HINT: LEARNING THE HARNESS PROTOCOL FOR INTERFACE-INVARIANT SWE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) agents now resolve real GitHub issues by exploring a repository, editing files and running tests. Each runs inside a *harness* such as Claude Code, Codex CLI, OpenCode or OpenHands SDK, the scaffold that fixes which tools the model sees, their names, and how calls and results are formatted. These details differ across harnesses and they matter: with model and tasks fixed, changing only the harness moves SWE-bench Verified accuracy by up to 9.0 points. Because training trajectories come from one harness, fine-tuning learns the interface with the task, and its gains largely stay there. This dependence is avoidable because the differences are superficial: 578,856 real tool calls from four production harnesses all reduce to 12 semantic actions, which we call the HARNESS PROTOCOL. A harness is this protocol plus a *surface* of tool names, documentation, output and turn formatting, and a trajectory rendered into any surface parses back to the same actions (1.87M checks, no mismatches). HINT (Harness-INvariant Training) parses each real trajectory into the protocol and re-renders it under many sampled surfaces, some with meaningless tool names, so that one harness's data trains an agent for any harness with no new task data. Fine-tuning Qwen3.6-35B-A3B on Claude Code trajectories, HINT matches a step-matched control on Claude Code itself, leads under every unseen production harness by up to 8.7 points, narrows the SWE-bench Multilingual accuracy range across five harnesses from 7.4 to 1.6 points, and is on a par with training on real trajectories from four harnesses. Harness variation is thus a nuisance factor that training can marginalize over, not a distribution shift the agent must absorb.

1 INTRODUCTION

Recent years have seen rapid progress in LLM agents for software engineering. Given an issue reported against a real GitHub repository, an agent such as Claude Code (Anthropic, 2025), Codex CLI (OpenAI, 2025), OpenCode (OpenCode contributors, 2025) or OpenHands SDK (Wang et al., 2025b) explores the codebase, edits files, runs tests, and returns a patch, and SWE-bench (Jimenez et al., 2024) has become the standard measure of how often that patch is correct. Underneath every such agent is a language model wrapped in a scaffold: a system prompt that documents a tool set, a protocol for emitting calls, an executor, and a formatter that returns observations. This wrapper, the *harness* (also called a scaffold), is what the model actually sees.

The four harnesses aforementioned all let an agent run shell commands and edit files, but they disagree on nearly every detail of how: whether reading a file is a first-class tool or a shell command, whether that tool is called `Read` or `read` or `file_editor` with `command="view"`, whether output carries line numbers, and whether an episode ends with a `finish` call or a plain reply. Changing *only* the harness for a fixed instruction-tuned model and task set (Table 4) moves SWE-bench Verified accuracy by up to 9.0 points, from 58.6% to 67.6% for Qwen3.5-35B-A3B across five harnesses. Training makes the dependence worse: trajectories are collected in whichever harness the data pipeline uses, and supervised fine-tuning (SFT) on them teaches the task *and* the surface at once, so a different harness at deployment is, from the model's perspective, a change of task (Figure 1, left). Fine-tuning Qwen3.6-35B-A3B on Claude Code trajectories raises its Claude Code accuracy on SWE-bench Verified from 69.0% to 78.4% but does not improve its Codex CLI accuracy (68.0% before, 66.7% after), widening its spread across harnesses from 3.4 to 11.7 points (Section 4.3).

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.

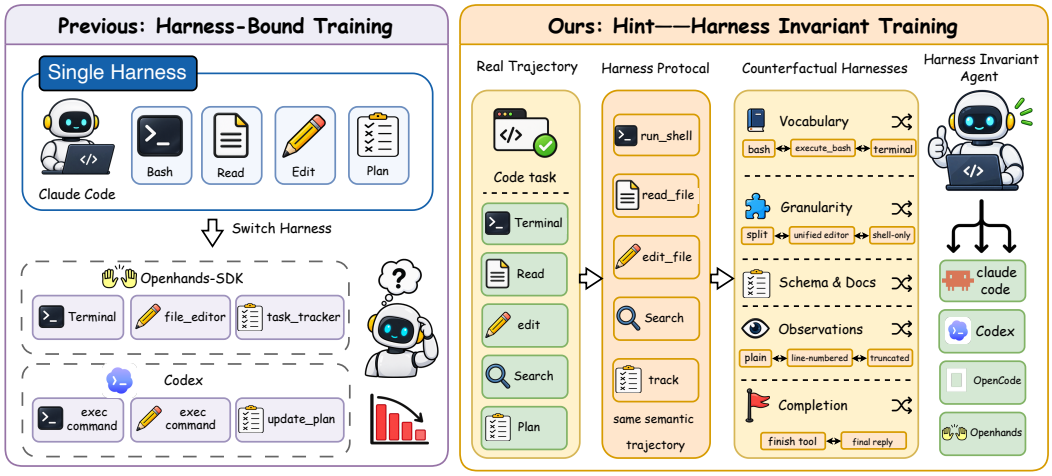


Figure 1: **Left:** standard training binds the policy to the harness its trajectories were collected in, and moving the model to a harness with different tools and conventions degrades performance on the same tasks. **Right:** HINT parses each real trajectory down to the HARNESS PROTOCOL, the core all harnesses share, and re-renders it into many counterfactual harnesses whose surface axes are independently resampled, training an agent invariant to the surface rather than bound to it.

We argue that this shift is avoidable, because the disagreement between harnesses is superficial in a measurable sense. Normalizing 578,856 tool calls from the four harnesses into semantic actions, we find that a vocabulary of 12 covers 99.96% of them (Table 1): the harnesses are not doing different things, they are naming and packaging the same things differently. We make this core explicit and call it the HARNESS PROTOCOL. A concrete harness is then the protocol plus a *surface*, five axes that can be set independently: tool granularity and naming, call syntax, prompt documentation, observation formatting, and turn conventions. The abstraction is not lossy: rendering a protocol trajectory into a surface and parsing it back recovers the original actions, with no mismatches in 1.87M render–parse checks over all 578,856 actions (Section 2). This is a *descriptive* claim about harnesses that already exist, not a proposal for a new standard.

Exact invertibility is what turns the protocol from an analysis tool into a training signal. If a real trajectory can be parsed into protocol form and re-rendered in an arbitrary surface, then a single collected episode represents an entire family of equivalent episodes. We introduce HINT (Harness-INvariant Training), which trains on that family (Figure 1, right). Counterfactual harness rendering (CHR) re-renders every trajectory under freshly sampled surface configurations, and interface anonymization replaces tool names with meaningless identifiers on a fraction of samples, so that reading the documentation rather than recognizing the name becomes the only reliable strategy. HINT needs no new task data and no architectural change.

We fine-tune Qwen3.6-35B-A3B on Claude Code trajectories and evaluate under real harnesses that training never showed as coherent wholes, against a step-matched control trained on the same trajectories in their original harness for the same number of optimizer steps. HINT matches the control on the source harness, is ahead on both benchmarks under each of the three unseen production harnesses, by up to 8.7 points, and on SWE-bench Multilingual its accuracy varies by 1.6 points across five harnesses against the control's 7.4 (Section 4.3). Harness variation, in other words, is not a property of the task that a model must absorb, but a nuisance factor that a training pipeline can marginalize over. Our contributions are:

- **The HARNESS PROTOCOL**: harness choice alone moves SWE-bench Verified by up to 9.0 points for a fixed model and task set, yet 578,856 real tool calls from four harnesses reduce to a 12-action vocabulary with invertible surface rendering, which we formalize as the protocol (Sections 2 and 4.2).
- **HINT**: training on counterfactual renderings of existing trajectories, with no new task data and no architecture change (Section 3).

- **Experiments**: a step-matched evaluation under five harnesses (Section 4.3), a comparison with real trajectories from four harnesses (Section 4.4), and analyses of test-time anonymization and of one surface axis at a time (Sections 4.5 and 4.6).

2 THE HARNESS PROTOCOL

2.1 SEMANTIC ACTIONS AND SURFACES

Reading a file is a `Read` call in Claude Code, `read` in OpenCode, `file_editor` with `command="view"` in OpenHands SDK, and `sed -n` or `cat` under Codex CLI; the protocol records all four as one action, `read_file`, and treats the rest as a property of the harness.

Semantic actions. Let $\mathcal{A}$ be a finite vocabulary of semantic actions, each a verb with a typed argument record. We take $\mathcal{A}$ to be the 12 actions of Table 1, obtained by normalizing four harnesses' tool sets rather than by design; only 0.04% of 578,856 real calls fall outside it. A *protocol trajectory* is a sequence $\tau = \big[(c_t, a_t, o_t)\big]_{t=1}^{T}$ where $a_t \in \mathcal{A}$ is an action, o_t the semantic content of the resulting observation with formatting removed, and c_t the agent's reasoning text.

Harnesses as renderings. A harness h is a rendering function R_h mapping a protocol trajectory to a surface trajectory: the concrete token sequence a model consumes and produces. We use *surface* and *interface* interchangeably for this rendering, and *harness* for the whole system that also executes the actions. Writing $\mathcal{H}$ for the space of harnesses, we factor the surface into five axes,

$$\mathcal{H} \;=\; A_1 \times A_2 \times A_3 \times A_4 \times A_5, \tag{1}$$

with A_1 tool granularity and naming, A_2 call syntax, A_3 system-prompt template and tool-documentation style, A_4 observation formatting, and A_5 turn and completion conventions. A harness is a point $h \in \mathcal{H}$; sampling a point is sampling a synthetic harness (Appendix A.7 shows one trajectory under three such points).

Table 2 places the four production harnesses in this space; the extremes are instructive. OpenHands SDK exposes 4 tools; Claude Code exposes 21. Codex CLI declares 13 tools but *none* of them reads, edits or searches files: 97% of its calls go to a single `exec_command` entry point, and of those commands 27% are file reads (`sed -n`, `cat`, `head`), 18% searches (`grep`, `find`) and 10% patch applications. More than half of Codex CLI's shell traffic is, semantically, what other harnesses expose as first-class tools. Surface variation of this magnitude over an identical semantic core is the regime in which a surface-bound policy fails and an interface-invariant one should not.

2.2 ROUND-TRIP EXACTNESS

Our renderer realizes the call syntax A_2 either as native function calling, the structured format production harnesses deploy and the one used for training (Section 3.1), or as one of four text families used for the measurement below: JSON, XML and YAML blocks with named tools, and a shell-only family in which file operations are lowered to shell commands.

Table 1: The 12-action vocabulary from normalizing four harnesses, with its share of 578,856 real calls; `unknown` is the residual outside the vocabulary. Percentages are rounded independently.

Action	Share	Action	Share
`run_shell`	71.52%	`edit_file/write`	0.84%
`read_file`	13.59%	`web_fetch`	0.39%
`edit_file/str_replace`	7.22%	`web_search`	0.14%
`search`	3.26%	`delegate`	0.04%
`submit`	1.62%	`edit_file/undo`	0.01%
`task_track`	1.34%	`edit_file/insert`	$< 0.01\%$
`unknown` (residual, outside the vocabulary): 0.04%			

Table 2: Four production harnesses agree on semantics and disagree on everything else. "Tools" counts the declared schema; granularity is the A_1 setting, where shell only means that files are read, searched and edited through shell commands. Tool names are those of the parsed releases; the evaluated releases rename a few of them (Appendix D).

