How Do AI Agents Spend Your Money? Analyzing and Predicting Token Consumption in Agentic Coding Tasks

Anonymous authors
Paper under double-blind review

Abstract

Wide adoption of AI agents in complex human workflows drives rapid growth of LLM token consumption.[1] When agents are deployed on tasks that can require a large amount of tokens, three questions naturally arise: **(1) How do AI agents spend the tokens? (2) What models are more token efficient?** and **(3) Can LLMs anticipate the token usage before task execution?** In this paper, we present a systematic study of token consumption in AI agents, analyzing up to **twelve frontier LLMs** across **two coding-agent harnesses** and **four agentic environments**. We find that: **(1)** agentic tasks are uniquely expensive, with input tokens rather than output tokens accounting for most of the consumption. **(2)** Token usage is **highly variable**: on the same task, the most expensive run typically consumes twice as many tokens as the cheapest, and higher usage does **not** guarantee higher accuracy. **(3)** Human-labeled task difficulty only **weakly aligns** with actual resource expenditure. **(4)** Models differ systematically in **token efficiency**, with similarly accurate models consuming up to 3$\times$ different amounts, but the **harness** can change consumption several-fold and even reorder models. **(5)** Finally, agents **show promise in anticipating their own cost**: the strongest models' pre-execution predictions correlate with actual usage at up to 0.77, a potential signal for **budget alerts**, although they tend to underestimate input tokens. Our study reveals important insights regarding the economics of AI agents and can inspire new studies in this direction.

1 Introduction

AI agents are being rapidly adopted across a wide range of tasks and domains, including but not limited to creative writing, customer support, data analysis and software development (Liu et al., 2023c; Yao et al., 2022; Schick et al., 2023; Liu et al., 2023a). Among these, *coding agents* that can read repositories, reason about issues, call tools, and propose patches with minimal human supervision (OpenAI, 2025; Liu et al., 2023b;c; Jimenez et al., 2024; Wang et al., 2023; 2024) are among the most widely used. Despite their wide adoption, the prevailing pricing model for coding agents has been widely criticized for two reasons: (1) lack of transparency—users do not know the final cost until a task is finished; and (2) no guarantee of completion—users still need to pay for the token costs even if the task fails (Kinde, 2024). These concerns converge on a central question: **Can we predict token consumption before a task is executed?** If we could estimate token usage up front, users could choose models or strategies accordingly, and providers could design clearer pricing tiers, enforce budget caps, and trigger early alerts for large bills.

In this paper, we present a systematic study of token consumption in AI agents. Our primary case study runs twelve frontier LLMs with the OpenHands agent (Wang et al., 2025c) on SWE-bench Verified (Jimenez et al., 2024), with four independent runs per task; we then test generalization under a second, lightweight harness (mini-SWE-agent (Yang et al., 2024)) and in customer-service (τ^2-bench (Barres et al., 2025)) and general-purpose (ClawEval (Ye et al., 2026)) environments. We ask how token consumption varies across tasks and runs, which token types drive costs, and whether differences between models persist across harnesses and environments.

[1] We use "token consumption" to refer to both input and output tokens used by LLM agents

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.

Our analysis reveals four key findings. First, **more complex tasks tend to consume more tokens on average**, but human-labeled difficulty only weakly predicts consumption, and on the same task the most expensive run typically consumes twice as many tokens as the cheapest. Second, unlike typical chat (Crystalcare AI, 2023) and reasoning (Gu et al., 2024) settings (Figure 7 in Appendix C.1), **input tokens dominate the overall bill** in agentic tasks, because every turn resends the accumulated context. Third, **more token consumption does not guarantee better outcomes**: holding the task fixed, extra tokens bring little accuracy gain. Finally, **token efficiency depends on both the model and the harness**: models can follow distinct strategies (few large steps or many small ones), and changing the harness alone can cut consumption by 79% at similar accuracy and reorder models. These patterns persist across harnesses and largely hold beyond coding.

Building on these observations, we study whether agents can predict their own input and output token usage before execution, using their tools and coding environment. We find that **agents show promise in anticipating their own cost**: the strongest models' predictions correlate with actual usage at up to 0.77, and prediction quality tends to rise with agent capability. Although they underestimate input tokens, such predictions could serve, once calibrated, as a **pre-execution signal for budget alerts**. Our work makes the following contributions:

- We conduct a large-scale study of token consumption in AI agents, covering **twelve frontier models** with **repeated runs per task**, **two harnesses**, and **multiple agentic environments**, and open-source **all agent trajectories** (over 34,000).
- We find that agent costs are **input-dominated** and **highly variable across runs**, that higher consumption does not reliably improve success, and that **human-labeled difficulty is a poor proxy for cost**: some tasks rated easy consume more tokens than the average hard task; these patterns **generalize across harnesses and environments**.
- We show that token efficiency is **partly a model property**: models follow distinct consumption strategies, taking few large steps or many small ones, but the **harness can change consumption several-fold and reorder models**.
- We propose and benchmark pre-execution **self-prediction** of token usage, showing that **agents show promise in anticipating their own cost** (correlation up to 0.77), a potential signal for **budget alerts and cost management**.

Taken together, our empirical analysis and prediction study illuminate where tokens go in AI agents and what can be anticipated before execution, providing concrete steps toward more predictable and user-aligned agent pricing.

2 Related Work

Token consumption and cost of LLM agents. Several studies decompose where tokens go in agent systems. Tokenomics (Salim et al., 2026) and AgentTaxo (Wang et al., 2025b) break down token usage in multi-agent software-engineering systems by phase or agent role, each with a single backbone model. Others evaluate agents on cost as well as accuracy: Kapoor et al. (2024) argue that agent evaluations should report cost, leaderboards such as HAL (Kapoor et al., 2025) and the OpenHands Index (OpenHands Team, 2026) now report cost alongside success rates, and SWE-Effi (Fan et al., 2025) and Efficient Agents (Wang et al., 2025a) study cost–accuracy trade-offs of agent systems. These works report aggregate cost per system or decompose a single model's usage. We instead characterize where tokens go within trajectories, how much consumption varies across repeated runs of the same task, and which patterns hold across twelve models, two harnesses, and several agentic environments (Appendix B compares these studies in detail).

Reducing agent cost and test-time compute. A complementary line of work reduces agent cost directly, through turn control for coding agents (Gao & Peng, 2025), budget-aware tool use (Liu et al., 2025), and memory and context management that limits how much history the agent resends (Packer et al., 2024; Lindenbauer et al., 2025). Studies of test-time scaling show that additional reasoning does not always improve accuracy and can even degrade it (Snell et al., 2024; Gema et al., 2025; Aggarwal et al., 2025), and that long inputs impair reasoning well before the context limit (Levy et al., 2024; An et al., 2024; Hong et al., 2025). Our findings connect the two: in multi-turn agents, cost is dominated by resent input context rather than generated reasoning, extra tokens on the same

task bring little accuracy gain, and the harness alone can change consumption several-fold at similar accuracy. This suggests context management, not limiting reasoning, is the main cost lever.

