SMALL EXECUTORS DERAIL ON INTERMEDIATES THEY PRODUCED THEMSELVES: COMPILING SKILLS INTO TOOLS THEY CAN ACT ON

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent skills package procedural knowledge as natural-language instructions that an LLM agent loads at inference time, and are expected to carry over to any model without retraining. We find that they largely fail to reach small open executors. Offered through the standard skill channel, skills are often left unopened; placed directly in context, they are read and still fail. Small executors derail not on understanding the procedure but on the intermediate results they must produce themselves to carry it out. In controlled interventions, once these results are messy, the next step fails for executors of every size, and in a pre-registered handoff, a larger executor that takes over a small executor's state fails more often than after its own turns. We therefore compile each skill into tools that return clean intermediate results, once per task family from the public skill package and a single development instance, while the executor still decides which tools to call and how to finish. On SkillLearnBench, compiled tools raise a 4B executor from 9 to 53 of 80 held-out instances, above a 27B executor given the skill in context, and the gains replicate on a second model family. Written out as text, the same suite does not help: what small executors lack is not the knowledge in a skill, but tools they can act on.

1 INTRODUCTION

Agent skills package the procedure for a class of tasks, written as instructions and optionally accompanied by scripts, so that an agent harness can load it when a task calls for it (Anthropic, 2025; Xu & Yan, 2026). They promise procedural knowledge without retraining: written once, and reused by any model that runs the harness. On hosted frontier models, curated skills do raise pass rates (Li et al., 2026), and benchmarks now treat human-written skills as the target that automatically generated skills should reach (Zhong et al., 2026).

We ask whether skills also reach small open executors, the models one runs locally. On SkillLearnBench, under its official harness and grader, they largely do not (Figure 1). Offered through the standard skill channel, the official skills are almost never opened by the Qwen executors, which stay at their no-skill level; the Gemma executors open them more often but gain far less than from compiled tools. Placed directly in context, the same skills help mainly the largest executor; a 4B executor still passes only 9 of 80 held-out instances. One might conclude that small executors cannot follow a written procedure. Our evidence points elsewhere.

Small executors read the skill and attempt its steps. What they fail at is the material those steps work on. A skill describes a procedure whose steps feed one another: one step inspects the input, the next extracts or transforms what it found, the last writes the deliverable. Every intermediate result in this chain has to be produced by the executor's own commands and code. In court-form filling, for example, a 4B executor decides, without running the check that the skill prescribes, that the form has no fillable fields, and builds every later step on that conclusion (§2.3). The skill does not tell it how to notice such a mistake, and once derailed, the trajectory rarely recovers. Controlled interventions confirm the direction: once an intermediate result is messy, the next step fails for executors of every size, and it succeeds far more often from states in which that result is clean.

Is this state merely a symptom of a weak executor, or an obstacle in itself? We answer with a pre-registered handoff (§2.4). A small executor works on a task for a few turns, and a larger executor then

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.

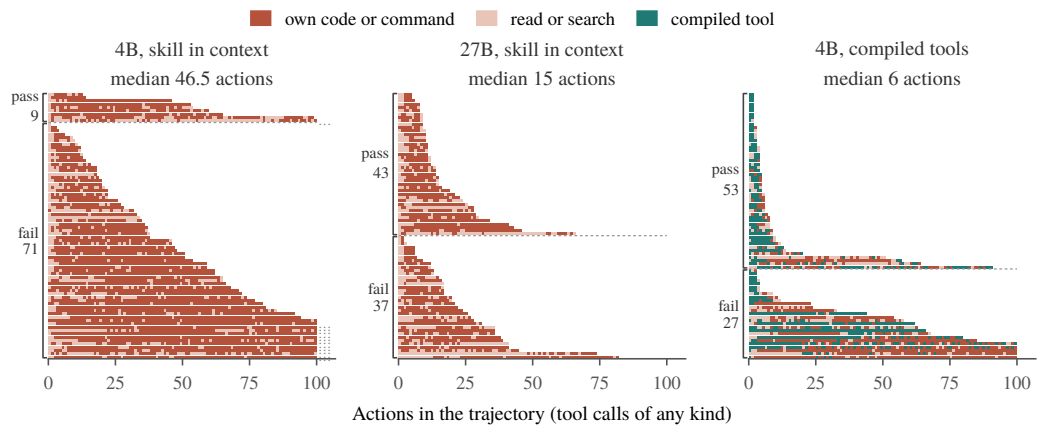


Figure 1: **With compiled tools, small executors carry out skills that they fail to carry out as text.** Every held-out trial of three Qwen3.5 conditions, one row per trial (80 per panel) and one cell per action; brackets mark passed and failed trials, each sorted by length, and trials longer than 100 actions are cut (. . .). From text, the 4B executor produces its intermediate results itself and mostly derails; with compiled tools, it mostly passes after a few calls.

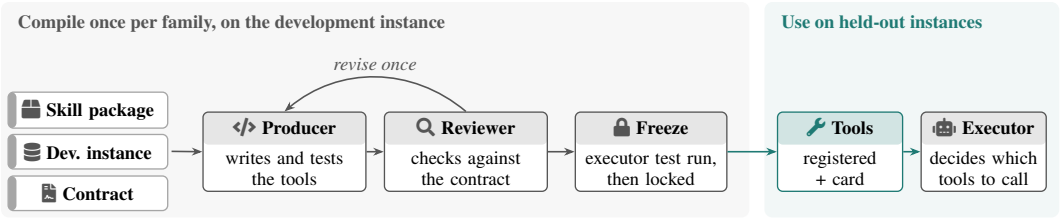


Figure 2: **Compiling a skill from one development instance.** Once per family, a producer builds tools from the public skill package, an independent reviewer checks them, the producer revises once, and the suite is frozen before it is used on held-out instances.

continues the same session, with the same files and conversation. As a control, the larger executor first works the same number of turns itself. The larger executor fails markedly more often when it inherits the small executor's state than when it inherits its own, and the gap widens the longer the small executor has worked. The state a small executor produces derails even an executor that could solve the task.

This diagnosis suggests a remedy: take the production of intermediate results out of the executor. We compile each skill into a suite of tools registered in the executor's action space (Figure 2, §3). An agentic producer builds the suite once per task family, from the public skill package and a single development instance, and an independent reviewer checks it before it is frozen; neither ever sees tests or reference solutions. The executor still decides which tools to call, with which arguments, and how to finish the task, but each tool returns a clean, labeled intermediate result instead of leaving it to the executor.

On 80 held-out instances from 20 families, compiled tools raise the 4B executor from 9 to 53 passes, more than a 27B executor achieves with the skill in context, and the gains replicate on a second model family. They are largest where the skill in context helps least. Ablations show that the gain does not come from better-informed text: the same tool suite written out as text does not help, and running its code as scripts recovers only part of the gain for small executors. Our contributions are:

- **A measurement of where skills fail small executors.** Under the official harness, skills fail at two points: access, since the executor rarely opens the skill, and execution, since it fails even with the skill in context (Sections 2.2 and 2.3).

- **Evidence that execution fails on self-produced intermediate results.** Controlled prefix interventions and a pre-registered cross-family handoff show that the state a small executor produces blocks later steps, even for a larger executor (Sections 2.3 and 2.4).
- **A method that compiles a skill into tools from one development instance.** Its gains hold across two model families, and ablations tie them to tools the executor can act on rather than to better-informed text (Sections 3.2 to 3.4).

2 Skills Fail Small Executors

We first describe how a skill reaches an executor and what carrying it out involves (§2.1). Small executors then fail at two points. They rarely open the skill (§2.2), and when they do read it, they fail on intermediate results they produce themselves (§2.3). A larger executor that takes over their state cannot undo this failure (§2.4).