Harness	Tools	Granularity	Shell tool	File read
OpenHands SDK	4	unified editor	`terminal`	`file_editor`
OpenCode	11	split	`bash`	`read`
Codex CLI	13	shell only	`exec_command`	— (via shell)
Claude Code	21	split	`Bash`	`Read`

Table 3: Round-trip exactness. Every action of each corpus is rendered under sampled surface configurations (four per action for the mixed set, three otherwise) and parsed back; exact means that the verb and all arguments match, $\simeq$ equality up to the operational equivalence of Proposition 1, which arises only in the shell family. The 232 calls outside the vocabulary are not rendered.

Corpus	Trajectories	Actions	Checks	Exact	$\simeq$	Mismatches
Mixed set (four corpora)	2,832	130,537	522,004	492,429	29,575	0
Claude Code (large)	5,000	295,613	886,407	828,974	57,433	0
Codex CLI	2,041	152,706	457,962	455,040	2,922	0
Total	9,873	578,856	1,866,373	1,776,443	89,930	**0**

Proposition 1 (round-trip exactness)**.** *Under the premises (P1)–(P4) of Appendix A.6, for every $h \in \mathcal{H}$ realized by our renderer there is a parser P_h with $P_h \circ R_h = \mathrm{id}$ on the action sequence of every protocol trajectory: exactly for the three syntax families with named tools, and up to an operational equivalence $\simeq$ (a file read and the shell command that performs it; a final reply absorbed into the submission message) for the shell-only family and for the plain-reply completion convention of A_5. In particular R_h is injective on actions, modulo $\simeq$.*

Appendix A proves this by construction, one stage at a time: granularity and naming form an injective table lookup, and each syntax family is a decodable encoding of the named record. The proof covers the design; Table 3 checks that the implementation meets its premises on real data, rendering every action of the 9,873 parsed trajectories under three or four sampled configurations and parsing the result back: 1.87M checks with zero mismatches, 95.2% of them exact and the remaining 4.8%, all in the shell family, equal up to $\simeq$. The composition $T_{h\to h'} = R_{h'} \circ P_h$ is therefore a semantics-preserving translation between any two harnesses, and the protocol acts as a normal form.

3 HINT: HARNESS-INVARIANT TRAINING

A policy π is *interface-invariant* if its induced semantic behaviour is unchanged by surface transport: for all h, h' and all protocol states x, $\tilde{\pi}(R_{h'}(x)) = \tilde{\pi}(R_h(x))$, where $\tilde{\pi}$ denotes the policy's output pushed back to protocol level. Section 4 measures deviation from this equality, and HINT is a training method that pursues it, whence its name. HINT has two components, counterfactual harness rendering and interface anonymization, the second implemented as one tier of the first's name sampler; both touch only the data; the model architecture and training objective are unchanged. Figure 2 traces the pipeline end to end. Because real harnesses cannot be perturbed one axis at a time, we also implement CHAMELEON, an executable harness in which every surface property is a configuration parameter drawn from the five axes of Eq. (1); it parses calls, executes them in a real workspace and formats observations according to its configuration, serving as the instrument for single-axis experiments (Section 4.6) and as an evaluation environment under synthetic surfaces.

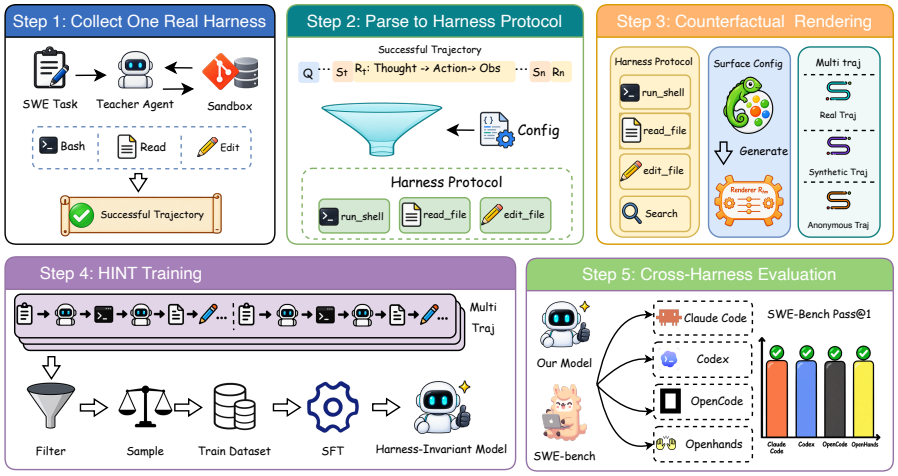


Figure 2: The HINT pipeline: trajectories collected with a teacher agent in one real harness are parsed down to the HARNESS PROTOCOL, re-rendered under freshly sampled surface configurations with real, synthetic or anonymized tool vocabularies, filtered and sampled to a token budget, and used for fine-tuning; the model is then evaluated under real harnesses never seen in training.

3.1 COUNTERFACTUAL HARNESS RENDERING

Given a corpus of real trajectories collected in a single harness h_0, we parse each into protocol form and re-render it under M configurations sampled independently from $\mathcal{H}$:

$$\mathcal{D}_{\text{CHR}} = \left\{ R_h(P_{h_0}(\tau)) \;:\; \tau \in \mathcal{D}_{\text{real}},\; h \sim \text{Unif}(\mathcal{H}_{\text{train}})^M \right\}, \tag{2}$$

where $\mathcal{H}_{\text{train}} \subset \mathcal{H}$ samples four axes and fixes the call syntax A_2 to native function calling, so that every rendered sample carries the `tools` schema and structured `tool_calls` production harnesses deploy and merges directly with ordinary agent SFT data. Every sample keeps the action sequence and the observation content of a real, verified episode; only their rendering is synthetic, and training renderings never truncate observations. What a rendering does not carry is the executor behind a surface, such as a tool's preconditions or the calls a harness rejects; Appendix B.4 measures these as interface errors on the real harnesses. Rendering is domain randomization (Tobin et al., 2017) applied to the agent interface rather than to visual appearance, and two results motivate it. The domain-generalization bound of Ben-David et al. (2010) charges the divergence between training and deployment domains, which training over a distribution of surfaces is meant to reduce. And the identifiability assumption behind augmentation-based representation learning, that augmentation perturbs *style* and leaves *content* untouched (von Kügelgen et al., 2021), holds on the action channel by Proposition 1.

3.2 INTERFACE ANONYMIZATION

Sampling from real vocabularies alone leaves a shortcut intact: each semantic function has only a handful of attested names, so a model can still map `Bash`, `bash` and `exec_command` onto "the shell tool" by memorization. We therefore draw tool names from three tiers: names harvested verbatim from real harnesses but *recombined across* them ($\approx 43\%$ of samples), plausible names that no parsed harness uses ($\approx 27\%$), and meaningless pronounceable identifiers (30%). The third tier is the anonymization component: every tool name in such a sample is replaced, while the documentation, which is never anonymized, becomes the only channel carrying the information needed to act.

Table 4: Harness choice alone moves accuracy. Pass@1 (%) of three open-weight models on SWE-bench Verified (V, 500 tasks) and SWE-bench Multilingual (M, 300 tasks) under four production harnesses and CHAMELEON (fixed configuration), one evaluation per cell. Spread: max minus min.

	Qwen3.5-35B-A3B		Qwen3.6-35B-A3B		KAT-Coder-V2.5-Dev	
Harness	V	M	V	M	V	M
Claude Code	64.2	54.7	69.0	62.3	69.4	57.3
OpenCode	66.4	54.3	71.4	68.3	69.8	58.7
OpenHands SDK	67.6	58.3	71.0	64.3	68.8	60.3
Codex CLI	58.6	49.7	68.0	61.3	68.6	59.3
CHAMELEON	67.2	55.0	70.8	62.3	67.2	60.0
Spread	9.0	8.7	3.4	7.0	2.6	3.0

4 EXPERIMENTS

4.1 SETUP

Model and data. We fine-tune Qwen3.6-35B-A3B (Qwen Team, 2026b) on 9,366 Claude Code trajectories collected with GLM-5.3 (Z.ai, 2026) as teacher on SWE-rebench-V2 instances (Badertdinov et al., 2026), one per instance, deduplicated against both evaluation benchmarks. The control trains on these trajectories as collected, and HINT on their CHR renderings alone: $M = 3$ counterfactual surfaces per trajectory, 26,031 samples in all with 21,476 distinct tool names, 30% of them anonymized, all in native function calling (surface statistics in Appendix D). Both runs are full-parameter fine-tuning with a global batch of 128 packed sequences of up to 256K tokens, learning rate 5×10^{-5} and cosine decay.

Step-matched control. Rendering multiplies the data: the rendered set holds about $1.7\times$ the original tokens, so a gain could come from extra optimization rather than surface diversity. We therefore match optimizer steps: the step-matched control trains on the original Claude Code trajectories for three epochs, 67 steps, and HINT for the same 67 steps, about 1.8 passes over its rendered set; with sequence packing, equal steps at equal batch size are also approximately equal training tokens. The two runs differ only in their data: the control sees each trajectory three times in its original surface, HINT in three different surfaces, none of them Claude Code's own.

Evaluation. We report pass@1 on SWE-bench Verified (Jimenez et al., 2024), 500 human-validated tasks from 12 Python repositories, and on SWE-bench Multilingual (Yang et al., 2025), 300 tasks in nine other programming languages; we call them Verified and Multilingual. Every model runs under the four production harnesses of Table 2 and under CHAMELEON with one fixed configuration, served with vLLM (Kwon et al., 2023) on one serving stack. Section 4.2 evaluates Qwen3.5-35B-A3B (Qwen Team, 2026a), Qwen3.6-35B-A3B and KAT-Coder-V2.5-Dev (Huang et al., 2026) once per cell, Section 4.3 averages three evaluations per cell, and the later experiments state their own conditions. We call OpenCode, OpenHands SDK and Codex CLI unseen because training never presented any of them as a coherent harness (Appendix E counts the rendered samples that carry one of their tool names). Appendix D summarizes the settings of every experiment in Table 8.

4.2 HARNESS CHOICE ALONE

Table 4 establishes the phenomenon. The ordering is not explained by tool count or interface simplicity: Codex CLI, which declares 13 tools but routes file operations through a shell, is the lowest of the four production harnesses on Verified for all three models, while OpenHands SDK, with 4 tools, is the best or second best for the two Qwen models. The size of the spread is a property of the model: 9.0 points for Qwen3.5-35B-A3B, 3.4 for Qwen3.6-35B-A3B and 2.6 for KAT-Coder-V2.5-Dev on Verified, in the same order on Multilingual (8.7, 7.0, 3.0). KAT-Coder-V2.5-Dev, post-trained with harness randomization (Huang et al., 2026), shows the smallest spread; HINT aims to obtain the same property from training data alone.

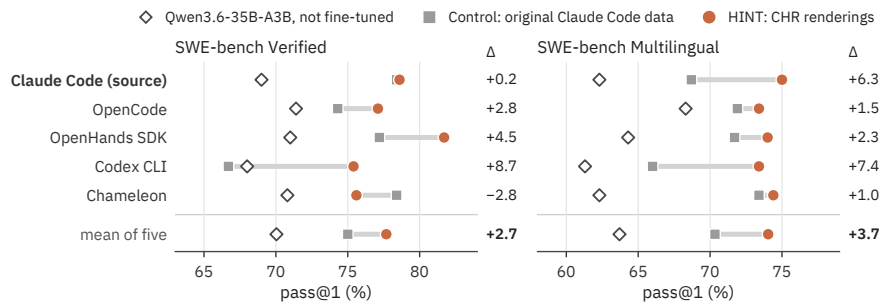


Figure 3: HINT versus the step-matched control, both fine-tuned from Qwen3.6-35B-A3B: pass@1 (%) on SWE-bench Verified and SWE-bench Multilingual under each harness, the fine-tuned models averaged over three evaluations. Each row places the base model (open diamond), the control (grey square) and HINT (orange circle); the bar joins the two fine-tuned models, and the number to its right is HINT minus control. Full numbers, with the worst harness and the spread, are in Table 7.

