reconciled. Summing the per-message costs of a run does not reproduce the host's total for that run in every case; the gap exceeds two percent on about half the runs, most of them on SSB, where a run costs a few tenths of a dollar and the gap a cent or two. We use the host's total for cost ratios and the per-message account for the split, and report the difference between a losing pair's total extra cost and the three named terms as unattributed (hatched in Fig. 5(b)), which can be negative; it is neither distributed over the terms nor scaled away. Loading the skill is a coordination turn under the frozen rules and carries the first cache write of the run, so the coordination increase in panel (a) is not evidence that the load alone costs that much.

F RUNTIME-LAYER OBSERVATIONS

Data. The behavior analysis of §4.4 reads the 19 mini20 tasks, 9 from SLB and 10 from SSB, solved under six conditions: the static package and the packages after one and two folds, each with the interface off and on, one run per cell, solved by Opus 5 in the same batch as the rest of mini20. All 114 traces under these six conditions, of the 133 mini20 traces (19 tasks, seven conditions), could be classified.

Labels. Each package script in each trace receives two labels, assigned by a read-only checking model, gpt-6-astra at low reasoning effort and distinct from the solving model, that reads the trace together with the list of package scripts. The first is the script's output: right, wrong, or not called. The second is what the agent did with it: *accepted directly*, the output was used as delivered; *outputs inspected*, the agent looked at the output before using it; *scripts recomputed*, the agent re-derived the output with its own code and compared; *redone by agent*, the agent set the output aside and did the step itself; and for a script never called, *skipped, but needed* or *skipped, not needed*, by whether the task required that step. A script whose output or handling the checker could not decide is *undecided* and drawn in the hatched row of Fig. 6. On a sample of 25 script rows re-judged, 23 agreed with the first judgment (92%). The undecided row of the three interface-off columns holds 12 scripts: 9 were not called and the trace gives no reason, nearly all of them scripts that write the final deliverable,

[2] `https://platform.claude.com/docs/en/about-claude/pricing`, accessed 26 September 2026.

which the agent wrote itself; 3 were called and inspected, but whether their output was right could not be settled (the candidate-collecting script of one itinerary task, in all three rounds).

Counts behind §4.4. Crossing the two labels gives the outcomes of §2: a wrong output accepted directly is $t{=}1$ where it should be 0; a wrong output caught by inspecting or redoing the step is *Redone*; a right output recomputed or redone is *Re-checked*; a needed script never called is *Ignored*. Without the interface, per trace, scripts accepted directly go from 0.2 to 2.2 on short tasks, and on long tasks package runs go from 4.3 to 4.8 and direct acceptances from 0.2 to 0.8. Pooled over rounds, without against with the interface: right outputs redone or recomputed, 10 of 48 against 6 of 46 (long) and 9 of 52 against 7 of 43 (short); needed scripts skipped, 2 of 79 against 1 of 77 and 2 of 72 against 2 of 63. The two conditions run independently compiled packages, so these are not the same errors handled twice.

Self-evolving runtime. What the model does after a script returns—accept the summary, open the log, re-check, recompute—is a natural objective for adapting the runtime itself, not only the package. We leave this to future work.

G Extended limitations

Sample size and noise. The diagnostic subset has one primary run per task–condition cell, and we report no significance test, so small per-task differences, especially on short tasks, are not read as effects.

Generalization and staleness. An executable artifact fails more quietly than a document: outdated text is read by a model that may notice the mismatch, whereas an outdated script simply runs, and injected guidance can hurt when it disagrees with its environment (Han et al., 2026; Peng et al., 2026). A stated boundary makes failure visible, not impossible, and a fold sees only seen inputs: an assumption that first fails on a held-out input cannot reach the compiler without spending that input, so H is only as good as the seen inputs are varied.

Generalization: declaring more is not generalizing more. The compiler's weakest point is the boundary of what it has seen. In one case it turned the two distance formulas observed on seen inputs into a hard whitelist inside a script and raised an error on any other value, while the package's instructions did not list the admissible values. A soft gap can be worked around; a hard whitelist blocks. The current compile skill requires every input-dependent value to be a parameter the model fills, which addresses the second half of this; carrying an evidence grade on value domains, so that an unseen value is admitted as unverified rather than rejected, is future work.