2.1 Executing a Skill

Skills and their delivery. A skill package holds a `SKILL.md` file, with a name, a description and a written procedure, and optionally scripts and reference files (Anthropic, 2025). A harness can hand it to the executor in two ways. Claude Code lists installed skills by name and description and loads a skill's text only when the executor invokes its Skill tool; text placed in the project instructions (`CLAUDE.md`) is instead loaded into context on every turn. Table 1 lists the conditions we compare, and Figure 3 contrasts carrying out a skill from text and with compiled tools. Only the material handed to the executor differs between them.

Intermediate results. A skill describes a procedure whose steps feed one another. We call the output of a step an *intermediate result* when it is neither part of the task input nor the graded deliverable, but is the input of a later step. It is *self-produced* when the executor obtains it through actions it composed itself (code, commands, queries), rather than receiving it from the task, the skill text or a compiled tool. An intermediate result is *clean* when its content is correct for the instance, when it is compact and labeled with only the fields the next step needs, and when it sits where the next step will look (the latest observation or a named file). The output of the compiled tool for the same step (§3.1) serves as the reference for clean; a result that is not clean is *messy*. We say that an executor *derails* when its later steps build on a messy self-produced intermediate result.

Setup. We use SkillLearnBench (Zhong et al., 2026): 20 task families, each with an official skill package, a development instance (instance 1) and 80 held-out instances in total, all of which we use. The official runner drives Claude Code (v2.1.197) inside each instance's container with the benchmark's limits (100 turns, 1,800 s), and the official grader passes an instance when all its tests pass; we change none of them. The executors are Qwen3.5 4B, 9B and 27B, and Gemma 4 E4B, 12B

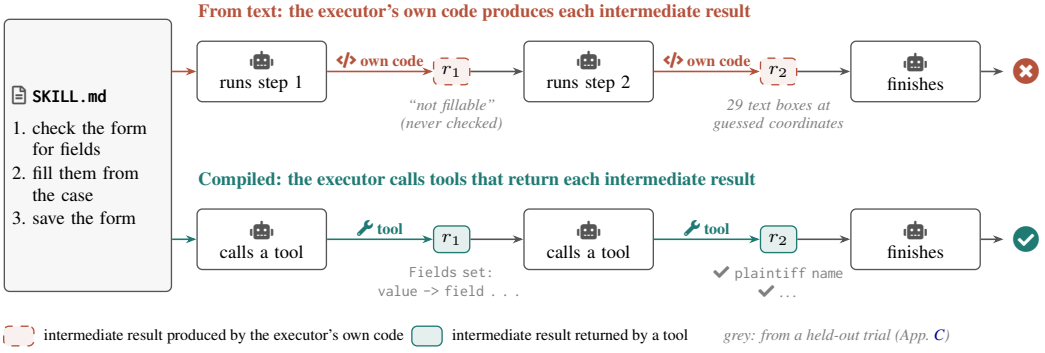


Figure 3: **One skill, two forms.** The executor takes the same steps in both; what changes is who produces each intermediate result: the executor's own code (top) or a tool that it chooses to call (bottom). A tool may cover one or several steps. Grey notes come from a held-out trial (Appendix C).

Table 1: Delivery conditions. All conditions use the same harness, container, limits and grader.

Condition	What the executor receives	Where
No skill	nothing	–
Skill channel	official skill package	installed skill, loaded on invocation
Skill in context	official skill package + its `SKILL.md` text	installed skill + `CLAUDE.md`
Tools	compiled tools (§3.1)	registered tools
Tools + card	compiled tools + a short card naming them	registered tools + `CLAUDE.md`

and 31B, served with vLLM behind a proxy that translates Claude Code's requests, with one context window (81,920 tokens) for all and each model's recommended sampling settings. We report one trial per held-out instance, with a second run for two conditions, for 3,693 valid trials across all experiments; trials ended by an infrastructure error (3.0%) are re-run (Appendix A).

2.2 Small Executors Rarely Open Skills

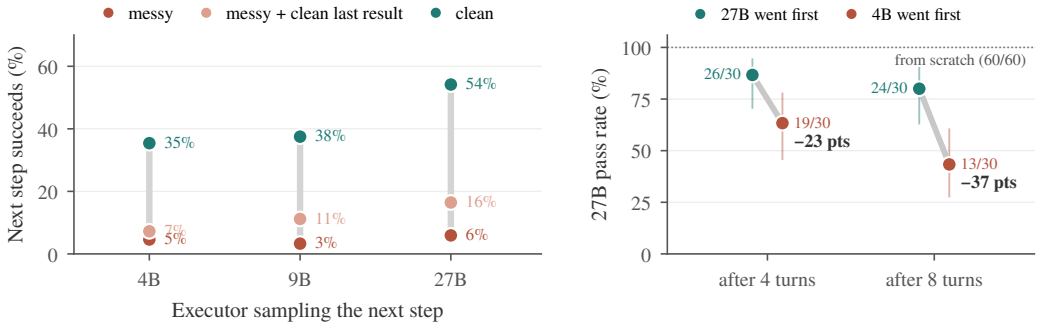
We first ask whether the skill reaches the executor at all, comparing no skill, the skill channel and the skill in context on all 80 held-out instances. Offered through Claude Code's standard skill channel, the official skill packages leave all three Qwen3.5 executors where they are without any skill: 0, 1 and 10 of 80 held-out instances pass, against 1, 1 and 10 with no skill. The executors almost never invoke the skill: 4B, 9B and 27B call the Skill tool in 1, 1 and 8 of 80 trials. This is not a general reluctance to use tools. In the same harness, the same executors call registered tools in 96–100% of trials (§3.2). The Gemma 4 executors open the skill more often, in 29, 13 and 23 of 80 trials at E4B, 12B and 31B. The channel then helps the two larger ones somewhat (7 and 19 passes, against 1 and 10 without a skill), while E4B passes none (Table 2).

Access explains part of the gap. Placing the same package text in the project instructions (`CLAUDE.md`), which Claude Code loads on every turn, raises pass counts to 9, 21 and 43 of 80. The skill is now in front of the executor on every step, yet the 4B executor still fails 89% of instances. The rest of this section examines that failure.

2.3 Small Executors Derail on Self-Produced Intermediates

We next ask why the skill in context still fails, and first look at where the failed 4B trajectories go wrong. The executor reads the skill and follows its steps, but the intermediate results it produces along the way are unusable: in video counting it deletes and renames the key frames it has just extracted, together with the task's reference images, and extracts them again; in court-form filling it decides, without running the check that the skill prescribes, that the form has no fillable fields, and spends some twenty calls repairing text boxes it placed at guessed coordinates (Appendix C); in enterprise search its attempt to read a 734 KB data file is refused, and it assembles the answer from scattered search hits and a truncated employee table. The executor derails: each next step of the skill is applied to a state that the skill does not describe.

Controlled prefix interventions. To probe this more directly, we replay a trajectory up to a chosen step, change at most one observation, and sample only the next step. This requires the prefix to be reproduced exactly, so we run the interventions in a lightweight ReAct scaffold whose conversation is a plain message list under our control; a production harness such as Claude Code inserts its own reminders and manages the context between steps. The executors and their sampling settings are those of the main experiments, and the prefixes come from the family's held-out instances. Instead of the official package, the skill is a plain-language statement of the procedure without code blocks, so that the next step cannot be copied from the skill and has to be built on the intermediate result (Appendix F). We took prefixes of 4B trajectories in one family (enterprise search) at the point where the key intermediate result, the list of shared posts with their timestamps, first becomes available: 6 prefixes in which it is clean and 19 in which it is messy. From each prefix we sampled the next step eight times per executor. In this family, the state of the intermediate result goes with a large difference in next-step success at every size (Figure 4a). From clean prefixes the next step succeeds in 35%, 38% and 54% of samples for 4B, 9B and 27B, and from messy prefixes in 5%, 3% and 6%; the 27B executor cannot continue from the 4B executor's messy state either. Replacing only the



(a) Next step from a 4B state (one family).