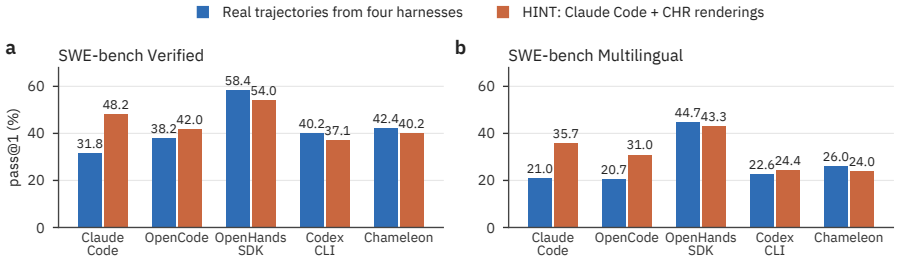


Figure 4: HINT versus training on real trajectories from four harnesses, both fine-tuned from Qwen3-30B-A3B-Instruct-2507 for equal steps, under the four production harnesses and CHAMELEON (pass@1, one evaluation per cell). **(a)** SWE-bench Verified. **(b)** SWE-bench Multilingual.

4.3 HINT VERSUS A STEP-MATCHED CONTROL

Figure 3 gives the result (full numbers in Table 7). Fine-tuning on one harness raised accuracy on that harness and made the model more dependent on it: on Verified the control gains 9.4 points over the base model under Claude Code, the harness its data came from, but none under Codex CLI, so its spread across harnesses grows from 3.4 to 11.7 points. HINT improves on the base model by 7.6 points on Verified and 10.3 on Multilingual on average, against the control's 5.0 and 6.6, and its gain is spread across harnesses. On the source harness it is level with the control on Verified (78.6% against 78.4%) and ahead on Multilingual (75.0% against 68.7%), although it never trained on Claude Code's own surface. Under the three unseen production harnesses it is ahead on both benchmarks, by 2.8, 4.5 and 8.7 points on Verified and by 1.5, 2.3 and 7.4 on Multilingual; the largest gap is under Codex CLI, whose shell-only surface lies outside the surfaces CHR renders for training (Appendix E). The advantage is largest where the control is weakest: HINT's worst harness is 8.7 points (Verified) and 7.4 points (Multilingual) above the control's, and on Multilingual its five accuracies lie within 1.6 points (73.4–75.0%), a tighter range than the base model's own 7.0. HINT is ahead in eight of the ten cells by at least one point, within a point of the control in one (Claude Code on Verified), and behind in one.

That one cell is CHAMELEON on Verified, where the control leads by 2.8 points; CHAMELEON's configuration comes from the space HINT's training surfaces were sampled from, so this is not a failure to reach an unseen surface, and we have no explanation for it. An earlier run on a weaker base model with a different recipe (Appendix C) showed HINT's advantage on Multilingual only.

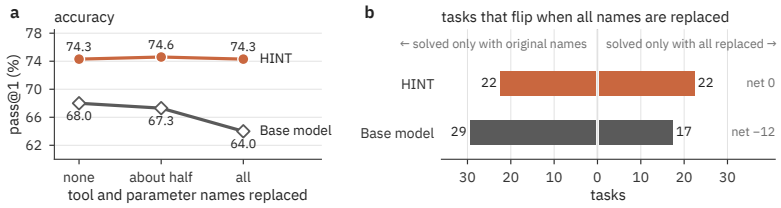


Figure 5: Anonymizing tool and parameter names under CHAMELEON on SWE-bench Multilingual, for Qwen3.6-35B-A3B before fine-tuning and the HINT model of Section 4.3, one evaluation per setting. **(a)** Pass@1 as names are replaced. **(b)** Tasks that flip with all names replaced.

4.4 REAL TRAJECTORIES FROM MORE HARNESSES

A direct alternative to rendering counterfactual surfaces is to collect real trajectories in several harnesses. We compare the two on Qwen3-30B-A3B-Instruct-2507, the base model of Appendix C: the HINT model of that appendix trains on Claude Code trajectories mixed with their CHR renderings, and the alternative trains on real trajectories collected in all four production harnesses with the same schedule and number of optimizer steps. Four of the five evaluation surfaces thus occur in the alternative's training data, whereas for HINT every surface but Claude Code is unseen. As Figure 4 shows, HINT is nonetheless ahead under Claude Code, by 16.4 and 14.7 points on Verified and Multilingual, and under OpenCode, by 3.8 and 10.3 points, although the other model trained on OpenCode trajectories, and within a few points of it under the other three surfaces. Averaged over the five harnesses it is 2.1 points higher on Verified and 4.7 on Multilingual, and its accuracy varies across harnesses by 16.9 and 19.3 points against the other model's 26.6 and 24.0. At an equal step budget, then, synthetic surfaces perform on a par with real trajectories distilled in the target harnesses. The two training sets also differ in their tasks, so the comparison is between two data pipelines rather than between surfaces alone.

4.5 TEST-TIME ANONYMIZATION OF THE INTERFACE

Whole harnesses differ from the training surfaces in many respects at once, so they cannot show which cue a model relies on. We isolate the cheapest cue, the identifiers of tools and parameters, which let a model that recognizes `Bash` act without reading its description. Holding the CHAMELEON configuration of Table 4 fixed (9 tools, 17 parameters), we replace about half of the names, then all of them, with meaningless identifiers (`run_command` becomes `zete`). Tool descriptions are identical across settings and the system prompt names no tool, so what each tool does stays readable. Qwen3.6-35B-A3B before fine-tuning and the HINT model of Section 4.3 are evaluated once per setting on SWE-bench Multilingual, each compared with its own run under the original names. As Figure 5 shows, HINT's accuracy is unchanged (74.3%, 74.6% and 74.3%), and with all names replaced it newly fails exactly as many tasks as it newly solves (22 each). The base model falls from 68.0% to 67.3% and 64.0%, losing 29 tasks and gaining 17, and its new failures coincide with the new identifiers: calls to tools that do not exist rise from 5 to 27 of the 300 tasks, misspelling the new identifiers (`kazu_op` for `kozu_op`) rather than reverting to the original ones. HINT's accuracy thus does not depend on recognizable identifiers, the behaviour the anonymized tier is meant to teach, although the 22 flips in each direction show that individual episodes still change. The test is in distribution, since training used the same identifier generator, so it shows that the behaviour was acquired, not that it extends to unseen naming schemes.

4.6 ONE SURFACE AXIS AT A TIME

Production harnesses differ on several axes at once, so Section 4.3 cannot say which axis a model is bound to; CHAMELEON can move them one at a time. From an origin configuration that mimics Claude Code, we move each of A1, A3, A4 and A5 in turn to a far setting (Appendix D lists them), and two further surfaces change A1 and A5 together and all four axes. A2 stays native function calling, and every surface exposes the same actions. We evaluate the HINT model of Appendix C and a control fine-tuned from Qwen3-30B-A3B-Instruct-2507 on the same Claude Code trajectories alone,

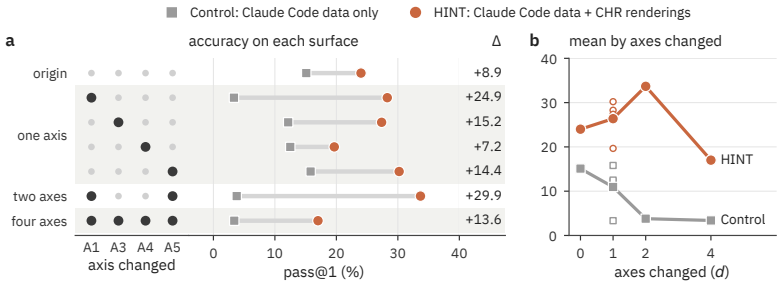


Figure 6: CHAMELEON surfaces that move surface axes, one or several at a time, away from a Claude Code-like origin (SWE-bench Multilingual, one evaluation per surface). **(a)** The axes each surface changes (A1 naming and granularity, A3 prompt and documentation, A4 observation format, A5 turn conventions) and the accuracy of HINT and of the control; Δ is HINT minus control. **(b)** Mean accuracy over the surfaces that change d axes (one surface each at $d = 0$, 2 and 4); open markers are the four single-axis surfaces.

once per surface on SWE-bench Multilingual, in an environment that differs from that appendix, so only differences between surfaces are interpreted. The control is bound to A1 (Figure 6a): changing naming and granularity alone takes it from 15.1% on the origin to 3.3%, and it stays there whenever A1 is changed, whereas changing A3, A4 or A5 alone costs it at most 2.9 points. HINT is not: with A1 changed alone it scores 28.3% against 24.0% on the origin, and it is ahead of the control on every surface, by 8.9 points on the origin and by up to 29.9 (Figure 6b); only the surface that changes all four axes brings it down, to 17.0%. A model trained on one harness has thus absorbed its tool vocabulary and granularity as part of the task; HINT has not, and loses only when every axis changes at once.

5 RELATED WORK

Yang et al. (2024) showed that the agent–computer interface materially determines performance, and recent measurements find that the choice of scaffold moves scores by as much as the choice of model (Zhang et al., 2026; Yao et al., 2026; Zheng et al., 2026). Data pipelines such as SWE-Gym (Pan et al., 2024), SWE-smith (Yang et al., 2025) and SWE-rebench (Badertdinov et al., 2026) scale tasks and trajectories within one scaffold, while a growing line of work trains across many, by mixing real trajectories from several scaffolds (Abadji et al., 2026) or randomizing the harness during reinforcement learning (Huang et al., 2026; Xu et al., 2026); to our knowledge, we are the first to synthesize surfaces by re-rendering one harness's real trajectories (Section 4.4 compares the two). Gu et al. (2026) show that trajectory SFT amplifies interface memorization over semantic understanding, which HINT counters by domain randomization (Tobin et al., 2017) over an explicit, invertible surface space rather than by a penalty as in invariant risk minimization (Arjovsky et al., 2019). The Agent Data Protocol (Song et al., 2025) also defines a scaffold-independent trajectory representation, compiled into fixed target formats; ours is recovered from production harnesses by normalization, factored into sampled axes, and proved invertible. Appendix F discusses further related work.

6 CONCLUSION

Agent harnesses differ far more on the surface than in substance: four production harnesses reduce to 12 semantic actions, and making that core explicit as the HARNESS PROTOCOL turns interface variation into a nuisance factor that a training pipeline can marginalize over, and HINT does so with no new task data and no architectural change: it matches a single-harness control on that control's harness, is ahead under every unseen production harness by up to 8.7 points, and is on a par with training on real trajectories from four harnesses. Aligning hidden representations across renderings is the natural next step. Appendix E discusses the limitations of this study.

AI USE STATEMENT

In this work, we used generative AI tools for editing and polishing the text of the manuscript, that is, for improving grammar, wording and readability of sentences that we wrote. We have not used generative AI tools for research ideation, for the design of the method or of the experiments, for writing the code of the renderer, the harness or the evaluation pipeline, or for producing or analysing the data and results reported here, and the remaining tasks with required disclosure are not applicable to this work. We have reviewed all AI-assisted text. We take responsibility for the final content of this work, including any text produced with the aid of generative AI.