Self-knowledge and cost estimation. LLMs can partly assess their own knowledge and outputs (Kadavath et al., 2022; Madaan et al., 2023), which suggests they might also anticipate the effort a task requires. Pre-execution cost estimation, however, has mostly been left to external predictors or post-hoc accounting. We formalize the task of an agent predicting its own token consumption before execution and benchmark it across twelve models, relating its accuracy and overhead to the variability of consumption we observe.

3 Data and Method

Our core setting is SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024), a benchmark of 500 real-world GitHub issues paired with their repositories and tests, solved by agents under OpenHands (Wang et al., 2025c). We evaluate twelve models from five providers (Claude Sonnet 3.7, 4, 4.5, and 5, Claude Opus 5, GPT-5, GPT-5.2, GPT-6 Astra, Qwen3-Coder-480B, Gemini 3 Pro, Kimi K2, and Kimi K3), with four independent runs per task. To test generalizability (Section 5), we run four of these models on the same tasks under mini-SWE-agent (Yang et al., 2024), and three of them on τ^2-bench airline and retail (Barres et al., 2025) and ClawEval (Ye et al., 2026) under their native harnesses. In total, the study comprises over 34,000 agent trajectories (Appendix A.1).

In each run, the agent receives a task description and interacts with the environment autonomously over multiple rounds: the model generates a response, typically a tool call, whose result is appended to the interaction history and sent with the next request. From each round we record token usage by type (non-cached input, cache reads, cache writes, and output), monetary cost, and the agent's actions. Because prices and caching schemes differ across providers, we analyze token counts rather than dollars unless noted. Trajectory selection, token and cost accounting, and statistical procedures are described in Appendix A.

4 Token Consumption of Coding Agents

We use coding agents as our primary case study: as they are among the most widely deployed agent applications, and their core capabilities (file editing, command execution, tool use) underpin many other agentic workflows. We analyze SWE-bench Verified under OpenHands (Wang et al., 2025c) across twelve models, four runs each. Like production coding agents such as Claude Code (Anthropic, 2026) and Codex (OpenAI, 2025), OpenHands gives the agent specialized file-editing and shell tools and carries a long interaction history. Section 5 tests whether the findings hold under a different harness and in other environments.

Agentic tasks are uniquely expensive, and input tokens dominate their cost We first compare token usage across three coding-related tasks: code reasoning (Gu et al., 2024), coding chat (Crystalcare AI, 2023), and agentic coding (Jimenez et al., 2024; Chowdhury et al., 2024) (Appendix C.1). On average, an agentic coding task consumes 2.00M tokens, roughly $1{,}700\times$ as many as a single-round reasoning task and $600\times$ as many as a multi-round chat. This gap comes mainly from input tokens, which outnumber output tokens by more than 100 to 1 in agentic coding, compared with ratios near or below one in the other two tasks. Agentic workflows accumulate context from many sources, and because every LLM call resends the full interaction history, the same context is processed repeatedly and cumulative input grows roughly quadratically with the number of calls. Even though caching bills repeated context at a fraction of the input price, agentic tasks remain over $100\times$ more expensive per task. This contrast highlights how differently agentic tasks consume tokens and motivates our study of agent token consumption.

Token usage is highly variable across problem instances and runs Agents' token usage has large variances across different problems: averaged over runs and models, the most expensive problem costs $\sim$6 million more tokens than the cheapest (6.69M vs. 0.53M), and high-token-cost problems exhibit larger variance (Figure 1a). Consumption also varies when an agent works on exactly the same problem again: the most expensive successful run typically doubles the token cost of the least

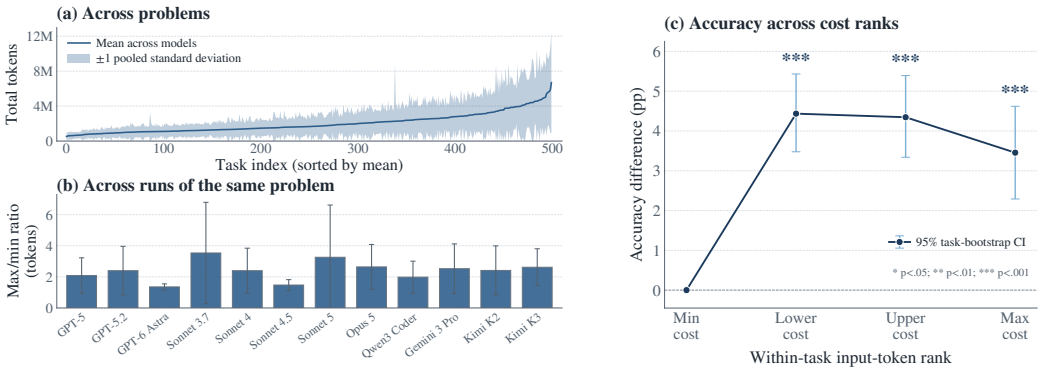


Figure 1: **Token consumption varies widely across problems and repeated runs, and costlier runs are not more accurate.** (a) Per-problem mean ± 1 SD across runs and twelve models, sorted by mean. (b) Per-model max/min token ratio among successful runs of the same problem (c) Accuracy of within-task input-token ranks relative to *MinCost*.

expensive one (mean ratio 2.39×; median 2.03× across all runs; Figure 1b). Together, these results suggest that token costs have large variances across problems and runs, making token usage prediction and agent pricing a fundamentally challenging task. Per-model breakdowns are in Appendix C.3.

More token consumption does not guarantee better outcomes Given the large variance in agent token consumption, one may wonder whether more tokens lead to better performance. Excluding Opus 5 (95.2%) and Kimi K3 (93.3%) to avoid ceiling effects, accuracy decreases as average token consumption increases across problems (Appendix C.4), a pattern also observed in recent evaluations of software agents (Fan et al., 2025; Wang et al., 2025a). This decline largely reflects problem difficulty, since harder problems both consume more tokens and fail more often. Yet even after controlling for difficulty, spending more tokens barely helps: in a between–within logistic regression with model fixed effects, the between-problem coefficient is strongly negative ($\beta = -0.55$, $p < 0.001$), while the within-problem coefficient, which compares runs of the same problem, is close to zero ($\beta = +0.03$, $p = 0.03$).