Scope. A coding agent given raw trajectories is already a strong skill optimizer (Shen et al., 2026); we use one as our compiler and test against the same agent writing experience into a skill (Anthropic, 2026). One host and one model compile and solve throughout; tokens are compared only with systems run under both. We do not evaluate on EvoClawBench (Peng et al., 2026), which re-runs each task on the same fixtures and so offers no held-out input (Appendix C).

H An illustrative scenario: from Alice's Workflow to a personalized Harvey

Consider Alice, a lawyer who uses an agent for routine legal work. She begins with *Alice's Workflow*, a short document describing how she prefers tasks to be performed. During the first executions the agent repeatedly searches project files and learns which sources hold relevant matter history; the compiler recognizes this recurring structure and compiles it into a retrieval component. As further regularities emerge it adds indexing strategies, structured databases, and code for particular legal tasks, each exercised and debugged in Alice's actual environment. After many executions, Alice's Workflow resembles a personally customized version of Harvey, a legal AI platform, built around Alice's matters, routines, data, and preferences. The analogy is not that the compiler recreates Harvey; it is that compiling traces lets a simple textual skill grow into similarly rich infrastructure for one user.

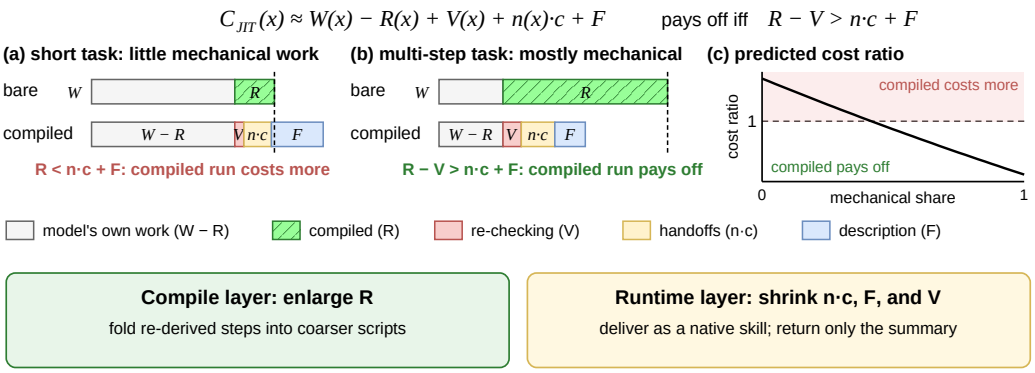


Figure 8: **Compilation pays off exactly when the compiled work exceeds the cost of handing it over.** A schematic with illustrative magnitudes, not a claim about any task length. (a) The work removed by scripts (R) is smaller than the handoff terms, and the compiled run costs more. (b) R dominates and the compiled run pays off. Bars decompose the bare cost W into work the model still does (grey), work removed by scripts (R, green hatch), re-checking of script results (V, red), per-call handoff cost ($n \cdot c$, yellow), and the package description in context (F, blue). (c) Predicted cost ratio against a task's mechanical share, with the break-even line; the curve is a shape, not a fit. Bottom: the compile layer enlarges R; the runtime layer shrinks $n \cdot c$, F, and V.

I THE COST MODEL OF THE HANDOFF

Let $W(x)$ be the work the model performs on input x with no package, in whatever the deployment pays for; $R(x)$ the mechanical work a compiled package removes from the trace; c the per-call cost of composing, returning and reading a script call, paid $n(x)$ times; F the fixed cost of carrying the package's description in context; and $V(x)$ the work of re-verifying or recomputing what a script already returned. Then

$$C_{\text{JIT}}(x) \;\approx\; W(x) \;-\; R(x) \;+\; V(x) \;+\; n(x)\cdot c \;+\; F, \tag{2}$$

and compilation pays off only when

$$R(x) \;-\; V(x) \;>\; n(x)\cdot c \;+\; F. \tag{3}$$

Condition (3) is an accounting identity, not a theorem. In the terms of §2, ignoring a package forfeits R, recomputing and re-verifying are V, and an ineffective fold adds nothing to R. Fig. 8 draws its terms schematically. The model separates the design into two halves that optimize the same quantity from opposite sides. The compiler (§3.1) exists to enlarge R: to move as much deterministic work into scripts as it can without harming generalization. The runtime (§3.2) exists to shrink V, c, and F: to let the model call a script cheaply and to make it willing to use the returned result.