ETHICS STATEMENT

This work trains and evaluates software-engineering agents on public GitHub repositories: the training tasks are SWE-rebench-V2 instances and the evaluation tasks are the public SWE-bench Verified and SWE-bench Multilingual sets. No human subjects, personal data or code beyond these public repositories are involved, and repository content is used under its own licenses. Trajectories are collected and evaluated in isolated sandboxes that cannot affect systems outside the task container. The method aims to make agents less sensitive to their interface, and we foresee no harm specific to it beyond those of coding agents in general.

REPRODUCIBILITY STATEMENT

Section 2 and Appendix A define the protocol, the renderer and the parser, state the premises (P1)–(P4) and prove Proposition 1; the round-trip check that verifies the implementation is described in Section 2 and Table 3, with the $\simeq$ cases enumerated in Appendix A.6. Section 3 specifies the training-surface space $\mathcal{H}_{\text{train}}$ and the three naming tiers, and Section 4.1 gives the task source, teacher, filters, sample counts, per-axis composition and training hyperparameters of both fine-tuning runs. Appendix D lists every experiment's models, harnesses, evaluation counts and conditions (Table 8), the rounding rule and the harness releases evaluated. We plan to release the parsers, the renderer, the CHAMELEON harness, the surface sampler and the evaluation configurations, together with the rendered training samples, as supplementary material.

REFERENCES

Julien Abadji, Marah Abdin, Connor Adams, Eric Alcaide, Mustafa Altun, Michele Artoni, et al. Laguna M.1/XS.2 technical report. *arXiv preprint arXiv:2605.27605*, 2026.

Carl Allen and Timothy Hospedales. Analogies explained: Towards understanding word embeddings. In *International Conference on Machine Learning (ICML)*, 2019.

Anthropic. Claude Code. `https://github.com/anthropics/claude-code`, 2025. Version 2.1.140.

Martin Arjovsky, Léon Bottou, Ishaan Gulrajani, and David Lopez-Paz. Invariant risk minimization. *arXiv preprint arXiv:1907.02893*, 2019.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*, 2025.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench V2: Language-Agnostic SWE task collection at scale. *arXiv preprint arXiv:2602.23866*, 2026.

Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. A theory of learning from different domains. *Machine Learning*, 79:151–175, 2010.

Bjarni Haukur Bjarnason, André Silva, and Martin Monperrus. On randomness in agentic evals. *arXiv preprint arXiv:2602.07150*, 2026.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, et al. Qwen3-Coder-Next technical report. *arXiv preprint arXiv:2603.00729*, 2026.

Aili Chen, Aonian Li, Baichuan Zhou, Bangwei Gong, Binyang Jiang, Boji Dan, et al. The MiniMax-M2 series: Mini activations unleashing max Real-World intelligence. *arXiv preprint arXiv:2605.26494*, 2026.

Weizheng Gu, Chengze Li, Zhuohao Yu, Mengyuan Sun, Zhibang Yang, Wei Wang, et al. What do agents learn from Trajectory-SFT: Semantics or interfaces? *arXiv preprint arXiv:2602.01611*, 2026.

Jie He, Jennifer Neville, Mengting Wan, Longqi Yang, Hui Liu, Xiaofeng Xu, et al. GenTool: Enhancing tool generalization in language models through Zero-to-One and Weak-to-Strong simulation. *arXiv preprint arXiv:2502.18990*, 2025.

Irina Higgins, David Amos, David Pfau, Sebastien Racaniere, Loic Matthey, Danilo Rezende, and Alexander Lerchner. Towards a definition of disentangled representations. *arXiv preprint arXiv:1812.02230*, 2018.

Bo Huang, Fengxiang Li, Hao Xu, Haoyang Huang, Hongyi Fu, Jinhua Hao, et al. KAT-Coder-V2.5 technical report. *arXiv preprint arXiv:2607.05471*, 2026.

Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *arXiv preprint arXiv:2504.07164*, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Kimi Team et al. Kimi K2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

Qiqiang Lin, Muning Wen, Qiuying Peng, Guanyu Nie, Junwei Liao, Jun Wang, et al. Hammer: Robust Function-Calling for On-Device language models via function masking. *arXiv preprint arXiv:2410.04587*, 2024.

Xiaofeng Lin, Hejian Sang, Zhipeng Wang, and Xuezhou Zhang. Debunk the myth of SFT generalization. *arXiv preprint arXiv:2510.00237*, 2025.

Victor May, Aaditya Salgarkar, Yishan Wang, Diganta Misra, and Huu Nguyen. Agents learn their runtime: Interpreter persistence as Training-Time semantics. *arXiv preprint arXiv:2603.01209*, 2026.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

OpenAI. Codex CLI. `https://github.com/openai/codex`, 2025. Version 0.142.0.

OpenCode contributors. OpenCode. `https://github.com/anomalyco/opencode`, 2025. Version 1.17.15.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. *arXiv preprint arXiv:2412.21139*, 2024.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

Qwen Team. Qwen3-30B-A3B-Instruct-2507 model card. `https://huggingface.co/Qwen/Qwen3-30B-A3B-Instruct-2507`, 2025b.

Qwen Team. Qwen3.5-35B-A3B model card. https://huggingface.co/Qwen/Qwen3.5-35B-A3B, 2026a.

Qwen Team. Qwen3.6-35B-A3B model card. https://huggingface.co/Qwen/Qwen3.6-35B-A3B, 2026b.

Abhinav Rastogi, Adam Yang, Albert Q. Jiang, Alexander H. Liu, Alexandre Sablayrolles, Amélie Héliou, et al. Devstral: Fine-tuning language models for coding agent applications. *arXiv preprint arXiv:2509.25193*, 2025.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. *arXiv preprint arXiv:2310.11324*, 2023.

Yueqi Song, Ketan Ramaneti, Zaid Sheikh, Ziru Chen, Boyu Gou, Tianbao Xie, et al. Agent data protocol: Unifying datasets for diverse, effective fine-tuning of LLM agents. *arXiv preprint arXiv:2510.24702*, 2025.

SWE-agent Team. mini-SWE-agent. https://github.com/SWE-agent/mini-swe-agent, 2025.

Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017.

Julius von Kügelgen, Yash Sharma, Luigi Gresele, Wieland Brendel, Bernhard Schölkopf, Michel Besserve, and Francesco Locatello. Self-supervised learning with data augmentations provably isolates content from style. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, et al. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025a.

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang Tran, Engel Nyst, et al. The OpenHands software agent SDK: A composable and extensible foundation for production agents. *arXiv preprint arXiv:2511.03690*, 2025b.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Binfeng Xu, Hao Zhang, Shaokun Zhang, Songyang Han, Mingjie Liu, Jian Hu, et al. Polar: Agentic RL on any harness at scale. *arXiv preprint arXiv:2605.24220*, 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.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025.

Yilun Yao, Xinyu Tan, Chao-Hsuan Liu, Yaoming Li, Zhengyang Wang, Wenhan Yu, et al. HarnessBench: Measuring harness effects across models in realistic agent workflows. *arXiv preprint arXiv:2605.27922*, 2026.

Xiao Yu, Baolin Peng, Ruize Xu, Hao Zou, Qianhui Wu, Hao Cheng, et al. OpenForgeRL: Train harness-native agents in any environment. *arXiv preprint arXiv:2607.21557*, 2026.

Z.ai. GLM-5.3 model card. https://huggingface.co/zai-org/GLM-5.3, 2026.

Liang Zeng, Yongcong Li, Yuzhen Xiao, Changshi Li, Chris Yuhao Liu, Rui Yan, et al. Skywork-SWE: Unveiling data scaling laws for software engineering in LLMs. *arXiv preprint arXiv:2506.19290*, 2025.

Yunbei Zhang, Janet Wang, Yingqiang Ge, Weijie Xu, Jihun Hamm, and Chandan K. Reddy. Stop comparing LLM agents without disclosing the harness. *arXiv preprint arXiv:2605.23950*, 2026.

Mengyu Zheng, Kai Han, Boxun Li, Haiyang Xu, Yuchuan Tian, Wei He, et al. Claw-SWE-Bench: A benchmark for evaluating OpenClaw-style agent harnesses on coding tasks. *arXiv preprint arXiv:2606.12344*, 2026.

A Proof of Proposition 1

Proposition 1 states that the surface map of every harness our renderer realizes is exactly invertible on the action channel. The proof is by construction: the renderer is a composition of two stages, and each stage is shown to have a left inverse. Section A.1 fixes notation, Sections A.2–A.4 prove one lemma per stage, Section A.5 assembles them, Section A.6 lists the premises and states precisely what the measurement of Table 3 adds to the proof, and Section A.7 illustrates the construction on one trajectory.

A.1 Setting

Actions. An action is a pair $a = (v, r)$ of a *verb* $v \in \mathcal{V}$ and a *record* r. The verb set $\mathcal{V}$ has the twelve elements of Table 1 (the four `edit_file` sub-actions count as distinct verbs). Every verb has a finite set of parameter keys K_v, each with a declared type (string, integer, boolean, or JSON value); a record is a partial map $r : K_v \rightharpoonup \mathbb{V}$ from keys to values of the declared type. Unset keys are simply absent; two records are equal when they are equal as partial maps. $\mathcal{A}$ is the set of all such pairs.

Configurations. Of the five axes of Eq. (1), only A_1 and A_2 touch the text of a call. A configuration h contributes

- a granularity $g \in \{\text{SPLIT}, \text{UNIFIED}, \text{MONO}\}$, which fixes the set T_g of surface tools: twelve tools, one per verb; eight tools, one *editor* tool shared by the five file verbs; or a single tool;
- a tool-name map $N : T_g \to \Sigma^*$ and a parameter-name map $\Pi : K \to \Sigma^*$, where K is the union of all K_v together with the *selector* keys of the editor and the single tool;
- a syntax family $s \in \{\text{JSON}, \text{XML}, \text{YAML}, \text{SHELL}\}$.

The remaining components (prompt template, documentation style, observation decoration, turn conventions) are constant with respect to the action text and play no role below.

Factorization. On one action the renderer is $R_h = E_s \circ N_h$, where the naming stage N_h maps an action to a *named record* (n, ρ) (a surface tool name and a record over surface parameter names) and the syntax stage E_s serializes the named record to text. The parser is $P_h = N_h^{-1} \circ D_s$ with D_s the block decoder of the family. The shell family has no named tools and defines E_{SHELL} directly on actions (Section A.4). All four families share an *escape encoding* $E_{\text{esc}}(a)$: a fenced block tagged `call` whose single line is the JSON serialization of $\{\texttt{action} : v, \texttt{args} : r\}$ with the canonical verb and keys. Since JSON escapes newlines, the payload occupies one line and the closing fence is unambiguous, so E_{esc} is decoded exactly by reading the fence and applying the JSON parser. Every use of E_{esc} below is therefore exact and bypasses the naming stage.

Premises. The lemmas use four premises, each discharged in Section A.6:

(P1) N is injective on T_g;

(P2) Π is injective on the key set of each surface tool;

(P3) (JSON only) no argument value contains a well-formed call of the same configuration;

(P4) values carry their declared types, and a string-valued JSON parameter is not itself a JSON document.