(b) 27B takes over after K turns (11 families).

Figure 4: **A small executor's self-produced state derails the next step, even for a larger executor.** (a) Next-step success from prefixes of 4B trajectories in one family whose key intermediate result is messy (19), messy with the last observation made clean, or clean (6); eight samples per prefix and executor (§2.3). (b) Pre-registered handoff under the official harness: 27B pass rate after it, or the 4B executor, worked the first K turns, on the 30 instances it solves from scratch; whiskers: 95% Wilson intervals.

last observation of a messy prefix with a clean one raises success to just 7%, 11% and 16%: once a trajectory has gone astray, a single repair does not bring it back. The size difference therefore lies mainly in whether an executor brings itself into a clean state, which in this scaffold the 27B executor typically does by its third step and the 4B executor rarely does.

2.4 Larger Executors Inherit the Derailment

The prefix interventions cover one family and a single next step. We therefore ask, across families and up to the grader's verdict, whether a larger executor can finish from a small executor's state.

Handoff across families (pre-registered). We ran a handoff experiment under the full benchmark protocol; its design, predictions and falsification criterion were registered before any handoff trial ran (Appendix G). We took the 30 held-out instances (11 families) that the 27B executor solves from scratch with the skill in context in both runs (60/60). In each trial, a first executor works for K turns. The 27B executor then resumes the same session in the same container, with the same files, conversation and remaining turn budget, and the official grader scores the result. The first executor is the 4B executor or, as a control for interrupting and resuming a session, the 27B executor itself.

Figure 4b confirms the two main predictions. At both K, the 27B executor succeeds less often after the 4B executor's turns than after its own. The pre-registered paired test, pooled over both K, gives a two-sided exact sign test $p = 0.0009$ (discordant instances 10:3 at $K = 4$ and 13:2 at $K = 8$). The gap widens with K (87% vs. 63% at $K = 4$; 80% vs. 43% at $K = 8$): the longer the small executor works, the less the larger one can recover. The third prediction, that self-handoff stays within 15 points of solving from scratch, is met at K=4 (87%) but not at K=8 (80%, against 100% from scratch): interrupting and resuming a session is itself costly. We therefore read the result only through the paired contrast between the two first executors, which holds the interruption fixed. The state a small executor produces in its first few turns, its files together with its conversation, is an obstacle in its own right: it derails even an executor that solves the same instances from scratch.

3 Compiling Skills into Tools

3.1 Compiling from One Instance

We compile each family's skill package into a small suite of tools (two or three per family) that the executor can call (Figure 2). Each tool returns, clean, an intermediate result that the executor would otherwise produce itself, which removes the points at which small executors derail (§2.3).

Table 2: Held-out pass counts (out of 80) by delivery condition and executor. The best condition for each executor is in bold.

	Qwen3.5			Gemma 4		
Condition	4B	9B	27B	E4B	12B	31B
No skill	1	1	10	0	1	10
Skill channel	0	1	10	0	7	19
Skill in context	9	21	43	0	22	33
Tools	39	55	**71**	**54**	**52**	71
Tools + card	**53**	**60**	69	47	51	**72**

Compilation is done once per family, from the development instance, and each family's final suite is frozen before its held-out runs.

Inputs. The producer is an agentic Claude Opus 5 session that can read files and run code, either in an isolated workspace or inside the development instance's container. It receives the family's public skill package, which it reads first; the development instance, meaning its instruction and input files; a generation contract; a description of the runtime the executor will see; and a short family note, written from the development instance and the skill package, on where a small executor is likely to derail and how to divide those steps into tools. The producer never sees tests, official solutions or any other instance. We verified this from the logs of all 84 producer and reviewer sessions, and no frozen suite contains a held-out answer (Appendix E).

Contract. The contract (Appendix B.1) asks for tools that carry the *procedure* of the skill and not the content of the development instance. Its main rules are: (i) the producer first lists the capabilities the package teaches and the aspects that can vary between instances of the family; (ii) aspects that vary are parsed from the task text or discovered from the input files, with overridable defaults from the development instance, so that arguments are required only where the task itself supplies facts (the case details in court-form filling and the product name in enterprise search); (iii) each call returns a compact report whose last line names the next step; (iv) a checker tool verifies the deliverable against the requirements stated in the instruction; (v) tool names use the words of the task.

Review and freeze. A second Claude Opus 5 session with a fresh context reviews each suite against the contract. It looks in particular for constants copied from the development instance and for silent fall-backs to it. The producer then revises the suite once. We run the revised suite on the development instance under the full protocol as a final check, then freeze it. Compilation is a one-off cost per family, amortized over all of its instances: producer and reviewer together take a median of about an hour of active agent time, 172k output tokens and $12 at API list prices per family (Appendix B.4).

Delivery. Inside the container, a stdio MCP server exposes the frozen suite as registered tools and runs each call in the task's working directory. With the card, `CLAUDE.md` additionally lists each tool's name, its required arguments, if any, and the first sentence of its description. The executor decides whether, when and how to call the tools.

3.2 Compiled Tools Close the Gap

We present the main results in Table 2 and Figure 5, and per family in Appendix D. With compiled tools, the 4B executor passes 53 of the 80 held-out instances, more than the 27B executor passes with the skill in context (43). Compared with the skill in context, the tools add 44, 39 and 26 passes at 4B, 9B and 27B. On the same instances, the discordant pairs split 45:1, 43:4 and 30:4 in favor of the tools (two-sided exact sign test, $p < 10^{-11}$, $p < 10^{-8}$ and $p < 10^{-5}$). Resampling whole families gives 95% intervals of [42, 68], [33, 63] and [14, 51] points for the gain. The gain shrinks as the executor grows, as expected if it comes from the steps where small executors derail (§2).

The executors use the tools (Figure 1). They call at least one tool in 96%, 100% and 99% of trials, and the 4B executor needs a median of 6 actions per trial, against 46.5 with the skill in context

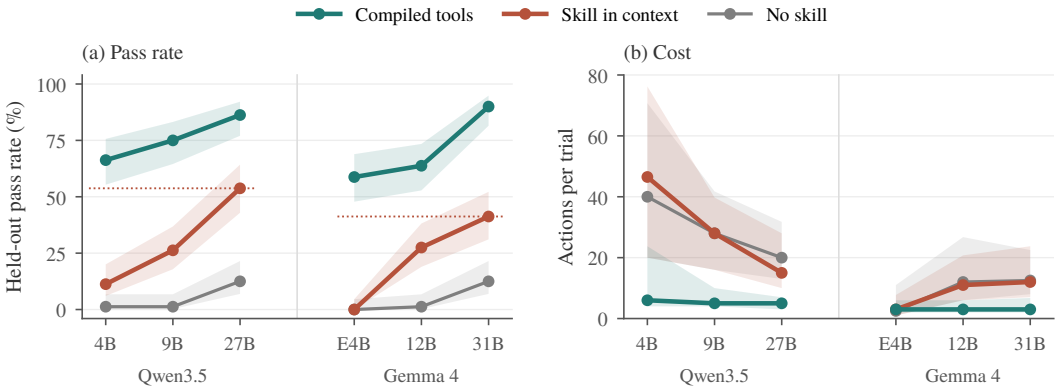


Figure 5: **Compiled tools lift every executor, the smallest most, in a fraction of the actions.** (a) Held-out pass rate by executor (80 instances per point; bands: 95% Wilson intervals; dotted line: the largest executor of each family with the skill in context). (b) Actions per trial (median; bands: interquartile range).