Figure 1c ranks the four runs of each problem by input tokens into *MinCost*, *LowerCost*, *UpperCost*, and *MaxCost*. Accuracy increases by 4.4 percentage points (pp) from *MinCost* to *LowerCost* but no further: *UpperCost* is flat and *MaxCost* is about one point lower (-0.89 pp relative to *UpperCost*, 95% CI $[-1.87, 0.11]$). This plateau is consistent with findings on *inverse test-time scaling*, where longer reasoning does not necessarily improve accuracy (Snell et al., 2024; Gema et al., 2025; Aggarwal et al., 2025), although our ranks compare naturally differing trajectories rather than assigned budgets. The pattern also differs by model generation (Appendix C.4): the seven older models tend to lose accuracy at their most expensive runs (*MaxCost* vs. *LowerCost*: -1.4 pp, 95% CI $[-2.7, -0.1]$), whereas the four most recent models show no such loss. One possible explanation is that costlier runs accumulate longer contexts, which can degrade reasoning well before the context limit (Levy et al., 2024; An et al., 2024; Hong et al., 2025), and that newer models are more robust to such long contexts. Ranking runs by output tokens gives a similar pattern (Appendix C.4).

We further examine repeated actions and find that high-cost runs involve more redundant work: compared with the cheapest run of the same problem, the most expensive run re-views already inspected files 1.34 more times and re-edits already modified files 2.81 more times on average, suggesting inefficient exploration (Appendix C.4).

Human-labeled difficulty only weakly predicts token consumption SWE-bench Verified labels each problem by the time a professional developer would need to resolve it (Jimenez et al., 2024; Chowdhury et al., 2024); we use three groups ("<15 min", "15 min–1 hour", and ">1 hour", merging the three ">4 hours" problems into the last). While consumption tends to rise with difficulty, the relationship is far from linear (Kendall $\tau_b = 0.36$; Figure 2): 4.6% of "<15-minute" tasks cost more than the average ">1-hour" task, and 6.7% of ">1-hour" tasks cost less than the average

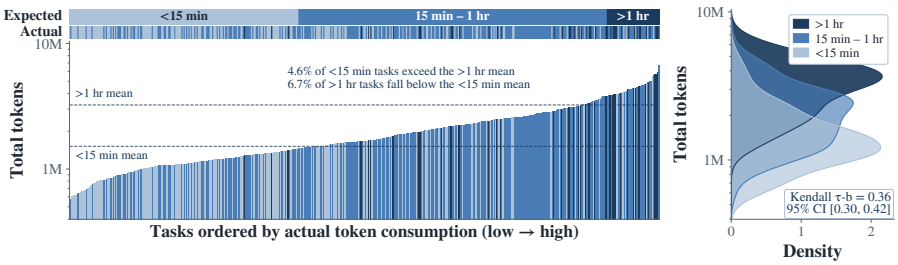


Figure 2: **Expert-rated task difficulty is a weak predictor of agent token consumption.** Left: the 500 SWE-bench Verified tasks sorted by token consumption (averaged over twelve models) and colored by human difficulty rating; the top strip shows the expected coloring under perfect alignment, while the actual coloring is scrambled throughout. Right: token-consumption density per difficulty group, with substantial overlap.

"<15-minute" task. Tasks that seem easy to humans may still demand extensive exploration, whereas some "hard" problems may be efficiently solvable given the model's prior knowledge or search strategies. Consequently, human-labeled difficulty is an imperfect predictor of resource expenditure. Per-model results are in Appendix C.5.

5 Generalizability of the Findings

Section 4 studied a single harness and benchmark. We test whether its findings persist in two steps: changing the harness while keeping the tasks fixed (Section 5.1), and changing the task domain, interaction structure, and harness together (Section 5.2). Table 1 summarizes both for the three models evaluated in every setting (GPT-6 Astra, Sonnet 5, and Kimi K3).

5.1 Across Harnesses

We run four of the twelve models (Sonnet 5, Opus 5, GPT-6 Astra, and Kimi K3) on the same 500 tasks, four runs each, under mini-SWE-agent (Yang et al., 2024), a lightweight harness in which the agent acts only through a single bash tool. The harness changes consumption nearly fivefold at similar accuracy: averaged over the four models, mini-SWE-agent consumes 79% fewer tokens per task than OpenHands (0.34M vs. 1.60M) while resolving a similar share of tasks (91.3% vs. 90.1%). The reduction comes mainly from interaction length: mini-SWE-agent makes half as many LLM calls (19.3 vs. 38.8), while observation size per call is nearly unchanged (478 vs. 504 tokens), and because cumulative input grows roughly quadratically with the number of calls, halving the calls reduces input by about three quarters. OpenHands' longer system prompt (3,058 vs. 17 tokens) accounts for only about 7% of its per-task tokens. The reduction ranges from 65% for Kimi K3 to 90% for Opus 5, enough to reorder models: Opus 5 consumes the most tokens of the four under OpenHands but the second fewest under mini-SWE-agent (Appendix D).

The patterns of Section 4, however, persist. Input tokens dominate under both harnesses, runs of the same task vary to a similar degree (median max/min ratio 1.87× and 1.84×), human-labeled difficulty aligns similarly weakly with consumption ($\tau_b = 0.41$ and 0.39), and neither harness shows a reliable accuracy gain at the most expensive runs (Table 1). How many tokens an agent consumes is thus a property of the model–harness pair, whereas the patterns are not specific to OpenHands. Since the two setups also differ in prompts and model configurations, this comparison characterizes the harnesses as deployed rather than isolating a single design choice.

5.2 Across Environments

We next evaluate the three models on two customer-service domains in which the agent converses with a simulated user, τ^2-airline (50 tasks × 4 runs) and τ^2-retail (100 × 4) (Barres et al., 2025), and on a 100-task subset of ClawEval (3 runs) (Ye et al., 2026), a suite of general autonomous-agent tasks, each under its native harness. For τ^2 we count only the agent's tokens; difficulty is estimated from the other two models' success rates; and variability uses the first three runs in every setting.

Table 1: **Generalizability of the findings** for the three models evaluated in every setting. SWE-OH and SWE-mini denote SWE-bench Verified under OpenHands and mini-SWE-agent; tokens are agent-only. Within-task contrasts compare runs of the same model on the same task, ranked by total tokens (*LowerCost* is the median run when there are three runs). On SWE-bench Verified, the accuracy contrast uses GPT-6 Astra and Sonnet 5, since Kimi K3 exceeds 90% accuracy. Green: consistent across settings; red: varies across settings. †: statistically significant (95% task-bootstrap CI excludes zero; Appendix E.2).