A.2 LEMMA 1: THE NAMING STAGE IS AN INJECTIVE TABLE LOOKUP

Lemma 1. *Under (P1), (P2) and (P4), N_h is injective and the inverse tables of the parser recover (v, r) from (n, ρ).*

Proof. Let $\tau : \mathcal{V} \to T_g$ assign each verb its surface tool. Under SPLIT, τ is a bijection and $N_h(v, r) = (N(\tau(v)),\ r \circ \Pi^{-1})$. Under UNIFIED, the five file verbs share the editor tool, and N_h adds to the renamed record one selector entry $\Pi(\mathrm{sel}) \mapsto \ell(v)$ with ℓ an injection of the five verbs into the literals $\{\texttt{view}, \texttt{str_replace}, \texttt{insert}, \texttt{create}, \texttt{undo_edit}\}$; all other verbs are treated as under SPLIT. Under MONO, every verb uses the single tool and the selector carries one of twelve distinct literals. In each case the decoder proceeds in three steps. (i) From n it recovers the surface tool, because N is injective (P1). (ii) If the tool is the editor or the single tool, it reads the selector entry, which is present because the encoder always writes it and unambiguous because the selector key is distinct from every data key of that tool (P2); the literal determines v because ℓ is injective. Otherwise the tool determines v because τ is a bijection on the remaining verbs. (iii) It renames the data keys back through Π^{-1}, which is well defined on the tool's key set by (P2), and re-types the values by the declared type of each key, which returns the original values by (P4). Hence the decoder is a left inverse of N_h and N_h is injective. □

A.3 LEMMA 2: THE NAMED-TOOL SYNTAX FAMILIES ARE DECODABLE ENCODINGS

Lemma 2. *For $s \in \{\text{JSON}, \text{XML}, \text{YAML}\}$ and every named record (n, ρ), $D_s(E_s(n, \rho)) = (n, \rho)$; for* JSON *this uses (P3).*

Proof. Write J for JSON serialization. J is injective on finite records with distinct string keys and values of the types above (every string becomes a quoted string in which quotes, backslashes and control characters are escaped), and $J^{-1} \circ J$ is the identity on them.

JSON: $E(n, \rho)$ is $J(\{\texttt{name} : n, \texttt{arguments} : \rho\})$ between an opening and a closing tag. D scans for the opening tag, takes the first brace after it and the shortest prefix that is a balanced object under a string-aware bracket count, and applies J^{-1}. The output of J is itself such a balanced object, and braces inside its string literals are skipped by the string-aware count, so the recovered prefix is exactly J's output. A tag occurring inside a value can only start a second candidate block; by (P3) that block is not a well-formed call of the configuration and is rejected, so the decoder returns the single record. In the structured tool-calling channel used for training, the object is carried by the API rather than located by tag scanning, and (P3) is not needed.

XML: if any value contains a closing marker of the family (`</arg>`, `</invoke>`, or the opening of an invoke element), E routes the action to E_{esc}, which is exact. Otherwise E writes an invoke element carrying n and, per key, an `arg` element carrying the value: strings verbatim, other types as $J(\cdot)$. D takes the shortest span from the opening element to `</invoke>` and, inside it, the shortest span from each `arg` opening to the next `</arg>`. Because no value contains these markers, both spans are the intended ones; newlines are matched, so strings come back verbatim, and non-string values are recovered by J^{-1} followed by re-typing (P4).

YAML: E opens a fence tagged `action`, writes `tool:` n, then one line per key. A value that is not a multi-line string is written on the key's own line as $J(\cdot)$; a multi-line string is written as a block scalar (`|-`) whose lines follow, each prefixed by exactly four spaces, an empty line becoming four spaces. The fence closes at column 0. No value can produce a line starting at column 0: single-line values are on the key line with newlines escaped by J, and block lines are indented. Hence the first column-0 fence after the opening is the true end of the block. D reads key lines at two spaces of indentation, collects the four-space lines of a block scalar while stripping the prefix, and drops trailing lines that are genuinely empty; the only such line is the one produced by the newline before the closing fence, since empty value lines were encoded as four spaces. Scalars on key lines are recovered by J^{-1}. □

A.4 LEMMA 3: THE SHELL-ONLY FAMILY IS EXACT UP TO $\simeq$

The SHELL family has no named tools, so the naming stage is empty and E_{SHELL} is defined verb by verb. Some verbs have no surface form of their own in this family and are *lowered* to the shell

command that performs them; the decoder then returns that command. This is the one place where the round trip is not literally the identity, and we make the slack explicit.

Operational equivalence. Write $a \simeq a'$ when the two actions induce the same environment transition and the same observation content, with decoration removed, from every workspace state. The relation identifies: a file read with the shell command that performs it, `read_file`$(p, s, e) \simeq$ `run_shell(sed -n 's,ep'` p) and `read_file`$(p) \simeq$ `run_shell(cat` p); a search with the corresponding `grep` or `find` command; a fetch with `curl`; two shell commands that differ only in leading or trailing newlines; a submission whose message is empty with the documented filler message; and an action with an argument the executor ignores (the display label of a delegation, the `view`/`plan` selector of a task-list update) with the same action without it.

Lemma 3. *For every action a, $P_{\text{SHELL}}(E_{\text{SHELL}}(a)) \simeq a$. Consequently $E_{\text{SHELL}}(a) = E_{\text{SHELL}}(a')$ implies $a \simeq a'$.*

Proof. We list the encoding and its decoding per verb. `run_shell`: the command in a fenced block tagged `bash`; a command containing a fence goes to E_{esc}. The decoder takes the shortest span to the closing fence and strips outer newlines ($\simeq$). `edit_file`: a patch envelope. The header line carries the path, so a path that is empty, contains a newline, has outer whitespace, or contains the envelope terminator goes to E_{esc}; the body carries the old and new text line by line, each line prefixed by − or +, so any content is expressible except one containing the terminator itself, which goes to E_{esc}. Insertion adds `@@ after line` k to the header, creation uses an `Add File` header with + lines, undo a `Revert Last Edit` header, and a replace-all edit, which no patch expresses, goes to E_{esc}. The decoder inverts each form line by line and is exact. `delegate`: an envelope whose header carries the agent kind and whose body carries the prompt verbatim (the terminator inside the prompt goes to E_{esc}); the display label is dropped ($\simeq$). `task_track`: a `PLAN UPDATE` block whose body, the list serialized by J when structured, runs to the next block; a body containing a block opener goes to E_{esc}; the selector is dropped ($\simeq$). `web_search`, and `web_fetch` with an extraction prompt: E_{esc}, exact. `read_file`, `search` and `web_fetch` without a prompt: lowered to `sed`/`cat`, `grep`/`find` and `curl` commands inside a `bash` fence, decoded as `run_shell` ($\simeq$). `submit`: the message as plain text; a message that is empty or whitespace becomes the filler ($\simeq$), and one containing a block opener goes to E_{esc}. The decoder collects fences, escape blocks, patch and delegation envelopes and the plan block, keeps the earliest non-overlapping ones so that a fence quoted inside a patch body is not a second call, and reads block-free text as a submission. In every case the decoded action is the original or its $\simeq$-image, which proves the first claim; the second follows because $\simeq$-images are unique up to $\simeq$. □

A.5 Proof of the proposition

Proof of Proposition 1. For a configuration with a named-tool family, Lemmas 1 and 2 give $P_h \circ R_h = N_h^{-1} \circ D_s \circ E_s \circ N_h = N_h^{-1} \circ N_h = \mathrm{id}$ on $\mathcal{A}$. For the shell family, Lemma 3 gives $P_h \circ R_h \simeq \mathrm{id}$. A map with a left inverse is injective, so R_h is injective on actions, modulo $\simeq$ in the shell case. The trajectory-level statement follows step by step: an assistant turn is the reasoning text followed by the call blocks of that step, separated by blank lines, and the decoder returns the text outside the blocks as the reasoning. This is exact as long as the reasoning contains no block opener of the family, which the normalization of Section 4.1 guarantees by moving every call out of the text channels. Two turn conventions absorb the final reasoning into the submission message (the shell family, and the plain-reply completion convention of A_5); we count them under $\simeq$. □

Two remarks. First, the composition $T_{h \to h'} = R_{h'} \circ P_h$ is defined on the image of R_h, so it is a total map between surfaces, and $P_{h'} \circ T_{h \to h'} = P_h$ on that image when h' has named tools, and $\simeq P_h$ when h' is the shell family: transport preserves the action sequence, exactly or up to $\simeq$. Second, the observation channel is decorated rather than encoded: A_4 adds line-number gutters on file views, wrapper tags, and empty-output and error markers, all of which are stripped by the inverse decoration, and may truncate long outputs to a character budget when a configuration sets one. The training sampler never sets one, so training renderings keep observation content whole; truncation occurs only in the evaluation surface of Section 4.6 that imitates production behaviour. Proposition 1 is in any case a statement about the action channel.

A.6 PREMISES, AND WHAT THE MEASUREMENT ADDS

(P1) and (P2) hold for every configuration the sampler produces. Tool names are drawn from synonym pools with rejection of a name already used in the configuration, and anonymized identifiers with the same rejection. For parameters, the synonym pools of any two keys that share a surface tool are pairwise disjoint, a static property of the name tables that we check mechanically, and anonymized parameter names are drawn with rejection. (P3) and (P4) are properties of the data: (P3) fails only if an argument value, such as the content of a file being written, itself contains a well-formed call in the configuration's own vocabulary; (P4) holds because the normalizer types every argument when it maps a harness call to a protocol action.

The proof covers the design of the maps; the measurement of Table 3 covers the implementation and the data. Rendering all 578,856 real actions under three or four randomly sampled configurations each, 1.87M render–parse pairs, and parsing them back checks (a) that the code realizes the encodings described above and (b) that the corpus satisfies (P3) and (P4). Both hold: zero mismatches, with the $\simeq$ cases of the shell family (89,930 checks, 4.8% of the total) counted as consistent but never as exact. A mechanical companion check exercises the encodings on adversarial values rather than real ones: 600 sampled configurations across all families, each rendering 72 actions whose arguments contain every family's delimiters, empty strings, outer whitespace and newlines, Unicode and $10{,}000$-character payloads, for $43{,}200$ render-parse pairs with zero mismatches. Four of the special cases of Lemma 3 (a path with outer whitespace or containing the terminator, a task list containing a block opener and a replace-all edit, which go to E_{esc}, and the whitespace-only submission, which becomes the filler) were added as a result of this check; none of them occurs in the corpora of Table 3.

A.7 ONE TRAJECTORY, THREE SURFACES

Figure 7 makes the construction concrete on a single episode. The source harness is parsed (P_A) into protocol form $\tau = [(c_t, a_t, o_t)]$; the same τ is then rendered under two other configurations, one that draws synthetic tool names (R_B) and one that replaces every identifier with an anonymized token (R_C). Neither counterfactual surface ever existed as a harness, yet each carries the same reasoning, the same actions and the same observations as the source, differing only in the text that wraps them. Proposition 1 is the statement that all three surfaces parse back to the same τ: the three panels are three renderings of one trajectory, and the training set of Section 3 is built from such renderings.