(Figure 5b). The tools do not finish the task on the executor's behalf. In 7 families, most passing trials include commands or files that the executor writes itself, for example an answer assembled from the lists a tool returns, and the 4B executor passes these families about as often as the others (17/26 against 36/53 instances). Where the 4B executor still fails with tools, it calls them about as often as in its passing trials (a median of 4 against 3 calls) but takes many more actions of its own (21 against 2): it derails where it still produces intermediate results itself (Figure 1). A second, independent run reproduces the result: 57, 60 and 71 passes with tools, against 8, 15 and 37 with the skill in context, and 78%–95% of instances receive the same outcome under the tools in both runs. The gain also holds on the five evaluation families, fixed in advance and never used to design the method: on their 13 instances the tools lead for every executor (Appendix E).

3.3 Replication on Gemma 4

We next ask whether the result depends on the executor family. *Setup:* we run Gemma 4 at E4B, 12B and 31B under the same protocol and in all five conditions (Table 2). The tool suites are the frozen ones from §3.1. They were written for and checked with the Qwen executors, and no Gemma executor ran them before these trials.

The pattern repeats (Figure 5). With the skill in context, Gemma E4B, 12B and 31B pass 0, 22 and 33 of the 80 held-out instances; with compiled tools they pass 47, 51 and 72. On the same instances, the discordant pairs split 47:0, 31:2 and 41:2 in favor of the tools ($p < 10^{-13}$, $p < 10^{-6}$ and $p < 10^{-9}$; family-bootstrap intervals [45, 72], [19, 54] and [33, 64] points). Every Gemma executor calls a tool in all of its trials, and on Gemma the tools do about as well without the card (54, 52 and 71 passes; §3.4). Without tools, the Gemma executors do not run long trajectories as the small Qwen executors do; they stop early, E4B after a median of 3 actions with the skill in context (Figure 5b). As with Qwen, the executor that gains least from the skill gains most from the tools.

3.4 Ablations

The compiled tools are written by a strong producer that could run code on the development instance, whereas the human skills were not. The gain could therefore come from better-informed content rather than from compiling it into tools. We separate the two by removing one part of the full method at a time (Table 3), each time on all held-out instances and paired with the full method on the same instances: (a) the card in `CLAUDE.md` (tools only); (b) the independent review, using the producer's draft of the same suite; (c) registration, with the same code run as one script per step, each script named in `CLAUDE.md`; (d) the code, with the same suite written as a text skill in `CLAUDE.md` (each tool's full description and parameters as one step, without code). Both rows keep

Table 3: **Ablations: removing one part of the method at a time.** Held-out pass counts with Qwen3.5 executors; in parentheses, the change against the full method on the same instances. Paired sign tests are reported in the text. Row (b) covers the 71 instances whose suite's draft was kept (all families except financial analysis and travel planning).

	4B	9B	27B
Full method: tools + card	53	60	69
(a) w/o card	39 (−14)	55 (−5)	71 (+2)
(b) w/o review	50 (+3)	59 (+5)	60 (−1)
(c) w/o registration (scripts)	26 (−27)	34 (−26)	63 (−6)
(d) w/o code (text only)	3 (−50)	3 (−57)	20 (−49)
Official skill in context	9	21	43

the producer's knowledge. Row (c) also keeps its code, its reports and a pointer in `CLAUDE.md`, and changes only how the executor invokes the code; row (d) removes the code.

Card. Without the card, the tools stay registered, but the executor has to find them in its tool list. Removing the card costs the Qwen 4B executor (discordant pairs 18:4, $p = 0.0043$) but not the larger Qwen executors ($p = 0.27$ at 9B, $p = 0.75$ at 27B). On Gemma 4 the card does not help: without it, E4B, 12B and 31B pass 54, 52 and 71, against 47, 51 and 72 with it (discordant pairs 3:10, 5:6 and 4:3; $p = 0.09$, $p = 1.00$ and $p = 1.00$). The card is therefore a cheap aid that helps the smallest Qwen executor find the tools, not a necessary part of the method.

Review. Using the producer's draft instead of the reviewed suite does not measurably change the result: +3, +5 and −1 passes at 4B, 9B and 27B (discordant pairs 6:9, 2:7 and 4:3; $p = 0.61$, $p = 0.18$ and $p = 1.00$). The reviewer did find, and the producer fixed, problems such as silent fall-backs to values of the development instance, but these fixes do not change the held-out pass counts.

Code as an action, not text. Written as text, the producer's suite gives the executor everything the tools know except the code that runs it. In this form it does worse than the official skill in context at every size (3, 3 and 20 passes, against 9, 21 and 43), and far worse than the same suite as tools (discordant pairs 51:1, 57:0 and 50:1; $p < 10^{-13}$, $p < 10^{-16}$ and $p < 10^{-13}$). The executor tries to carry out the described procedure itself; the 4B executor spends a median of 57.5 actions doing so. Running the same code as scripts named in `CLAUDE.md` recovers part of the gain at 4B and 9B (26 and 34 passes, against 53 and 60 with tools; discordant pairs 30:3 and 27:1, $p < 10^{-5}$ and $p < 10^{-6}$) and most of it at 27B (63 against 69; $p = 0.15$). The difference lies largely in whether the executor uses the code at all: the 4B executor never runs a script in 41 of 80 trials, the 9B executor in 32, and the 27B executor in only 13, whereas registered tools are called in 96%–100% of trials. The tools do add computation that text cannot supply; moving it out of the executor is the purpose of compiling. The ablations show that knowing the procedure does not replace this computation, and that it helps a small executor fully only as tools it can act on.

4 Related Work

Agent skills. Agent skills package instructions, scripts and resources that a harness loads on demand (Anthropic, 2025; Xu & Yan, 2026). SkillsBench reports that curated skills raise pass rates across 18 model–harness configurations built on hosted models (Li et al., 2026). SkillLearnBench, which we use, compares automatically generated skills with human-authored ones and treats the latter as the target (Zhong et al., 2026). SkillCraft asks whether agents abstract compositions of tools into skills and reuse them across long-horizon tasks (Chen et al., 2026). Studies of smaller executors are fewer and less favorable. Very small open models struggle to select skills reliably, while models of 30B–80B parameters benefit (Xu et al., 2026). Skills written for frontier models do not transfer directly to 4B–9B executors, which motivates generating skills tailored to them (Dang et al., 2026). Skills that look relevant can also lead agents to implement task elements incorrectly or omit them (Dong et al., 2026). These studies ask whether skills help and how to write or generate better ones, judged by the pass rate of the executor that receives them. They do not trace where, in a

small executor's trajectory, a skill that serves strong executors stops working; we study that question under the benchmark's own harness and grader.

Language models that make tools. A strong model can write tools that a weaker model then uses (Cai et al., 2024). It can create tools to separate abstract planning from concrete execution (Qian et al., 2023), build and retrieve task-specific toolsets (Yuan et al., 2024), induce compact and verifiable toolboxes (Wang et al., 2024c), or turn papers and code repositories into agent tools (Wölflein et al., 2025). Libraries of executable skills go back to Voyager (Wang et al., 2024a). We compile an existing human skill package from a single development instance, without access to the grader, and use the result to show where small executors derail.

Interfaces and executable actions. The interface an agent acts through shapes what it can do. Purpose-built agent–computer interfaces improve software-engineering agents (Yang et al., 2024), and executable code actions outperform text or JSON actions (Wang et al., 2024b). The same holds for procedural knowledge: one compiled suite, delivered as text, as runnable scripts or as registered tools, leads to very different outcomes for a small executor (§3.4).