	SWE-OH	SWE-mini	τ^2-airline	τ^2-retail	ClawEval	Across settings
Token profile						
LLM turns per task	35.4	20.9	11.4	12.4	5.6	Varies
Static prompt text / input (%, lower bound)	8.3	—[a]	22.6	21.0	—[a]	Varies
Findings from Section 4						
Input share of tokens (%)	98.9	97.5	96.6	97.7	96.8	Dominant
Cost–difficulty alignment (Kendall τ_b)	$0.16^{\dagger}$	$0.21^{\dagger}$	$0.40^{\dagger}$	$0.15^{\dagger}$	-0.05	Weak (except airline)
Accuracy, *MaxCost* − *LowerCost* (pp)	$+0.5$	$+1.0$	$+1.4$	$+5.4^{\dagger}$	-1.7	No reliable gain
Run variability (median per-task log-SD)	0.23	0.26	0.14	0.10	0.11	Higher on SWE

[a]Not measurable: mini-SWE-agent places its instructions in the first user message, and ClawEval does not export its system prompt.

Which findings generalize? Three patterns hold (Table 1). Input tokens account for 97–99% of consumption everywhere, even in ClawEval, whose trajectories average fewer than six LLM turns, so input dominance arises in any loop that resends its context each turn, not only in long-horizon coding. Cost aligns only weakly with difficulty in four of five settings ($\tau_b \leq 0.21$), with airline the exception (0.40). And more token consumption does not reliably improve accuracy: a task's most expensive run is not significantly more accurate than a cheaper run except in retail ($+5.4$ pp). Run-to-run variability, in contrast, depends on the environment: it is roughly twice as large on SWE-bench Verified (median log-SD 0.23–0.26) as elsewhere (0.10–0.14).

Why environments differ. Similar input shares arise from different sources: static prompt text accounts for at least 21–23% of τ^2 input but about 8% under OpenHands, where input grows mainly from accumulated tool observations, which suggests different mitigations (caching the prefix versus compressing history). Variability is highest where trajectories are longest (21–35 turns on SWE-bench Verified versus 6–12 elsewhere), although within an environment, trajectory length explains it only partly (Appendix E.4); in τ^2, the agent's token count also co-varies with the simulated user's across runs of the same task ($r = 0.48$–0.90).

6 Which Models Are More Token Efficient?

Token usage varies substantially across backbone models, reflecting different problem-solving strategies and interaction patterns. We first compare models under OpenHands (Section 6.1), then examine whether their differences persist when the harness changes (Section 6.2).

6.1 Model Differences under OpenHands

Figure 3(a) shows that models differ widely in how they trade accuracy against token consumption. Three of the newest models form the frontier: GPT-6 Astra reaches 85% accuracy with the fewest tokens (about 0.7M per task), Kimi K3 reaches 93% at about 1.5M, and Opus 5 is the most accurate (95%) at about 2.3M. Among the older models, GPT-5 and GPT-5.2 are cheap but moderately accurate, while Kimi K2 combines the highest consumption with the lowest accuracy. Consumption is not simply the price of accuracy: GPT-6 Astra and Sonnet 5 resolve a similar share of tasks (85.4% vs. 86.7%), yet Sonnet 5 consumes nearly three times as many tokens.

Because models do not solve the same tasks, we also compare the ten models below 90% accuracy on the 228 tasks that all of them solve and the 40 that all of them fail (Figure 3b). The ordering is largely consistent across both subsets: GPT-6 Astra, GPT-5, and GPT-5.2 are the cheapest, and Sonnet 4, Sonnet 4.5, and Kimi K2 the most expensive, so efficiency differences are systematic rather than an artifact of task selection. Still, model identity accounts for only 22.5% of the variance in log tokens across the twelve models, comparable to repeated runs of the same task (25.2%) and the model–task

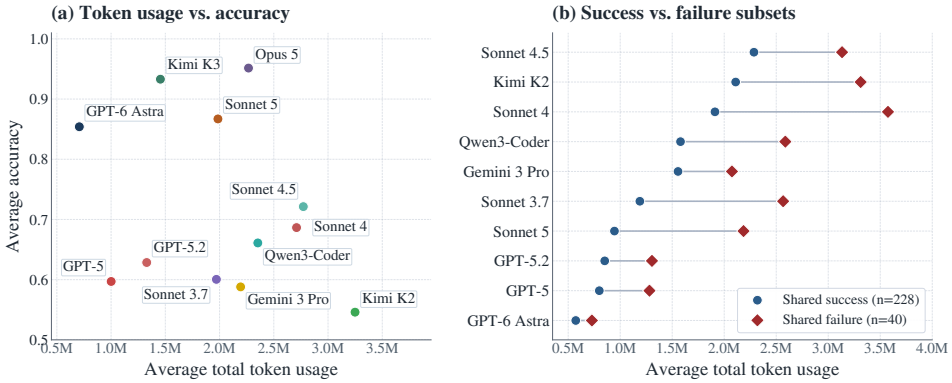


Figure 3: **Models exhibit systematic differences in token efficiency** (OpenHands). (a) Mean total token usage vs. accuracy for all twelve models. (b) Token usage on the shared success (n=228) and shared failure (n=40) subsets of the ten models below 90% accuracy; Opus 5 and Kimi K3 are excluded to keep the shared-failure subset large enough for comparison.

interaction (22.5%), with the task accounting for 29.8% (Appendix F.1). Is a model's efficiency its own, or does it depend on the setting in which it runs?

6.2 Do Model Differences Persist Across Harnesses?

To separate the model from the harness, we compare the four models run under both OpenHands and mini-SWE-agent on the same 500 tasks, relative to the average of GPT-6 Astra, Sonnet 5, and Kimi K3 on each task (Figure 4). The harness changes the ordering. GPT-6 Astra is the cheapest under both harnesses and Sonnet 5 stays at the same level (1.15×), but Kimi K3 rises from 1.21× to 1.79×, while Opus 5 falls from the most expensive (1.90×) to the second cheapest (0.65×) without losing accuracy.

The components in Figure 4(b) explain these shifts. Even on the same tasks and harness, models follow two distinct strategies. GPT-6 Astra takes few, large steps: fewer turns than the task mean (0.67× and 0.56×) but larger observations per turn (2.0× and 1.7×). Sonnet 5 and Kimi K3 take many small steps, with more turns (1.09–1.33×) and observations of only 0.49–0.74× the mean. Because every turn resends the accumulated context, the number of turns largely determines total consumption: in both harnesses, the models' ordering by total tokens matches their ordering by turns exactly.

Three of the four models keep their strategy across harnesses; Opus 5 does not. Under OpenHands it takes the most turns of any model (1.47×) and produces the most output per turn (1.61×); under mini-SWE-agent both fall to about average or below (0.70× and 1.04×), shifting it toward fewer, larger steps. Kimi K3 keeps its many-turn strategy but reads and writes more per turn, which raises its consumption. Token efficiency is therefore partly a model property: most models carry a consistent strategy across harnesses, but there can also be models, such as Opus 5, whose token consumption is highly sensitive to the harness.