B BEHAVIOURAL ANALYSIS OF THE EVALUATION TRAJECTORIES

Accuracy summarizes an episode in one bit. The trajectories behind this paper's cross-harness evaluations carry more: every tool call, the observation it produced, the tokens each step consumed, and how the episode ended. This appendix reads them. Two collections are used. The first is a models $\times$ harnesses evaluation: six instruction-tuned models (Claude Opus 5, GPT-5.6, DeepSeek-V4-Pro, GLM-5.2, Qwen3.6-35B-A3B, Qwen3.5-27B) run under the four production harnesses and CHAMELEON (one fixed synthetic configuration) on the 500 Verified and 300 Multilingual tasks (Qwen3.5-27B on a $100 + 60$ subset), $17{,}374$ completed episodes over 26 model–harness cells, of which $17{,}230$ made at least one tool call and enter Table 5; Figure 8 marks the cells that could not be run. The second is the HINT and control pair of the earlier Qwen3-30B-A3B-Instruct-2507 run (Appendix C), on Claude Code (the source harness), OpenCode and OpenHands SDK: $4{,}280$ episodes. Every call is mapped to the protocol vocabulary with the parsers of Section 2, and shell commands are further labelled by intent (read, search, edit, run tests, other), so that a `sed -n` issued in Codex CLI and a `Read` issued in Claude Code count as the same thing. Episodes that ended in an infrastructure exception are excluded from behavioural statistics, and one cell, Claude Opus 5 $\times$ Codex CLI, is dropped entirely: 57% of its episodes were cut short by an API-gateway error after two or three calls. The frontier models were reached through a gateway, so this appendix draws no conclusion from the absolute accuracy of any single API cell; the behavioural quantities below are robust to that.

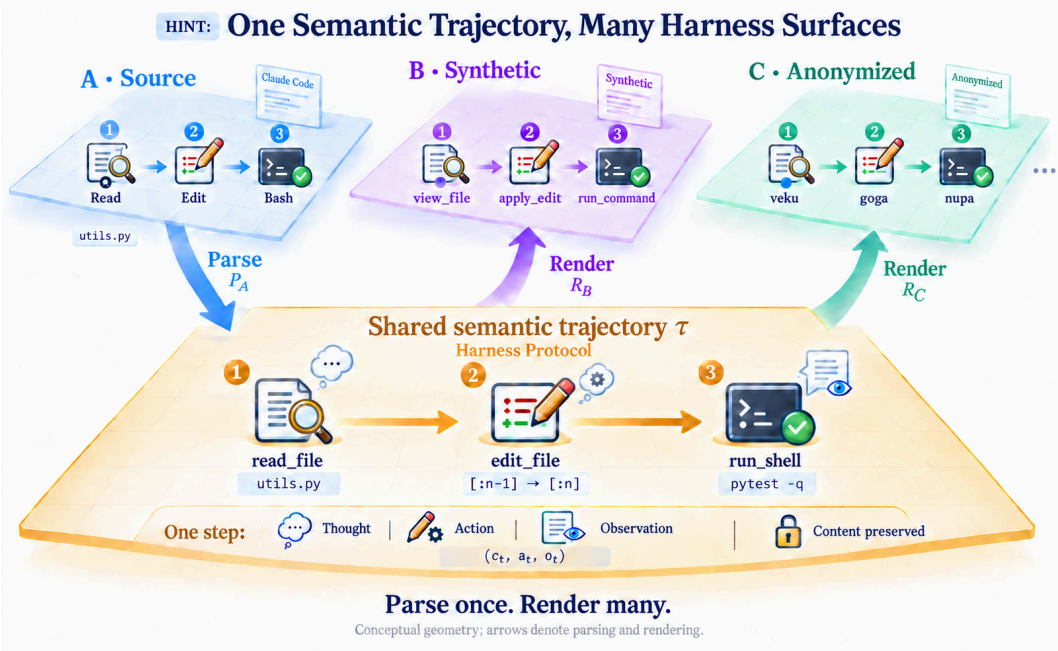


Figure 7: One protocol trajectory and three of its surfaces. Parsing the source trajectory (P_A) yields $\tau = [(c_t, a_t, o_t)]$; rendering it under two other configurations, one with synthetic tool names (R_B) and one with anonymized identifiers (R_C), produces counterfactual harnesses that never existed but carry the same episode. Proposition 1 states that every such surface parses back to the same τ: the content of the episode is preserved and only the surface changes.

B.1 Semantic action composition

Figure 8 gives, for each model and harness, the mix of semantic actions after normalization. The rows within a panel are strikingly alike. Moving a model between harnesses shifts its action mix by a median Jensen–Shannon divergence of 0.025 bits; moving between models under a fixed harness shifts it by 0.027 bits. The harness, in other words, changes what a model does about as little as the choice of model does, and both changes are small. What does move is exactly what the harness makes explicit: planning calls exist only where a harness declares a task tracker (`TodoWrite` in Claude Code, `todowrite` in OpenCode, `task_tracker` in OpenHands SDK), and a terminating call only where the harness requires one (OpenHands SDK, CHAMELEON). Codex CLI is the one surface that visibly reshapes behaviour: with no first-class search tool, GPT-5.6 spends 38% of its calls searching there against 13–25% elsewhere, the exploration that a single `Grep` or `Glob` delivers in other harnesses being spread over several shell invocations.

Figure 9 resolves the same calls over the course of an episode. Every harness shows one arc: reads and searches make up 70–90% of the first calls, editing grows through the middle, and testing rises toward the end. Conventions occupy the two ends. Claude Code and OpenCode open with planning and delegation calls (21% and 18% of the calls in the opening 5% of an episode), OpenHands SDK and CHAMELEON close with an explicit finish (26% and 30% of the calls in the closing 5%), and Codex CLI opens with a search-dominated burst (67% of its opening calls). Below the conventions, the coarse action composition is the same.

B.2 Episode length and context cost

If the harness leaves the semantic policy alone, what does it change? Figure 10 and Table 5 answer: how much the same work costs. Pooled over models, the median episode takes 22 calls in Claude Code and 46 in CHAMELEON, and the median prompt tokens per call range from 11k in CHAMELEON to 38k in Claude Code, a 3.4× difference for the same semantic actions. The per-call cost is set by the surface: the system prompt and tool documentation a harness prepends, the

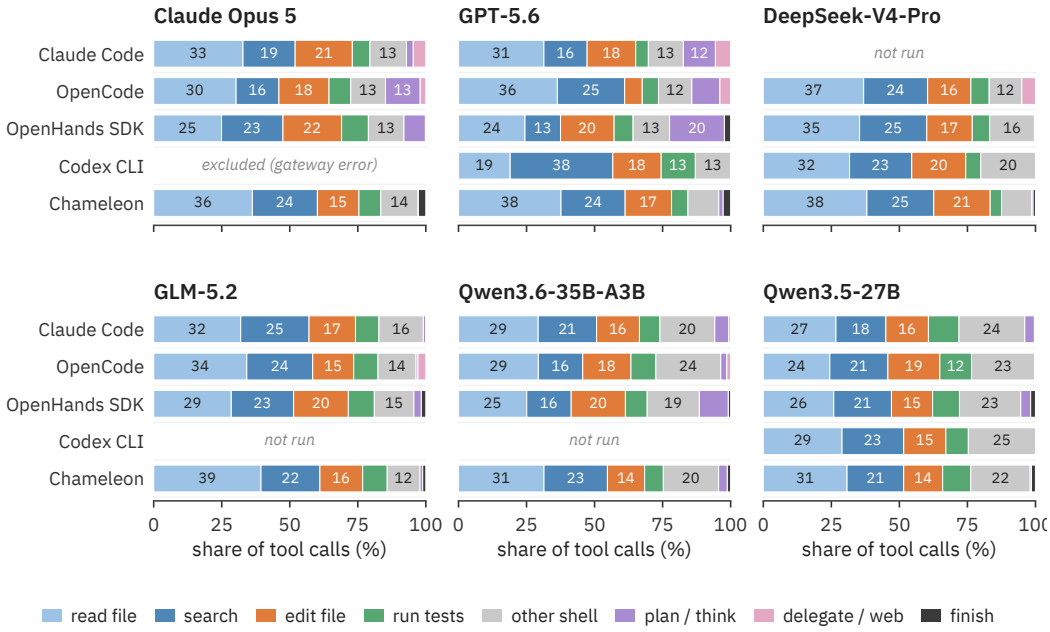


Figure 8: The same model does the same semantic work in every harness. Each row is one model–harness cell; segments are the share of the cell's tool calls that read a file, search, edit, run tests, run other shell commands, plan or think, delegate or fetch the web, or finish, after normalization to the protocol vocabulary and intent-labelling of shell commands. Numbers are percentages; rows within a panel differ from one another about as much as the same row differs across panels.

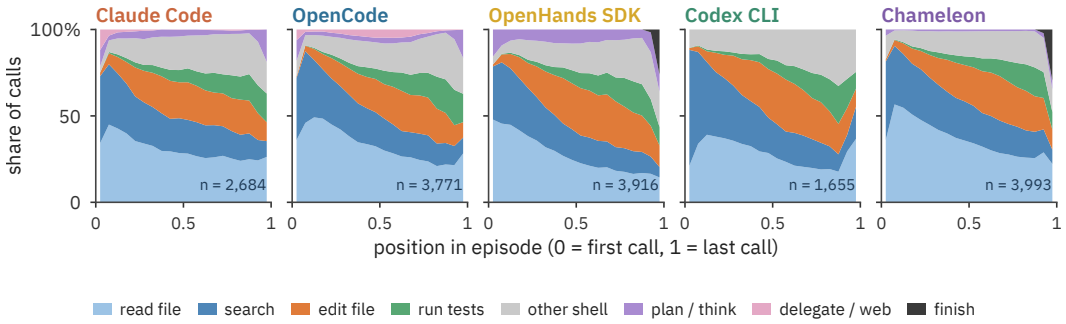


Figure 9: Where in an episode each kind of call happens, pooled over the six models (episodes with at least ten calls, n per panel). Every harness shows the same arc from exploration to editing to testing; harness conventions sit at the two ends, planning and delegation at the start where a harness offers them, an explicit finish at the end where a harness requires one.

verbosity of its observations, and whether it compacts context. Claude Code's 21-tool schema and Codex CLI's shell-only surface, whose observations are raw command output, are the most expensive per call; CHAMELEON's terse documentation and 2,000-character observation cap are the cheapest. The two quantities can move in opposite directions: CHAMELEON takes the most calls but the fewest tokens per call, and its total episode cost is the lowest of all harnesses for four of the six models.

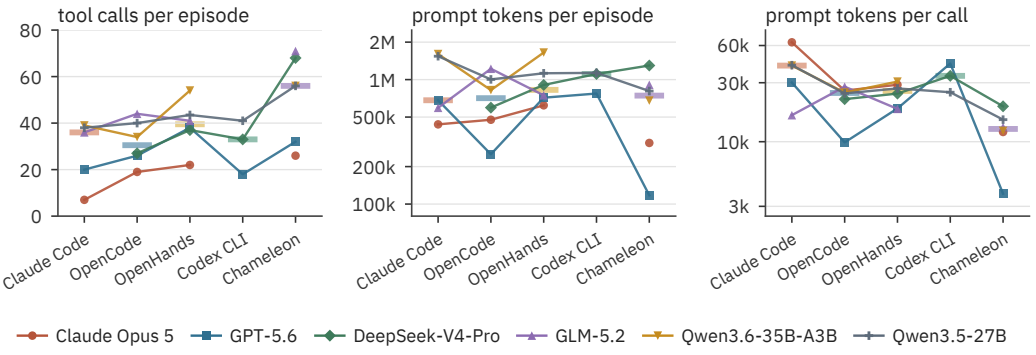


Figure 10: Episode length and context cost by harness. Per model (lines) and pooled over models (short bars): the median number of tool calls per episode, the median prompt tokens summed over an episode, and the median prompt tokens per call, a proxy for the context the model reads at each step. The same semantic work costs up to 3.4× more tokens per call depending only on the surface.