Compounding errors in long-horizon execution. Language models over-commit to their early mistakes (Zhang et al., 2024) and rarely repair their reasoning without external feedback (Huang et al., 2024). In a synthetic task that supplies both the knowledge and the plan, so that only execution is tested, models become more error-prone when their context contains their own earlier errors, and scale does not remove the effect (Sinha et al., 2026). When small open models act as database agents, most failures occur after the model has already invoked a tool (Bozoglan et al., 2026). In agentic skill execution, the state is carried by files and tool outputs that later steps must consume. We show that small executors fail mainly at producing this state rather than at continuing from it, and that the failure persists when a larger executor takes it over (§2).

5 Conclusion

Skills written for an agent largely fail to reach small open executors. Offered on demand, they are often left unopened; placed in context, they are read and still fail: small executors derail on the intermediate results they must produce themselves, and a larger executor that takes over their state fails more often than after its own turns. Compiling each skill, once per task family, into tools that return clean intermediate results lets small executors of two model families carry out skills they cannot carry out from text. Written as text, the same suite does not help: what small executors lack is not the knowledge in a skill but tools they can act on.

Limitations. SkillLearnBench has few held-out instances per family, so we rely on paired comparisons over all instances rather than on per-family results. All results use one harness, whose on-demand loading drives the access failure. The prefix interventions cover one family and take their prefixes from observed trajectories rather than from randomly edited states. The handoff runs in one direction only and transfers files and conversation together. Compilation needs a strong producer, a development instance and a short family note per family, and the tools carry over to new instances of that family, not to new families. A tool never authors the judgment-bearing part of a deliverable (Appendix B.1), so tasks whose value lies in such content gain less.

Outlook. For skill authors, our results suggest a different target: not a clearer description of each step, but code for the steps whose intermediate results a small executor cannot produce reliably. Compiling from the skill package alone, for other harnesses, and measuring the reverse handoff are natural next steps.

Ethics Statement

This work uses a public benchmark and open-weight models, and involves no human subjects or personal data. The tool suites were generated without access to held-out tests or reference solutions (Appendix E); one suite whose producer read a reference solution shipped in a task package was discarded.

Reproducibility statement

The supplementary material contains the frozen tool suites with their SHA-256 hashes, the generation contract (Appendix B.1), the family notes and the producer and reviewer prompts, the runner patches and proxy hooks, the executor serving and sampling settings (Appendix A), the record and condensed trajectory of every reported trial, the per-prefix outcomes of the prefix interventions, the scripts that compute the reported numbers and figures from these records, and the handoff preregistration (Appendix G); in the court-form suites, the few blocks copied from a skill package that may not be redistributed are replaced by references, which a script restores byte for byte from the reader's copy of the package. The setup, method and deviations are described in Sections 2.1 and 3.1 and Appendix A.

AI use statement

In this work, we used generative AI tools (Claude models, through Claude Code) to help develop theoretical models or conceptual frameworks, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, support qualitative and thematic data analysis, and interpret results. We have not used generative AI tools to generate synthetic data sets, to clean and reformat datasets or to assist with translation, and formulating mathematical claims, providing critical ingredients for proving mathematical claims and assisting in the writing of proofs are not applicable to this work. Additionally, we used generative AI tools to create or modify scientific figures, suggest experimental parameters, create or edit software code, create artifacts, draft parts of the research paper, summarize or analyse existing literature, brainstorm, source and search for information, edit the paper to improve readability, identify relevant literature, format references and suggest a structure for the paper. Generative AI is also part of the method itself: Claude Opus 5 produces and reviews the tool suites (§3.1). We have reviewed all AI-assisted work. Every number in the paper is generated by a script from the per-trial records and was checked by the authors against those records; qualitative statements about trajectories were checked against the raw logs; every cited paper was checked against its arXiv or proceedings page. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

References

Anthropic. Agent skills overview. https://platform.claude.com/docs/en/agents-and-tools/agent-skills/overview, 2025.

Cevheri Bozoglan, Yusuf Gundogdu, Abdullah Kaya, and Koray Sirin. What stops a small language model from driving a database agent. *arXiv preprint arXiv:2609.21341*, 2026.

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

Shiqi Chen, Jingze Gai, Ruochen Zhou, Jinghan Zhang, Tongyao Zhu, Junlong Li, Kangrui Wang, Zihan Wang, Zhengyu Chen, Klara Kaleb, Ning Miao, Siyang Gao, Cong Lu, Manling Li, Junxian He, and Yee Whye Teh. SkillCraft: Can LLM agents learn to use tools skillfully? *arXiv preprint arXiv:2603.00718*, 2026.

Chenhao Dang, Siyuan Xiong, Conghui He, and Weijia Li. SKILLER: Language-level reinforcement learning for reusable skill extraction in small language models. *arXiv preprint arXiv:2608.10538*, 2026.

Gen Dong, Yanjie Gao, Liqun Li, Tianyin Xu, Yu Hua, and Fan Yang. Agent skills can be harmful: An empirical study of skill-induced failures in LLM agents. *arXiv preprint arXiv:2608.11888*, 2026.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024.

Xiangyi Li et al. SkillsBench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026.

Cheng Qian, Chi Han, Yi R. Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023.

Akshit Sinha, Arvindh Arun, Shashwat Goel, Steffen Staab, and Jonas Geiping. The illusion of diminishing returns: Measuring long horizon execution in LLMs. In *International Conference on Learning Representations (ICLR)*, 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*, 2024a.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *International Conference on Machine Learning (ICML)*, 2024b.

Zhiruo Wang, Daniel Fried, and Graham Neubig. TroVE: Inducing verifiable and efficient toolboxes for solving programmatic tasks. In *International Conference on Machine Learning (ICML)*, 2024c.

Georg Wölflein, Dyke Ferber, Daniel Truhn, Ognjen Arandjelović, and Jakob Nikolas Kather. LLM agents making agent tools. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025.

Renjun Xu and Yang Yan. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *arXiv preprint arXiv:2602.12430*, 2026.

Yangjie Xu, Lujun Li, Lama Sleem, Niccolo Gentile, Yewei Song, Yiqun Wang, Siming Ji, Wenbo Wu, and Radu State. Agent skill framework: Perspectives on the potential of small to medium language models in industrial environments. *arXiv preprint arXiv:2602.16653*, 2026.

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

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R. Fung, Hao Peng, and Heng Ji. CRAFT: Customizing LLMs by creating and retrieving from specialized toolsets. In *International Conference on Learning Representations (ICLR)*, 2024.

Muru Zhang, Ofir Press, William Merrill, Alisa Liu, and Noah A. Smith. How language model hallucinations can snowball. In *International Conference on Machine Learning (ICML)*, 2024.

Shanshan Zhong, Yi Lu, Jingjie Ning, Yibing Wan, Lihan Feng, Yuyi Ao, Leonardo F. R. 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.

Appendix Contents

A Experimental Setup Details . . . 12
B Method Details . . . 13
B.1 Generation Contract . . . 13
B.2 What Each Condition Installs . . . 14
B.3 An Example Compiled Suite . . . 14
B.4 Cost of Compilation . . . 15
C Case Study . . . 16
D Task Families and Per-Family Results . . . 16
E Integrity Checks . . . 16
F Prefix-Intervention Details . . . 18
G Handoff Experiment Details . . . 18

A Experimental Setup Details

This section complements the setup in §2.1.

Harness and grader. We use the benchmark's official runner: it starts each instance's container, runs Claude Code (v2.1.197) inside it with the task instruction, and then runs the instance's tests. An instance passes when all of its tests pass. We keep the benchmark's limits (100 turns and 1,800 s of wall clock per trial) and change neither the tests nor the task files.