7 Predicting Agent Token Consumption before Execution

Knowing a task's cost before execution would enable transparent pricing and informed user decisions. Since this cost depends on what the agent discovers while working, we ask whether the agent itself can predict it, much as an engineer estimates effort after scoping a task. As a case study, we evaluate *self-prediction* on SWE-bench Verified under OpenHands with twelve models, measuring prediction quality (Pearson correlation with actual usage) and its *overhead* relative to task cost.

Experimentation settings We use the coding agent itself to predict token consumption, motivated by its access to its own workflow (e.g., planning, tool usage, and exploration). Prior work on self-feedback and calibration (Madaan et al., 2023; Kadavath et al., 2022) suggests models can reason

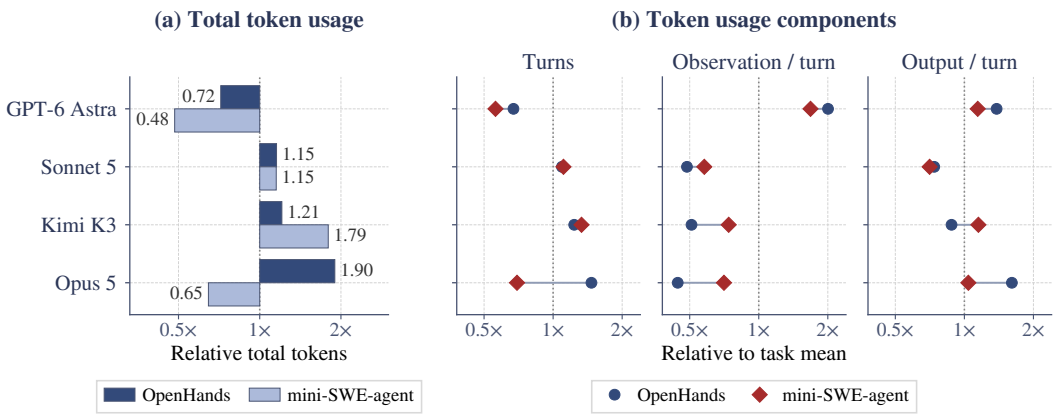


Figure 4: **Model differences across harnesses** on SWE-bench Verified, relative to the average of GPT-6 Astra, Sonnet 5, and Kimi K3 on the same task (log scale; 1× = average). (a) Total tokens (log-scale average over tasks); dark bars: OpenHands, light bars: mini-SWE-agent. (b) Turns, observation tokens per turn, and output tokens per turn; circles: OpenHands, diamonds: mini-SWE-agent. Intervals are in Appendix F.3.

about their outputs and uncertainty. Instead of solving the task, the agent estimates token usage while retaining full tool-calling capabilities to inspect the repository and anticipate execution paths, mirroring real-world effort estimation.

We adopt a fine-grained self-prediction setting when constructing the prompt, where the agent is explicitly instructed to decompose the task and produce detailed estimates of input tokens, output tokens, and total cost (see Appendix G.2). We additionally include a human-crafted prediction example to clarify the expected reasoning process and output format. We run three independent prediction runs per model on all 500 SWE-bench Verified tasks.

Results and observations Agents can partly anticipate their own token consumption (Figure 5a). GPT-6 Astra's predictions correlate with its actual usage at $r = 0.77$ for both input and output tokens, and Opus 5, Sonnet 5, and Kimi K3 reach 0.43–0.65. Prediction quality varies widely across models and tends to rise with agent capability: the four models that resolve the most tasks are also the four best predictors, suggesting that anticipating cost before execution may become increasingly feasible as agents improve. The strongest models also predict input tokens about as well as output tokens (Sonnet 5: 0.56 vs. 0.43; Kimi K3: 0.53 vs. 0.49), even though input depends on how context accumulates over the entire trajectory rather than on what the agent generates. GPT-6 Astra is also the model whose consumption varies least across runs (Figure 1b), consistent with run-to-run variability bounding how closely any pre-execution estimate can match a single run.

Calibration The remaining gap lies mainly in calibration rather than ranking (Figure 6). All models underestimate input tokens, including GPT-6 Astra, whose predictions track actual usage closely but fall below the diagonal, while its output predictions lie close to the diagonal. Correlation and calibration are distinct: Opus 5's output predictions correlate well with actual usage (0.65) yet fall well below it, whereas Kimi K3 tends to overestimate output. One plausible explanation for the input bias is that models estimate what they will read and write but not that every turn resends the accumulated context, which makes cumulative input grow roughly quadratically with the number of calls (Section 4). Because this bias is systematic, calibrating predictions on past runs could correct much of it. The bias persists without the in-context demonstration (Appendix G.3).

Potential for cost management Because the strongest models' predictions rank tasks reasonably well, they could help flag potentially expensive tasks before execution. Such flags could trigger budget alerts, request user approval, or send a task to a leaner harness, which can cut consumption several-fold at similar accuracy (Section 5.1). Self-prediction also complements human-labeled difficulty: difficulty labels align only weakly with consumption (Section 4) and are unavailable for new tasks, whereas self-prediction can be applied to any incoming task and reflects the specific

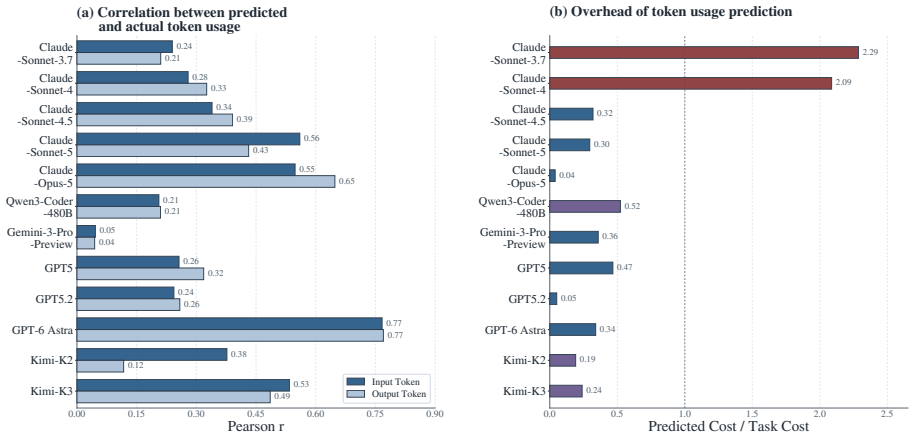
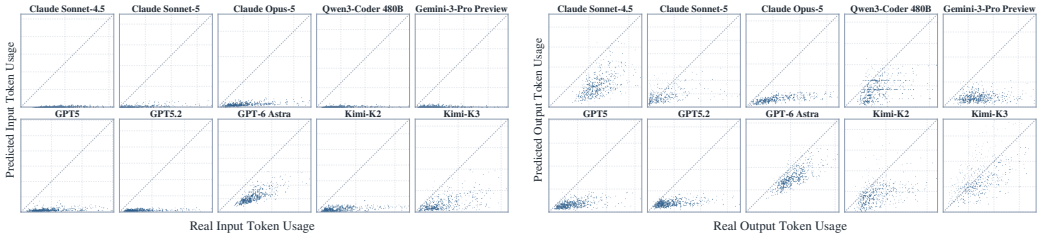


Figure 5: **Self-prediction performance and overhead across models.** (a) Pearson correlation between predicted and actual input and output tokens. (b) Prediction cost relative to task cost.