Table 5: Per-harness summary, pooled over the six models (episodes with at least one call; Codex CLI covers the three models that could be run under it). Tokens per call is the median over episodes of prompt tokens divided by calls. "Rejected" counts calls the harness refused before executing them, per 100 calls; "edit miss" is the share of edit calls whose anchor text was not found in the file.

Harness	Episodes	Calls	Prompt tok.	Tok./call	Reads	Rejected	Edit miss
Claude Code	3,304	22	741k	38.4k	31%	0.27	5.0%
OpenCode	4,084	29	569k	22.0k	33%	0.10	1.9%
OpenHands SDK	4,064	37	864k	24.4k	28%	0.99	0.6%
Codex CLI	1,747	23	925k	36.9k	28%	0.11	0.4%
CHAMELEON	4,031	46	487k	11.2k	36%	0.41	1.6%

B.3 TASK-LEVEL AGREEMENT ACROSS HARNESSES

A headline table reports one accuracy per harness; Figure 11 asks whether the harnesses solve the *same* tasks. They do not. Between 23% (DeepSeek-V4-Pro) and 51% (Qwen3.5-27B) of a model's tasks are harness-dependent, solved under some harnesses and not others (Figure 11a), and the agreement between any two harnesses on which tasks a model solves is only moderate, Cohen's κ between 0.25 and 0.54 (Figure 11b); Codex CLI agrees least with the rest. Consequently an oracle that may pick the harness per task beats the best single harness by 2.1 (GPT-5.6) to 11.9 (Qwen3.6-35B-A3B) points (Figure 11c), and the gap widens as the model weakens. Part of this gap is the run-to-run variance a single harness would also show ($\pm$2.0 points, Appendix D); the remainder is what the interface, not the task, decided. Two readings follow. The spread of a headline table understates interface dependence, since accuracies can agree while the solved sets do not; and the oracle gap bounds what a policy that is indifferent to the interface could recover.

B.4 INTERFACE ERRORS

Harnesses also differ in how often they refuse what the model says. Figure 12 counts two classes of harness-reported error. Calls rejected before execution (Figure 12a) are rare everywhere except OpenHands SDK, where every model has its highest rate (0.7–1.7 per 100 calls). Of those rejections, 1,831 across the six models carry the same message, "Cannot execute multiple commands at once", a convention no other harness has and every model violates. Edit calls whose anchor text is not found (Figure 12b) are highest under Claude Code for every model (1.5–7.1% of edits) and near zero under OpenHands SDK for the same models ($\leq$ 1.5%): Claude Code's `Edit` enforces a stricter contract than the others: a unique anchor (504 refusals for a missing and 82 for an ambiguous one, over all models), a change that is not empty (257), and, counted in panel (a), a read of the file before it is edited (166). Both panels read down the columns rather than across the rows. Friction follows the

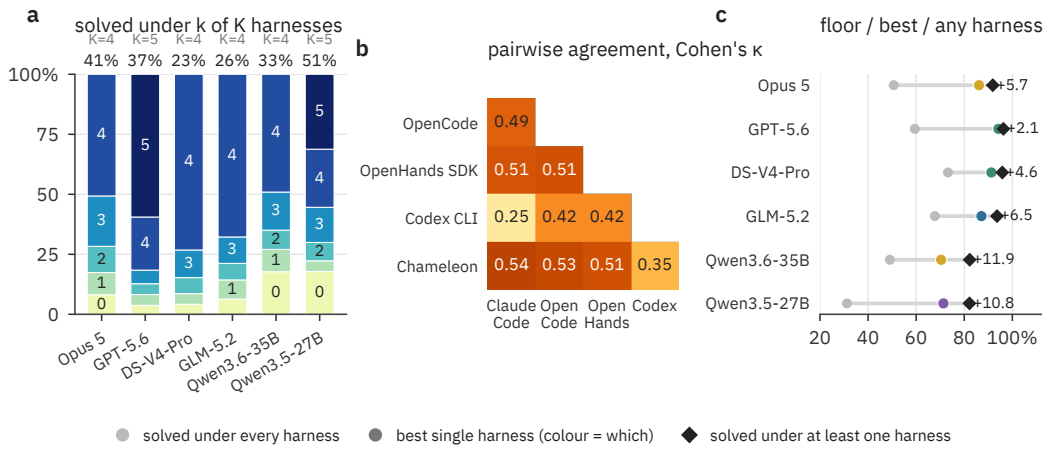


Figure 11: Task-level agreement across harnesses, per model, over tasks with a settled outcome under every harness the model was run in (K harnesses). (a) Share of tasks solved under exactly k of the K harnesses; the number above each bar is the harness-dependent share, $0 < k < K$. (b) Pairwise Cohen's κ between the sets of tasks two harnesses solve, averaged over the models that were run under both. (c) Per model: tasks solved under every harness, under the best single harness (coloured by which), and under at least one; the label is the gap between the last two, in points.

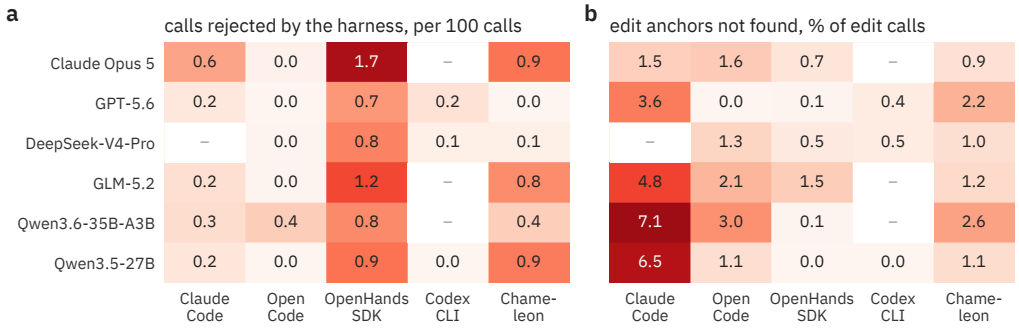


Figure 12: Harness-reported interface errors by model and harness. (a) Calls the harness refused before executing them: a tool that does not exist, malformed arguments, or a violated convention, per 100 calls. (b) Edit calls whose anchor text was not found in the file, as a share of edit calls. Both read down the columns: which harness a model is in predicts its friction better than which model it is.

interface more than the model, and it is exactly the kind of property a surface-bound policy learns to avoid in one harness and then runs into in the next.

B.5 HINT AND THE CONTROL OFF THE SOURCE HARNESS

Appendix C compares that run's two models by accuracy. Their trajectories show what the accuracy is made of. On the source harness the two arms are alike: 1.1 and 2.0 rejected calls per 100 (Figure 13a), the same action mix, the same edit precision (24% and 22% of edits miss their anchor). Off the source harness they diverge. On OpenCode the control's rejection rate rises to 5.1 per 100 calls, 1.6 of them calls to tools that do not exist, with names that belong to no harness (`bshell`, `browsetree`, `bqrtask`); HINT's stays at 2.3 with 0.1 non-existent tools. On OpenHands SDK the rates are 2.9 against 0.8. The control also reads paths that do not exist six times as often on OpenCode as HINT does (7.2 against 1.2 per 100 calls; 0.14 and 0.22 on the source harness), and

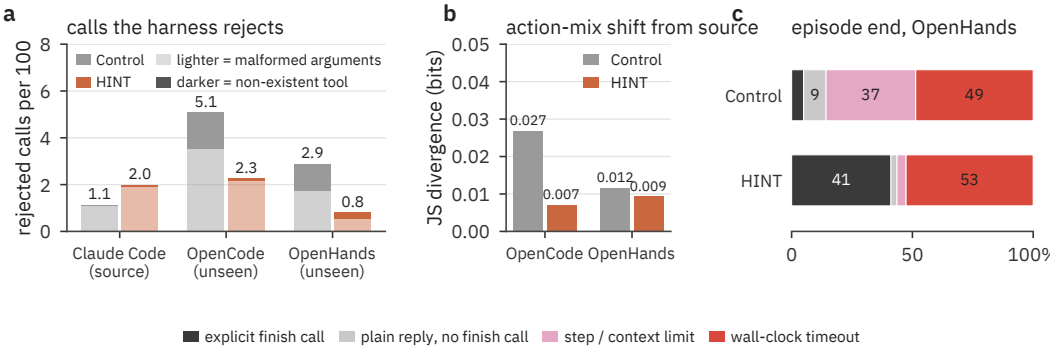


Figure 13: Interface behaviour of HINT and the step-matched control on the source harness and on two unseen ones (trajectories of the earlier run, Appendix C). (a) Calls the harness refused, per 100 calls, split into malformed arguments and calls to a tool that does not exist. (b) Jensen–Shannon divergence between a model's semantic action mix on an unseen harness and its own mix on the source harness. (c) How episodes end on OpenHands SDK, whose convention is an explicit finish call; both arms hit the wall clock often, which is why no OpenHands SDK accuracy enters a conclusion (Appendix C), but among the episodes that do not, HINT follows the convention and the control does not.

its edits miss their anchor more often (32% against 19%). Its semantic action mix moves away from its own source-harness behaviour by 0.027 bits on OpenCode where HINT's moves by 0.007 (Figure 13b). The clearest signal is the completion convention (Figure 13c). OpenHands SDK expects an explicit finish call; Claude Code, the source harness, expects a plain reply. Among episodes that did not hit the wall clock, HINT ends with the finish call 87% of the time and the control 10%, the control instead stopping with a plain reply or running into the step limit, that is, carrying the source harness's convention into a harness that does not have it. Even on the source harness, the control runs into the four-hour wall clock in 53% of episodes against 36% for HINT. These are the behaviours the accuracy in Table 6 averages over, and they are what interface invariance means at the level of a single call.

Figure 14 shows three of these failures verbatim. The first two are the control's: it calls OpenHands SDK's file editor with the argument schema of Claude Code's `Bash` tool, and it calls tools that exist nowhere. The third is not from a fine-tuned model but from an instruction-tuned model in the models $\times$ harnesses evaluation, and it is the purest form of interface reliance we observed: after one correct structured call, the model emits, as text, a call template it learned somewhere else, with a tool name from a different release of the same harness. The harness in front of it has told it, in the system prompt, what its tools are called and how to call them; the memorized surface wins anyway. 47 of the model's 81 failed Codex CLI episodes end on such a line, after a median of one call. This is the failure that HINT is designed to remove, and the reason the anonymized tier of Section 3 exists: a model that has learned to read the tool documentation in front of it has nothing to fall back on.