Executors. Table 4 lists the six executors. All are served with vLLM 0.29 with prefix caching, the same 81,920-token context window (the largest that fits the 9B executor on one GPU) and up to eight images per request. Claude Code is told the same window, so it compacts the conversation at the same point for every executor. The first run of Gemma 4 31B served more concurrent sessions than its KV cache could hold at full context, which slowed every session; all of its trials were discarded and re-run at two sessions per replica.

Proxy. Claude Code speaks the Anthropic Messages API; a LiteLLM proxy translates its requests to vLLM's OpenAI-compatible API. Four adjustments emulate the hosted API or keep the stack stable:

- system messages that Claude Code inserts mid-conversation are folded into the leading system prompt, as the Qwen chat template requires;
- `max_tokens` is clamped to the room left in the context window;
- PDFs read by the executor are expanded into per-page text plus images of the first eight pages, and only the eight most recent images are kept per request;
- the proxy's upstream HTTP clients are pinned, so that open streams are not closed when a client is recycled.

Invalid trials. A trial is invalid when it ends by an infrastructure error rather than by the executor's actions: a stalled response stream, a proxy or server error, an MCP server that fails to register, a

Table 4: Executors and their serving settings. Sampling follows each model's recommended settings.

Executor	Weights	Sampling
Qwen3.5 4B, 9B	bf16	temperature 1.0, top-p 0.95, top-k 20, presence penalty 1.5
Qwen3.5 27B	GPTQ-Int4	
Gemma 4 E4B, 12B	bf16	temperature 1.0, top-p 0.95, top-k 64
Gemma 4 31B	QAT w4a16	

container stopped from outside, or a grader that never runs the tests (for example, when its setup cannot download its test dependencies). Invalid trials are re-run and never counted. Across all reported conditions, 116 of 3,809 attempts (3.0%) were invalid and re-run.

Development gates. Before a suite ran on held-out instances, an executor had to call it on the development instance and produce a well-formed deliverable. When the 4B executor did not call the tools at this gate (for example, in enterprise search and schedule planning), the gate was run with the 9B executor instead; whether the 4B executor adopts the tools is then measured on the held-out instances.

B METHOD DETAILS

This section details the method of §3.1: the contract that producer and reviewer follow, what each condition installs in the executor's container, and one compiled suite as an example.

B.1 GENERATION CONTRACT

The producer and the reviewer receive the same contract; the family notes and the full producer and reviewer prompts are in the supplementary material. Its rules, condensed:

1. **Scope is the skill package, not the instance.** Before designing, list every capability the package teaches and the aspect of it that can vary between instances; every capability maps to a tool, and every varying aspect is an argument with a default, a value parsed from the task text, or discovered from the input files.
2. **Procedure, not content.** Tools encapsulate rules, formats, checks and mechanics that hold for every instance of the family. The content of the development instance (its theme, product, entities or files) may appear only as overridable defaults and in examples, never as a constant that silently shapes the output.
3. **A tool never authors the judgment-bearing part of the deliverable.** Tools compute, extract, transform, check, render from a given specification and fill from given facts. When the value of the deliverable is content that a person would write or decide, the tool gives the executor rules, structure and exact verification, and the executor writes the content.
4. **Names use the task's words.** A tool's name and the first sentence of its description reuse the verb and object of the instruction sentence that asks for the deliverable, not the development instance's entities.
5. **Variable cardinality and discovery.** Inputs that may vary in number are lists or globs; when an argument is omitted, the tool discovers candidates in the working directory and reports what it found and assumed. A missing optional input is reported and skipped, never a reason to abort.
6. **Compact reports with a next step.** Each call returns a short report (at most a few thousand characters) whose first line states the default verdict and whose last line names the next step, including which field of the deliverable a result belongs in.
7. **One-shot delegation of recurring judgments.** When a judgment recurs over many items, a preparing tool lists the minimal evidence for all items with a default verdict for each, the executor answers once, and an applying tool does the deterministic work.
8. **A checker.** One tool verifies the deliverable against the requirements stated in the instruction and reports each violation with its position and fix.
9. **Development discipline.** The producer tests the suite on the development instance and on at least three structural variants of it (entities added or removed, reference files missing or renamed, a different requested artifact), computing the expected outcomes independently, and answers in writing which argument or line would break on a sibling instance.
10. **Review.** The reviewer derives the expected outcomes from the development instance on its own, builds at least one structural variant that the producer did not list, and returns at most six changes; a revision may not remove a default verdict, only expose an argument to override it.

Table 5: The frozen tool suite for court-form filling (two tools). Descriptions are shortened to their first sentence.

Tool	Required arguments	First sentence of the description
`fill_small_claims_form_sc100`	`plaintiff_name`, `plaintiff_address`, `defendant_name`, `defendant_address`, `claim_amount`, `why_owed`, `filing_date` (7 of 45)	Fill the California Small Claims Court form SC-100 [...] at `/root/sc100-blank.pdf` from the case description and save the filled one to `/root/sc100-filled.pdf`.
`verify_small_claims_form_sc100`	none (0 of 42)	Re-read the filled Small Claims form `/root/sc100-filled.pdf` and check it against every requirement of the task [...].

B.2 What Each Condition Installs

All conditions run the same harness and task prompt; they differ only in what is installed in the container. The skill channel places the official package under `~/.claude/skills`, where Claude Code lists it by name and description and loads it only when the executor invokes it. In the other conditions, the material below is written to `/root/CLAUDE.md`, which Claude Code loads into context on every turn.

Condition	Content of `/root/CLAUDE.md`
Skill in context	A heading, one lead sentence, the text of every `SKILL.md` of the official package verbatim (with the location of its supporting files), and a closing sentence asking the executor to follow the skill and finish the deliverable.
Tools + card	A heading, one lead sentence, one line per tool with its name, required arguments and the first sentence of its description, and a closing sentence (see Appendix B.3). The tools are registered through an MCP server; without the card, only this registration remains.
Suite as text	The framing of the skill in context; each tool becomes a numbered step with its full description and parameters, without code (ablation d).
Suite as scripts	The text above plus, for each step, a command that runs the tool's code on a JSON argument and prints its report (ablation c).

B.3 An Example Compiled Suite

Table 5 lists the frozen suite for court-form filling, and the box below shows the card written to `CLAUDE.md`. It shows how the work is divided: the executor reads the case description and passes its facts as arguments (18 in the held-out trial of Figure 6); the tool maps them to the fields and check boxes of the PDF form, and reports every default it applies together with the argument that overrides it.

Card written to `/root/CLAUDE.md` (tools + card), abridged

```
# Tools installed for this task

This environment ships native tools that carry out the task's core steps.
Start with them:

- `mcp__maker_tools__fill_small_claims_form_sc100` (plaintiff_name,
  plaintiff_address, defendant_name, defendant_address, claim_amount,
  why_owed, filing_date): Fill the California Small Claims Court form
  SC-100 (...) at /root/sc100-blank.pdf from the case description and
  save the filled one to /root/sc100-filled.pdf.
- `mcp__maker_tools__verify_small_claims_form_sc100` (defaults work for
  this task): Re-read the filled Small Claims form ...

Each call returns a report whose last line names the next step; follow it,
then finish the task's deliverable.
```

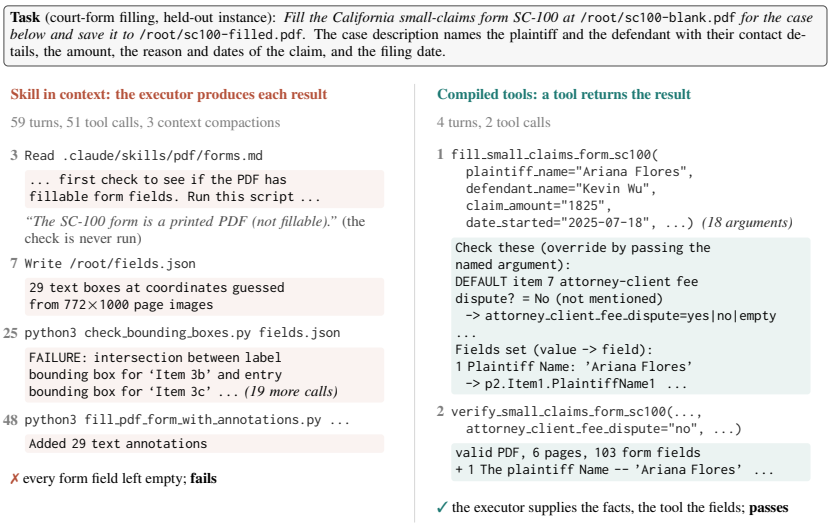


Figure 6: **The same task, carried out by the same 4B executor from text and with compiled tools.** Abridged excerpts from two held-out trials on one court-form instance; numbers are tool-call indices, and symbols are rendered in ASCII.