(a) (b)

Figure 6: **Predicted vs. real token usage across models.** The diagonal indicates perfect calibration. The strongest models' predictions track actual usage, but all models underestimate input tokens (a); GPT-6 Astra's output predictions lie close to the diagonal (b).

model that will execute it, which matters because consumption depends on the model–harness pair (Section 6). Since runs of the same task routinely differ by a factor of two, calibrated predictions are best presented as ranges rather than point estimates.

Self-prediction is also affordable. For ten of the twelve models, prediction costs less than half of the task itself (Figure 5b), and spending more on prediction does not by itself improve it: GPT-6 Astra predicts best at $0.34\times$ the task cost, whereas Sonnet 3.7 and Sonnet 4 spend more than $2\times$ with correlations of at most 0.33. Prediction quality therefore depends less on how much a model explores than on how well it anticipates the course of the task.

8 Conclusion

With the rapid growth of agent token consumption, anticipating token usage before execution is essential for transparent and reasonable pricing of AI agents. In this paper, we studied token consumption across up to twelve frontier LLMs, two coding-agent harnesses, and four agentic environments. Token consumption is dominated by input tokens, varies widely across runs of the same task, aligns only weakly with human-labeled difficulty, and does not reliably buy higher accuracy. Models differ systematically in efficiency, yet the harness can change consumption several-fold and reorder models. Finally, agents show promise in anticipating their own consumption: the strongest models' predictions correlate with actual usage at up to 0.77, and once calibrated for their underestimation of input tokens, such predictions could support budget alerts and cost-aware deployment. We discuss limitations and implications for agent pricing and user transparency in Appendix H.

AI USE STATEMENT

Large language models are the object of study in this paper: the agents we evaluate are built on the models listed in Appendix A.1, and all reported trajectories, token counts, and predictions come from running these agents. Apart from this, we used AI assistants in a supporting role: to help retrieve related papers, to help modify analysis and plotting code, and to polish the manuscript's language and organization. All experimental design, analyses, and conclusions are the authors' own; we reviewed all AI-assisted code and text and checked the reported numbers against our data. The authors take full responsibility for the content of this paper.

REPRODUCIBILITY STATEMENT

Appendix A.1 lists the models, harnesses, benchmarks, numbers of tasks and runs, and success criteria for every setting; Appendix A.2 defines how token counts and dollar costs are computed; and Appendix A.3 specifies the aggregation rules, ceiling exclusion, within-task cost ranks, regressions, variability measures, difficulty definitions, and variance decomposition. The full self-prediction prompts, including the in-context example, are given in Appendix G. All benchmarks we use are publicly available, and we will release all agent trajectories and analysis code upon publication. Because the agents rely on model APIs whose behavior may change over time, individual runs cannot be reproduced token for token; we therefore run each task multiple times, report task-level bootstrap confidence intervals, and base our conclusions on patterns that hold across runs and models.

REFERENCES

Pranjal Aggarwal, Seungone Kim, Jack Lanchantin, Sean Welleck, Jason Weston, Ilia Kulikov, and Swarnadeep Saha. Optimalthinkingbench: Evaluating over and underthinking in llms. *arXiv preprint arXiv:2508.13141*, 2025.

Chenxin An, Jun Zhang, Ming Zhong, Lei Li, Shansan Gong, Yao Luo, Jingjing Xu, and Lingpeng Kong. Why does the effective context length of llms fall short? *arXiv preprint arXiv:2410.18745*, 2024.

Anthropic. Claude code. https://github.com/anthropics/claude-code, 2026. Accessed: 2026-09-25.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-Bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025. URL https://arxiv.org/abs/2506.07982.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Crystalcare AI. Code-feedback sharegpt. https://huggingface.co/datasets/Crystalcareai/Code-feedback-sharegpt-renamed, 2023. A large-scale dataset of coding-related multi-turn chat conversations derived from ShareGPT, focusing on code feedback and interactive code discussion.

Zhiyu Fan, Kirill Vasilevski, Dayi Lin, Boyuan Chen, Yihao Chen, Zhiqing Zhong, Jie M Zhang, Pinjia He, and Ahmed E Hassan. Swe-effi: Re-evaluating software ai agent system effectiveness under resource constraints. *arXiv preprint arXiv:2509.09853*, 2025.

Pengfei Gao and Chao Peng. More with less: An empirical study of turn-control strategies for efficient coding agents. *arXiv preprint arXiv:2510.16786*, 2025.

Aryo Pradipta Gema, Alexander Hägele, Runjin Chen, Andy Arditi, Jacob Goldman-Wetzler, Kit Fraser-Taliente, Henry Sleight, Linda Petrini, Julian Michael, Beatrice Alex, et al. Inverse scaling in test-time compute. *arXiv preprint arXiv:2507.14417*, 2025.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 16568–16621. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/gu24c.html.

Kelly Hong, Anton Troynikov, and Jeff Huber. Context rot: How increasing input tokens impacts llm performance. Technical report, Chroma, July 2025. URL https://research.trychroma.com/context-rot.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, et al. Language models (mostly) know what they know. *arXiv preprint arXiv:2207.05221*, 2022.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. Ai agents that matter, 2024. URL https://arxiv.org/abs/2407.01502.

Sayash Kapoor, Benedikt Stroebl, Peter Kirgis, Nitya Nadgir, Zachary S Siegel, Boyi Wei, Tianci Xue, Ziru Chen, Felix Chen, Saiteja Utpala, Franck Ndzomga, Dheeraj Oruganty, Sophie Luskin, Kangheng Liu, Botao Yu, Amit Arora, Dongyoon Hahm, Harsh Trivedi, Huan Sun, Juyong Lee, Tengjun Jin, Yifan Mai, Yifei Zhou, Yuxuan Zhu, Rishi Bommasani, Daniel Kang, Dawn Song, Peter Henderson, Yu Su, Percy Liang, and Arvind Narayanan. Holistic agent leaderboard: The missing infrastructure for ai agent evaluation, 2025. URL https://arxiv.org/abs/2510.11977.