C AN EARLIER RUN ON QWEN3-30B-A3B-INSTRUCT-2507

Before the experiment of Section 4.3 we ran the same comparison at a smaller scale on Qwen3-30B-A3B-Instruct-2507 (Qwen Team, 2025a;b), with a different recipe on both sides. The data were 5,000 Claude Code trajectories; CHR rendered the 4,133 of them that passed the filters of Section 4.1 into $M = 2$ surfaces each, giving 8,266 samples under 8,204 distinct surface configurations with 11,205 distinct tool names. HINT trained on all 5,000 real trajectories *mixed with* the 8,266 renderings (13,266 samples) and the control on the real trajectories alone, each for the same number of optimizer steps at a global batch of 32 and learning rate 5×10^{-5}; for the control this is about 6.2 passes over its data. Matching sample counts instead would have given the control 27% more tokens, since the rendered samples are 34% shorter than the originals. Each cell was evaluated once, through a different serving endpoint from Section 4.1.

```
(1) Control on OpenHands SDK: the file editor called with the
    argument schema of Claude Code's Bash tool
    call     str_replace_editor
             {"command": "grep -n \"def write\\|class Basic\"
                          /testbed/astropy/io/ascii/basic.py | head -30",
              "description": ""}
    harness  ERROR: flag provided but not defined: -description

(2) Control on OpenCode: a tool that no harness declares
    call     bshell {...}
    harness  Model tried to call unavailable tool 'bshell'.
             Available tools: bash, edit, glob, grep, invalid, read,
             skill, task, todowrite, webfetch, write.

(3) Qwen3.5-27B on Codex CLI: a memorized call template emitted as
    text on the second turn, after one structured call to `shell`
    model    user
             <tool_call>
             <function=exec_command>
             <parameter=cmd>
             find /testbed -type f -name "python.py" | grep domain
             </parameter>
             </function>
             </tool_call>
    harness  (no call parsed; the text is taken as the final answer
             and the episode ends)
```

Figure 14: Three interface failures, verbatim. (1) The control applies the argument schema of its source harness's shell tool (`command` plus `description`) to a different harness's file editor (`str_replace_editor`, the editor of the OpenHands SDK build used in that run; Appendix D). (2) It names a tool that exists nowhere. (3) An instruction-tuned model, after one correct structured call, falls back to a call template it memorized, naming `exec_command`, the shell entry point of the Codex CLI release in our parsed corpus (Table 2), inside its own text markup; the release under evaluation exposes the shell as `shell` and accepts only structured calls, so the episode ends. 47 of Qwen3.5-27B's 81 failed Codex CLI episodes end this way.

Table 6: The earlier run: pass@1 (%), one evaluation per cell. Base is the model before fine-tuning, under the same harness; Δ is HINT minus control, and bold marks the better of the two.

	SWE-bench Verified				SWE-bench Multilingual			
Harness	Base	Control	HINT	Δ	Base	Control	HINT	Δ
Claude Code (source)	25.2	47.8	**48.2**	+0.4	14.7	33.7	**35.7**	+2.0
OpenCode	16.0	**43.0**	42.0	−1.0	13.3	23.3	**31.0**	+7.7

Table 6 gives the result. Two differences stand out, both on Multilingual: 2.0 points on Claude Code and 7.7 points in the unseen-harness cell, OpenCode (93/300 against 70/300). Measured as the accuracy each model loses when moved from Claude Code to OpenCode, HINT loses 4.7 points on Multilingual against the control's 10.3, but 6.2 on Verified against the control's 4.8: on Verified this run shows no benefit. The OpenHands SDK runs of this comparison are not reported: both models timed out at high rates (25% and 64% of settled trials), traced to a proactive-compaction threshold set above the context length actually reached, so that per-step prefill grew until the wall clock ran out. Appendix B uses this run's trajectories, OpenHands SDK included, only for statistics that do not depend on accuracy.

D EVALUATION DETAILS

Rounding. Pass rates are pass counts over task counts, shown to one decimal. Spreads and the differences quoted in the text are computed from the counts for single-evaluation cells and from the

per-cell means shown for the three-evaluation cells of Table 7, then rounded once, so they can differ from the difference of two displayed values by 0.1.

Harness releases. The releases evaluated here are Claude Code 2.1.140, Codex CLI 0.142.0, OpenCode 1.17.15 and OpenHands SDK 1.40.1. They differ from those of the parsed corpora (Table 2) in a few tool names: Codex CLI exposes its shell as `shell` rather than `exec_command`, and the OpenHands SDK build used for the earlier run (Appendix C) exposes `execute_bash` and `str_replace_editor` in place of `terminal` and `file_editor` and has no task tracker. Figure 14 quotes the evaluated names verbatim.

Training-surface statistics. Of the 26,031 rendered samples of Section 4.1, 10,335 use split tools, 10,511 a unified editor and 5,185 a monolithic tool; the samples vary A_1, A_3, A_4 and A_5 and fix A_2 to native function calling; a trajectory that contains a call outside the vocabulary (the `unknown` residual of Table 1) or fewer than three actions yields no rendering. Normalization moves every call out of the text channels, and we check that no call text remains in the reasoning or observations of any kept trajectory.

Full numbers of the step-matched comparison. Table 7 lists the cells behind Figure 3.

Table 7: Full numbers behind Figure 3: HINT versus the step-matched control, both fine-tuned from Qwen3.6-35B-A3B, pass@1 (%) on SWE-bench Verified and SWE-bench Multilingual. Fine-tuned cells are means of three evaluations; Base is the model before fine-tuning (Table 4). Δ is HINT minus control, and bold marks the better of the two (the lower, for spread). Mean, worst and spread are taken over the five harnesses.

	SWE-bench Verified				SWE-bench Multilingual			
Harness	Base	Control	HINT	Δ	Base	Control	HINT	Δ
Claude Code (source)	69.0	78.4	**78.6**	+0.2	62.3	68.7	**75.0**	+6.3
OpenCode	71.4	74.3	**77.1**	+2.8	68.3	71.9	**73.4**	+1.5
OpenHands SDK	71.0	77.2	**81.7**	+4.5	64.3	71.7	**74.0**	+2.3
Codex CLI	68.0	66.7	**75.4**	+8.7	61.3	66.0	**73.4**	+7.4
CHAMELEON	70.8	**78.4**	75.6	−2.8	62.3	73.4	**74.4**	+1.0
Mean	70.0	75.0	**77.7**	+2.7	63.7	70.3	**74.0**	+3.7
Worst	68.0	66.7	**75.4**	+8.7	61.3	66.0	**73.4**	+7.4
Spread	3.4	11.7	**6.3**	−5.4	7.0	7.4	**1.6**	−5.8

Surfaces of Section 4.6. The origin mimics Claude Code: split tools with its names and parameters, an engineering system prompt with formal tool documentation, numbered file views and Claude Code's turn conventions. The far settings are: A1 merges the tools into a single monolithic tool with anonymized tool and parameter names; A3 changes the persona of the system prompt and renders the tool documentation as tables; A4 drops line numbers, wraps observations in a fence, truncates them at 2,000 characters and changes the markers for empty and failed output; A5 removes the runtime reminder, adds a turn header, returns observations as user turns and ends an episode with a finish call. These runs exclude the few episodes still running when a cell was closed (at most 15 of 300), so their absolute accuracies are not comparable with Table 6.

Settings of every experiment. Table 8 lists, for each experiment, the models compared, the harnesses and benchmarks, the number of evaluations per cell, and whether the runs share the serving stack of Section 4.1.

E LIMITATIONS

Each experiment uses one base model and one source harness, and each fine-tuned model is a single training run. Because CHR's first naming tier recombines real tool names, 33% of the rendered samples carry a Claude Code name, 33% an OpenCode name and 21% a Codex CLI name, while

Table 8: Settings of every experiment. "Section 4.1 stack" is the vLLM serving stack of Section 4.1; "Appendix C conditions" is the earlier serving endpoint.

Where	Base model	Models compared	Harnesses	Bench.	Evals	Conditions
Section 4.2, Table 4	Qwen3.5-35B-A3B, Qwen3.6-35B-A3B, KAT-Coder-V2.5-Dev	not fine-tuned	four production, CHAMELEON	V, M	1	Section 4.1 stack
Section 4.3, Table 7	Qwen3.6-35B-A3B	HINT, step-matched control	four production, CHAMELEON	V, M	3	Section 4.1 stack
Section 4.4, Figure 4	Qwen3-30B-A3B-Instruct-2507	HINT of Appendix C, real-trajectory model	four production, CHAMELEON	V, M	1	Appendix C conditions for Claude Code and OpenCode, separate runs elsewhere
Section 4.5, Figure 5	Qwen3.6-35B-A3B	base, HINT of Table 7	CHAMELEON, three naming settings	M	1	separate runs
Section 4.6, Figure 6	Qwen3-30B-A3B-Instruct-2507	HINT of Appendix C, control	CHAMELEON, seven surfaces	M	1	separate runs; unfinished episodes excluded
Appendix C, Table 6	Qwen3-30B-A3B-Instruct-2507	HINT, control	Claude Code, OpenCode	V, M	1	Appendix C conditions

none carries an OpenHands SDK name or presents any held-out harness as a coherent whole; the held-out harnesses are therefore unseen as interfaces, not as vocabularies. Training surfaces keep first-class file tools, so a shell-only surface such as Codex CLI's is never rendered into training data. Whether the effect scales with model size or holds for a shell-only source harness remains open.

F EXTENDED RELATED WORK

SWE agents, scaffolds and their training data. Jimenez et al. (2024) established the benchmark, and Yang et al. (2024) showed that the agent–computer interface itself materially determines performance. Scaffolds since then range from a pipeline or a bare shell (Xia et al., 2024; SWE-agent Team, 2025) to the rich tool sets of OpenHands (Wang et al., 2025a;b) and of the production harnesses we study (Anthropic, 2025; OpenAI, 2025; OpenCode contributors, 2025), and recent measurements confirm that the choice among them is a first-order factor: it moves scores by as much as the choice of model (Zhang et al., 2026; Yao et al., 2026; Zheng et al., 2026), the two shares vary across benchmarks (Merrill et al., 2026), and single-run differences of a few points lie within evaluation noise (Bjarnason et al., 2026). On the training side, SWE-Gym (Pan et al., 2024), SWE-smith (Yang et al., 2025), R2E-Gym (Jain et al., 2025), SWE-rebench (Badertdinov et al., 2025; 2026), Skywork-SWE (Zeng et al., 2025) and Devstral (Rastogi et al., 2025) scale tasks and trajectories within one scaffold, while a growing line of work trains across many: mixing real trajectories from several scaffolds (Abadji et al., 2026), randomizing synthetic tool sets or environments (Kimi Team et al., 2025; Chen et al., 2026; Cao et al., 2026), or randomizing the harness during reinforcement learning (Huang et al., 2026; Xu et al., 2026; Yu et al., 2026). Those works vary the scaffold at evaluation time or collect trajectories in every harness; we treat the scaffold as a nuisance variable at training time and are, to our knowledge, the first to synthesize its surfaces by re-rendering one harness's real trajectories, and Section 4.4 compares the two directly.

Interface reliance, invariance and protocol representations. Gu et al. (2026) show with semantics-preserving renaming that trajectory SFT amplifies interface memorization over semantic understanding, May et al. (2026) that agents internalize runtime conventions, and prompt-format sensitivity (Sclar et al., 2023) and the loss of generalization under a frozen prompt template (Lin

et al., 2025) are the same phenomenon at the prompt level; diversity of tool settings (He et al., 2025) and function masking against name memorization (Lin et al., 2024) are precursors of CHR's naming tiers and anonymization. The remedy we adopt is domain randomization (Tobin et al., 2017) rather than a penalty as in invariant risk minimization (Arjovsky et al., 2019), and the bound of Ben-David et al. (2010) formalizes why exposure to a distribution of domains helps; our setting is unusually favourable to this family of methods, because the content-preserving property that von Kügelgen et al. (2021) must assume of an augmentation is here measured at 100.0000%, and the transformations between surfaces are explicit and composable rather than latent (Higgins et al., 2018). The Agent Data Protocol (Song et al., 2025) also defines a scaffold-independent trajectory representation and compiles it into several scaffolds' SFT formats; our protocol differs in being recovered from production harnesses by normalization, in factoring the surface into axes that are sampled rather than compiled to fixed targets, and in an invertibility that is proved and measured, with the algebraic reading of the protocol as a normal form following Allen & Hospedales (2019).