A call returns a compact report with what was set, why, and what the executor should check. An abridged report from the held-out trial of Appendix C (the first line with the default verdict is omitted; symbols are rendered in ASCII):

Report returned by `fill_small_claims_form_sc100` (abridged)

```
! Check these (override by passing the named argument):
  - ! DEFAULT item 7 attorney-client fee dispute? = No (not mentioned)
      -> attorney_client_fee_dispute=yes|no|empty
  - ! DEFAULT item 10 claim for more than $2,500? = No (amount 1825 <= 2500)
      -> claim_more_than_2500=yes|no|empty
  [...]
Checkboxes checked (reason):
  [x] p3.Item4.Checkbox50 -- 4 Have you asked the defendant to pay? Yes
  [x] p3.Lia.Checkbox5cb -- 5a (1) where the defendant lives or does business
  [...]
Fields set (value -> field):
  1 Plaintiff Name: 'Ariana Flores' -> p2.Item1.PlaintiffName1
  1 Street: '1724 Kifer Rd Apt 11' -> p2.Item1.PlaintiffAddress1
  [...]
```

B.4 Cost of Compilation

We measure the cost of compiling a family from the logs of its producer and reviewer sessions, including the revision. Active time sums the gaps between consecutive logged events, with gaps longer than ten minutes capped, so that pauses do not count. Token counts are taken once per model response. Over the 15 families whose sessions carry token accounting, compiling a family takes a median of 63 minutes of active agent time (range 36–228), 172k output tokens (114k–749k) and 10M input tokens (6M–61M), over 90% of them read from the prompt cache. At the API list prices of Claude Opus 5 (per million tokens: $5 input, $6.25 cache write, $0.50 cache read, $25 output), this is a median of $12 per family ($7–$67) and $256 for all 15. The largest values belong to Python-to-Scala, whose count includes the discarded session (Appendix E). The executors run locally, so inference on held-out instances adds no API cost.

Table 6: Task families. n: held-out instances; tools: size of the frozen suite.

Family	Task	n	Tools
Chinese poem	compose a seven-character regulated verse on a theme	4	3
Court-form filling	fill the California small-claims form SC-100 from a case	5	2
DBSCAN tuning	find the Pareto frontier of DBSCAN hyperparameters	4	2
Dependency check	audit third-party dependencies for vulnerabilities	4	2
Earthquake plate	find the earthquake furthest from a plate boundary	5	3
Enterprise search	retrieve employee IDs and URLs from enterprise data	5	3
Financial analysis	compare hedge-fund holdings across two quarters	5	3
GitHub analytics	summarize large-PR activity of a repository	4	2
Messy files	sort over 100 papers into subject folders	5	3
NLP reproduction	implement a paper's loss function in its code base	2	3
Offer letter	fill a Word offer-letter template from employee data	5	2
Poster design	render a branded technical visual and its parameters	4	2
Python-to-Scala	translate a Python tokenizer into Scala	1	2
Schedule planning	propose meeting times from a calendar and reply	4	3
Security bug fix	patch a remote-code-execution bug in Apache Druid	2	3
Stock plots	build a D3 web page from stock data	4	3
Temperature simulation	calibrate a lake model to observed temperatures	4	3
Travel planning	build a multi-city itinerary under constraints	4	3
Video counting	count objects in the key frames of a game video	4	3
Weighted GDP	fill workbook formulas for a weighted GDP statistic	5	2
Total		80	52

C Case Study

Figure 6 shows excerpts from two held-out trials of the 4B executor on the same instance of court-form filling; the notes in Figure 3 are taken from them. With the skill in context, the executor decides without running the skill's check that the form has no fillable fields, places text boxes at guessed coordinates, spends some twenty calls repairing their overlaps, and leaves every form field empty. With compiled tools, it passes the facts of the case as arguments; the tool fills the form and lists the defaults it applied, which the executor confirms in the verifying call.

D Task Families and Per-Family Results

Table 6 lists the 20 families of SkillLearnBench, and Figure 7 breaks the results of Table 2 down by family.

E Integrity Checks

The compiled tools could inflate results if they carried information from held-out instances. This section documents what the producer and reviewer could see, audits the frozen suites for answers, and describes how the suites were developed.

What the producer and reviewer saw. We read the logs of all 84 producer and reviewer sessions behind the 20 frozen suites and checked every file access and command. None reads a test, an official solution or an instance other than the development instance. The only flagged accesses are to a test specification that the Python-to-Scala development container itself provides to the executor.

No answers in the tools. We compared every string literal of the 20 frozen suites with the tests and official solutions of all held-out instances. No held-out answer occurs in any suite; the only overlaps are generic strings such as form field names and library class names.

A contaminated workspace, discarded. The Python-to-Scala task package ships a reference translation in its environment directory, beyond the files its container copies in. The first producer session for this family read it. We discarded that suite, rebuilt every family's workspace from the