Where R is small the compiled run can cost *more* than bare, because the handoff terms are paid regardless of R; §4.3 reads the terms off real traces and finds that on such runs they are dominated by the calls and the coordination around them, not by F.

Compilation itself is paid for once: if building a package costs K and $\bar{m}$ is the mean of $R - V - n \cdot c - F$ over reuse inputs, the cost is recovered after $N^* = \lceil K/\bar{m} \rceil$ reuses when $\bar{m} > 0$ and never otherwise. This is the measure on which self-authored skills do not pay back on a cost subset of EvoClawBench (Peng et al., 2026); we do not report it here.

J BOUNDARY FORMAT AND FOLD RULES

This appendix states the rules of §3.1 in full. Their common aim is to make each script's boundary explicit and to sharpen it at every fold.

The entry and the boundary. The instruction file gives each script one entry of three lines, not organized by steps: *run*, the command with its parameters left for the model to fill; *params*, each

parameter with where its value comes from; and *does*, one sentence. The boundary is printed by the script's `-h` in four parts: what it does, naming what it reads and writes; its parameters; its steps and their products; and its assumptions and limits, one line each giving the assumption, its source, what to do when it fails (the parameter that repairs it, or that the script does not apply and the model should follow the written steps), and how a misuse would show in the summary. When the boundary states that the script's values simply differ from the task's, the model declines the script and proceeds by hand. Assumption identifiers combine the script name and a sequence number, are never changed once assigned, and are never reused after deletion; every later fold refers to them.

Sources. An assumption's source is exactly one of: the task statement (may be hard-coded); the original skill given to static compilation (may be an admissible value of a parameter, never its default); seen input k (must be a parameter or remain a stated assumption); or a compile-time choice (with one sentence of reason). Defaults come only from the task statement or the seen inputs. The provenance scan extracts numbers other than 0 and ± 1, strings of two or more characters, and strings shaped like column letters, header names or file names, and reports for each where it occurs among the task statement, the original skill and the seen inputs. It runs after static compilation, again at every fold with the new seen input added, where a constant absent from the new input or taking a different value there marks a refuted assumption, and on each model edit during triage.

What is collected. Before a fold: edits inside the mounted package, as a diff against the delivered version; files in the workspace that share a package script's name or opening lines, as a diff against that script; and every other file the transcript shows the model writing, recorded by path and type, with the package scripts that program files among them call. Nothing collected means an empty list, and the fold proceeds.

Diagnosis. Each script step receives one or more entries, each with a trigger (deviation or assumption) and a verdict (keep or change) whose reason names the kind of change (a step that fails, a parameter that does not cover the input, a missing branch, a step that can be promoted). *Keep* requires both triggers to have been checked; a fold that did not rerun the provenance scan on the new input has not checked the assumption trigger, and none of its *keep* verdicts stands. Insufficient evidence is not a reason to keep.

Triage of model edits. Each edit, and each file the model wrote in place of a package script, is classified by the first criterion that decides it. (1) The reason the model gave before editing: a property of the current input (a column name, a sheet, a format, a requirement of the task) marks it input-specific; a logic error, a crash, or a missing function marks it general. (2) The direction of the edit: a literal replaced by a value read from the input or a parameter is general and is absorbed; a new constant that occurs in the current input is input-specific and is not. (3) The assumptions: an edit on code that backs a seen-input assumption refutes it, so the assumption is promoted to a parameter or its boundary rewritten, and the model's code is evidence only; an edit on no assumption is a genuine gap or bug, and its code is absorbed. Each classification is recorded with the edit, the criterion, the treatment, the assumptions involved, and the evidence. No regression test decides generality.

Checks. Each version is rejected unless: every script has an entry with its three lines and a `-h` with its four parts; every assumption has a well-formed, unique identifier and one of the four sources; every constant whose source is a seen input is a parameter default or a listed assumption; every line added relative to the previous version contains an assumption identifier or a script name; and every diagnosis entry uses one of the two triggers and one of the two verdicts, with at least one entry per step. Deletions are unrestricted, and no byte or line limit applies.