Kinde. AI token pricing optimization: Dynamic cost management for LLM-powered SaaS. https://kinde.com/learn/billing/billing-for-ai/ai-token-pricing-optimization-dynamic-cost-management-for-llm-powered-saas, 2024.

Mosh Levy, Alon Jacoby, and Yoav Goldberg. Same task, more tokens: the impact of input length on the reasoning performance of large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15339–15353, 2024.

Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. The complexity trap: Simple observation masking is as efficient as llm summarization for agent context management, 2025. URL https://arxiv.org/abs/2508.21433.

Lei Liu, Xiaoyan Yang, Yue Shen, Binbin Hu, Zhiqiang Zhang, Jinjie Gu, and Guannan Zhang. Think-in-memory: Recalling and post-thinking enable llms with long-term memory. *arXiv preprint arXiv:2311.08719*, 2023a.

Tengxiao Liu, Zifeng Wang, Jin Miao, I Hsu, Jun Yan, Jiefeng Chen, Rujun Han, Fangyuan Xu, Yanfei Chen, Ke Jiang, et al. Budget-aware tool-use enables effective agent scaling. *arXiv preprint arXiv:2511.17006*, 2025.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems, 2023b. URL https://arxiv.org/abs/2306.03091.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents, 2023c. URL https://arxiv.org/abs/2308.03688.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

OpenAI. Introducing codex, 2025. URL https://openai.com/blog/introducing-codex.

OpenHands Team. Introducing the OpenHands index. https://www.openhands.dev/blog/openhands-index, January 2026. Blog post, January 29, 2026.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024. URL https://arxiv.org/abs/2310.08560.

Mohamad Salim, Jasmine Latendresse, SayedHassan Khatoonabadi, and Emad Shihab. Tokenomics: Quantifying where tokens are used in agentic software engineering. *arXiv preprint arXiv:2601.14470*, 2026.

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

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

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

Ningning Wang, Xavier Hu, Pai Liu, He Zhu, Yue Hou, Heyuan Huang, Shengyu Zhang, Jian Yang, Jiaheng Liu, Ge Zhang, et al. Efficient agents: Building effective agents while reducing cost. *arXiv preprint arXiv:2508.02694*, 2025a.

Qian Wang, Zhenheng Tang, Zichen Jiang, Nuo Chen, Tianyu Wang, and Bingsheng He. Agenttaxo: Dissecting and benchmarking token distribution of llm multi-agent systems. In *ICLR 2025 Workshop on Foundation Models in the Wild*, 2025b.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025c. URL https://openreview.net/forum?id=OJd3ayDDoF.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

Bowen Ye, Rang Li, Qibin Yang, Yuanxin Liu, Linli Yao, Hanglong Lv, Zhihui Xie, Chenxin An, Lei Li, Lingpeng Kong, Qi Liu, Zhifang Sui, and Tong Yang. Claw-Eval: Towards trustworthy evaluation of autonomous agents. *arXiv preprint arXiv:2604.06132*, 2026. URL https://arxiv.org/abs/2604.06132.

APPENDIX

A Experimental Setup and Protocol . . . 14

A.1 Settings, Models, and Configurations . . . 14

A.2 Token and Cost Accounting . . . 14

A.3 Statistical Protocol . . . 14

B Extended Related Work and Positioning . . . 15

C Additional Results for OpenHands . . . 16

C.1 Agentic vs. Non-Agentic Coding . . . 16

C.2 Token–Cost Dynamics Across Phases and Rounds . . . 16

C.3 Per-Model Token Variability . . . 19

C.4 Accuracy and Repeated Actions by Cost Rank . . . 19

C.5 Per-Model Difficulty Alignment . . . 19

D mini-SWE-agent Comparison . . . 20

E Cross-Environment Details . . . 22

E.1 Per-Model Summary . . . 22

E.2 Confidence Intervals . . . 23

E.3 Failed versus Successful Runs . . . 24

E.4 Trajectory Length and Variability . . . 24

E.5 Agent–User Coupling in τ^2 . . . 25

F Stability of Model Consumption . . . 25

F.1 Variance Decomposition . . . 25

F.2 Pairwise Model Comparisons . . . 25

F.3 Component Values . . . 26

F.4 Tokens versus Recorded Dollars . . . 26

G Self-Prediction Details . . . 29

G.1 System Prompt . . . 29

G.2 In-Context Example and User Instruction . . . 31

G.3 Self-Prediction Without the In-Context Example . . . 36

H Discussion: Limitations, Pricing, and Transparency . . . 36

A Experimental Setup and Protocol

This appendix collects the settings, data selection rules, accounting conventions, and statistical definitions used throughout the paper.

A.1 Settings, Models, and Configurations

Table 2 summarizes all settings. On SWE-bench Verified under OpenHands, we evaluate twelve models: Claude Sonnet 3.7, Sonnet 4, Sonnet 4.5, Sonnet 5, Opus 5, GPT-5, GPT-5.2, GPT-6 Astra, Qwen3-Coder-480B-A35B-Instruct, Gemini 3 Pro (Preview), Kimi K2, and Kimi K3. The four most recent models (Sonnet 5, Opus 5, GPT-6 Astra, and Kimi K3) are also run under mini-SWE-agent. Three of them (Sonnet 5, GPT-6 Astra, and Kimi K3), which we call the *anchor* models, are run in every setting.

Table 2: Settings analyzed in this paper.

Setting	Harness	Models	Tasks × runs	Success criterion
SWE-bench Verified	OpenHands	12	500×4	Issue resolved
SWE-bench Verified	mini-SWE-agent 2.4.2	4	500×4	Issue resolved
SWE-bench Verified (self-prediction)	OpenHands	12	500×3	—
τ^2-airline	τ^2-bench	3 anchors	50×4	Reward $= 1$
τ^2-retail	τ^2-bench	3 anchors	100×4	Reward $= 1$
ClawEval	ClawEval	3 anchors	100×3	Task score ≥ 0.75

A.2 Token and Cost Accounting

Tokens Token consumption is the provider-reported input plus output tokens. Input includes non-cached input, cache reads, and cache writes; reasoning tokens are counted once, as part of output. In τ^2, only the agent's LLM calls are counted; simulated-user tokens are analyzed separately.