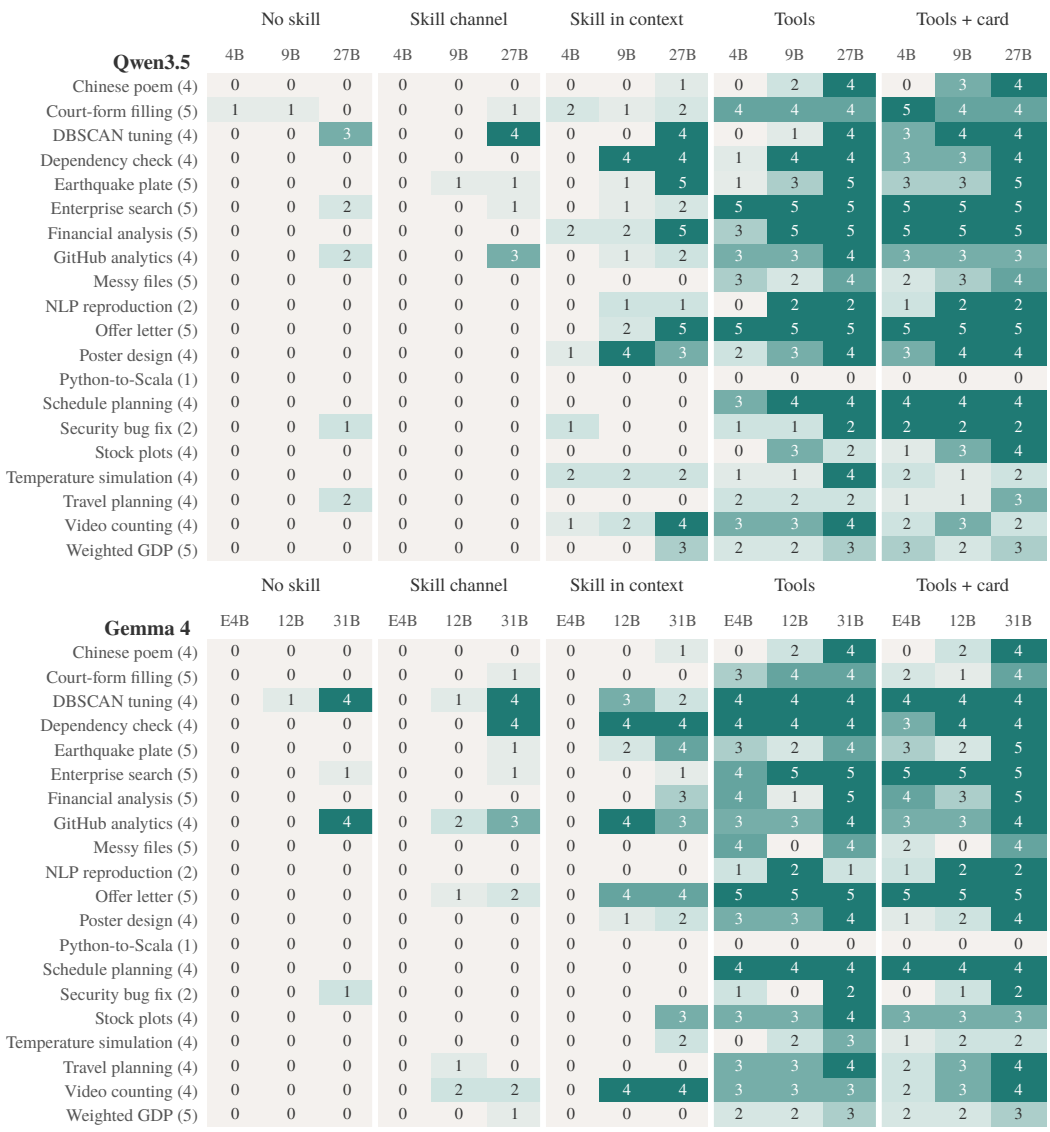


Figure 7: **Held-out pass counts per family (run 1).** Each cell counts the passed instances of a family, with the number of held-out instances in parentheses after its name; darker cells pass a larger share. Tools: compiled tools without the card.

files its container actually copies, and regenerated the suite in a fresh session; no other workspace contained such material.

Development and evaluation families. We split the 20 families into development and evaluation families before any suite for an evaluation family was built. On 15 development families we developed the contract and the family notes, iterating with the results of earlier suites on all of their instances; their results therefore show that the method works, not that it generalizes. The remaining five families (temperature simulation, GitHub analytics, Python-to-Scala, security bug fix and NLP reproduction) were fixed in advance for evaluation: their results never informed the contract, the notes or any other part of the pipeline. Every family's final suite was frozen before its held-out runs. Table 7 shows that the gain holds on the evaluation families alone.

Table 7: Held-out passes on the five evaluation families (13 instances). Discordant pairs: instances passed only with tools : only with the skill in context.

	Qwen3.5			Gemma 4		
	4B	9B	27B	E4B	12B	31B
Skill in context	3	4	5	0	4	5
Tools + card	8	8	9	5	8	10
Discordant pairs	5:0	6:2	4:0	5:0	5:1	6:1

F Prefix-Intervention Details

This section details the controlled interventions of §2.3 (Figure 4a).

Scaffold. An intervention needs three things that a production harness does not give directly: the exact message list of a prefix, the ability to replace one observation in it, and a single sampled step from the edited prefix. We therefore use a lightweight ReAct scaffold in which the conversation is a plain list of messages under our control, with the instance's tests hidden from the executor. The executors are the same Qwen3.5 models with the same sampling settings as in the main experiments.

Skill. Instead of the official package, the executor receives a plain-language statement of the family's procedure without code blocks. Official packages include scripts and code snippets that an executor can copy; without them, the next step has to be written against the intermediate result in context, which is what the intervention varies.

Prefixes. We cut 4B trajectories on held-out instances of enterprise search at the step where the list of shared posts with their timestamps first becomes visible. A prefix is *clean* when that observation is compact (under 3,500 characters), and *messy* otherwise; we obtain 6 clean and 19 messy prefixes. The *swap* condition replaces the last observation of a messy prefix with a clean list of the shared posts built from the instance's data, and leaves the rest of the prefix unchanged.

Outcome. A sampled next step succeeds when it computes the requested time window from the timestamps in context, that is, when its code uses a timestamp from the context together with a comparison. We draw eight samples per prefix, condition and executor.

G Handoff Experiment Details

This section details the handoff experiment of §2.4 (Figure 4b). We write *4B→27B* for a trial in which the 4B executor works the first K turns and the 27B executor continues, and *27B→27B* for the control in which the 27B executor works both segments.

Instances. The 30 instances are all held-out instances that the 27B executor passes with the skill in context in both runs; the selection uses only the 27B executor's results. By family: dbscan tuning, dependency check, earthquake plate, financial analysis and offer letter (4 each), video counting (3), court-form filling and temperature simulation (2 each), and poster design, enterprise search and weighted GDP (1 each).

Implementation. The first executor runs Claude Code with a fixed session identifier and a limit of K turns. The receiver then resumes the same session in the same container, with the prompt "Continue working on the task.", the remaining $100 - K$ turns and the remaining wall clock; only the model behind the proxy changes. Before grading, a check confirms that the session was resumed, that the second segment ran on the receiver and that both segments are in the log; a trial that fails the check is invalid, and all 120 trials passed it. $K \in \{4, 8\}$ was fixed before a two-instance pilot, which only checked the mechanism.

Registration. The registration is an internal document, not an entry in a public registry. It fixes the design, the predictions and the falsification criterion; it was written before any handoff trial and

Table 8: Registered predictions of the handoff experiment and their outcomes. Discordant pairs: instances passed only by 27B→27B : only by 4B→27B.

	Prediction	Result	Outcome
P1	4B→27B passes less often than 27B→27B at both K (sign test on instance pairs, pooled over K)	discordant pairs 10:3 and 13:2; $p = 0.0009$	met
P2	The gap is larger at K=8 than at K=4	37 vs. 23 points	met
P3	27B→27B passes at least 85% of instances	87% (K=4), 80% (K=8)	met at K=4 only
Falsif.	At K=8, 4B→27B comes within 10 points of 27B→27B	37 points apart	not triggered

not edited afterwards, and results were recorded in a separate file. We release it unchanged with its SHA-256 hash, together with the trial logs; it was last modified more than two hours before the first handoff trial, as its file metadata and the trial timestamps show. Table 8 lists the registered predictions and their outcomes. As registered, the failure of P3 restricts the interpretation to the paired contrast between 4B→27B and 27B→27B.

Secondary analysis: early ends. As registered, the main analysis includes every trial. At $K = 4$, every first segment used all four turns. At $K = 8$, the first executor ended on its own before eight turns in 5 of 30 trials for the 4B executor and 6 of 30 for the 27B executor; the 27B executor then passed all six, since it had finished the task itself. Excluding the eight instances with an early end in either condition, 27B→27B passes 17 of 22 and 4B→27B 10 of 22 (discordant pairs 9:2), so the contrast does not depend on these trials.

Secondary analysis: what the receiver inherits. We classify each failed 4B→27B trial by the files each segment writes, through file-editing tools or shell redirection. At $K = 4$, the 4B executor wrote no file in 8 of the 11 failed trials, so what the receiver inherits is mainly the conversation: the 4B executor's reads, outputs and plan. At $K = 8$, it wrote no file in 3 of 17 failed trials; in 10 the receiver edits or rewrites a file the 4B executor wrote, typically its analysis script, and in 4 it leaves those files aside.