Dollar cost For each LLM call, let I be the total input tokens, of which C_{r} are read from the cache and C_{w} are written to it, and let O be the output tokens. The cost of the call is

$$\mathrm{Cost} = r_{\mathrm{in}}\,(I - C_{\mathrm{r}} - C_{\mathrm{w}}) + r_{\mathrm{w}}\,C_{\mathrm{w}} + r_{\mathrm{r}}\,C_{\mathrm{r}} + r_{\mathrm{out}}\,O, \tag{1}$$

where r_{in}, r_{w}, r_{r}, and r_{out} are the published per-token rates for non-cached input, cache writes, cache reads, and output. Claude models use explicit caching, with 5-minute cache writes in our setting.[2] GPT-5 and similar models cache input implicitly, without cache writes ($C_{\mathrm{w}} = 0$); at the time of our experiments, GPT-5 was priced at $r_{\mathrm{in}} = \$1.25$, $r_{\mathrm{r}} = \$0.125$, and $r_{\mathrm{out}} = \$10$ per million tokens.[3]

Dollar coverage Recorded costs are unavailable for Qwen3-Coder and for Kimi K3 under OpenHands, and are not recorded in τ^2 or ClawEval. Dollar results are therefore limited to the models and settings with recorded prices, and our main analyses use token counts.

A.3 Statistical Protocol

Aggregation Unless stated otherwise, we average over a model's runs on each task first, then over tasks, and weight models equally.

Ceiling exclusion Accuracy–consumption analyses exclude models whose overall resolve rate is at least 90%, since near-ceiling accuracy leaves little room for variation across runs. On SWE-bench Verified this excludes Opus 5 (95.2% under OpenHands, 95.7% under mini-SWE-agent) and Kimi K3 (93.3%, 94.2%). No anchor model is excluded in the other environments. All other analyses use every model.

[2] https://docs.claude.com/en/docs/about-claude/pricing

[3] https://openai.com/index/introducing-gpt-5/

Within-task cost ranks For each model and task, runs are sorted by token count (input tokens in Section 4; total tokens in Section 5.2), with ties broken by run index. With four runs, the ranks are *MinCost*, *LowerCost*, *UpperCost*, and *MaxCost*; with three runs (ClawEval), *LowerCost* denotes the median run and there is no *UpperCost*. Rank contrasts compare runs of the same model on the same task and are then averaged over tasks and models.

Accuracy regressions The grouped plots bin each model's tasks into groups of 25 by mean input tokens, after winsorizing at the 1st and 99th percentiles within model, and fit a linear trend with model fixed effects. The between–within logistic regression includes model fixed effects, each model–task pair's mean token usage (between-task term), and each run's deviation from that mean (within-task term). Both predictors are standardized, and standard errors are clustered by task.

Run-to-run variability Section 4 reports the max/min token ratio across runs of the same model and task. Figure 1b restricts this to successful runs on tasks solved by every model, requiring at least two successful runs per model–task pair; median ratios are computed over all runs without this restriction. Section 5.2 reports the median over model–task pairs of the standard deviation of log total tokens, using the first three runs in every setting so that estimates are comparable across environments.

Difficulty On SWE-bench Verified, we use the benchmark's developer time estimates: 194 tasks under 15 minutes, 261 tasks between 15 minutes and one hour, and 45 tasks over one hour (merging the two hardest categories). In Section 5.2, where human labels are unavailable, the difficulty of task i for model m is one minus the success rate of the other two anchor models on that task, so that no model's difficulty is computed from its own outcomes. Alignment is measured with Kendall τ_b between difficulty and mean log tokens, averaged over models in Section 5.2.

Variance decompositions For the twelve OpenHands models, let y_{mir} be the log total tokens of model m on task i in run r, and $\bar{y}_{mi}$ its mean over runs. Writing $\bar{y}_{mi} = \mu + \alpha_m + \beta_i + \gamma_{mi}$ with row and column means, the total variance decomposes exactly into model ($\mathrm{mean}_m\, \alpha_m^2$), task ($\mathrm{mean}_i\, \beta_i^2$), model–task interaction ($\mathrm{mean}_{mi}\, \gamma_{mi}^2$), and within-cell run variation ($\mathrm{mean}_{mi}\, \mathrm{mean}_r (y_{mir} - \bar{y}_{mi})^2$).

B Extended Related Work and Positioning

Comparison with prior studies Table 3 compares our study with recent work on agent resource consumption. Prior studies trade off harness coverage, model coverage, and analysis granularity in different ways: Tokenomics and AgentTaxo analyze token usage by phase but with a single backbone model, while SWE-Effi and Efficient Agents cover more systems but report aggregate cost. None repeats runs of the same task, which our results show is necessary to separate variation across tasks from variation across runs. Leaderboards such as the OpenHands Index report mean cost per model on a fixed harness; we explain where that cost comes from, how much it varies, and how it changes with the harness and the environment.

Table 3: Comparison with recent work on agent resource consumption.

Work	Domain	Harnesses	Models	Benchmarks	Repeated runs	Granularity
Tokenomics (Salim et al., 2026)	Multi-agent SE	1	1	30 custom SDLC tasks	✗	Phase
AgentTaxo (Wang et al., 2025b)	Multi-agent SE, reasoning	3	1	2 SWE + reasoning tasks	✗	Phase
SWE-Effi (Fan et al., 2025)	Coding	5	3	SWE-bench Verified (50)	✗	Overall cost
More with Less (Gao & Peng, 2025)	Coding	1	3	SWE-bench (100)	✗	Turn
Efficient Agents (Wang et al., 2025a)	Web, general	1	5	GAIA	✗	Overall cost–accuracy
Budget-Aware Tool-Use (Liu et al., 2025)	Web search	2	3	3 web-search benchmarks	✗	Overall cost–accuracy
Ours	Coding, customer service, general	2 + native	12	SWE-bench Verified (500), τ^2, ClawEval	✓	Phase, round, action

Choice of harnesses We use OpenHands as the primary harness because, like production coding agents, it provides specialized file-editing and shell tools and carries a long interaction history, and it is among the most widely used open-source agent frameworks. mini-SWE-agent provides a contrasting design with a single bash tool and a minimal system prompt. Harnesses with aggressive memory compression or summarization may change absolute token volumes; our cross-environment results suggest that input dominance persists whenever context is resent each turn, but we do not test compression-based harnesses directly.

Choice of benchmark SWE-bench Verified is the standard benchmark for agentic coding and the primary evaluation for OpenHands, SWE-agent, and many other systems. We use all 500 tasks with repeated runs, whereas several prior efficiency studies use smaller subsets (e.g., 50 tasks in Fan et al., 2025). Frontier models have been extensively optimized for SWE-bench, so absolute resolve rates are likely optimistic; our claims concern consumption patterns rather than accuracy levels, and Section 5.2 tests them beyond coding.

C Additional Results for OpenHands

This appendix supplements Section 4. Relevant definitions are described in Appendix A.3.

C.1 Agentic vs. Non-Agentic Coding

Figure 7 compares the input/output token ratio, average token usage, and average cost per task across three coding settings: code reasoning (Gu et al., 2024) (single-turn problem solving without tool interaction), code chat (Crystalcare AI, 2023) (multi-turn dialogue about a coding problem), and agentic coding (twelve OpenHands models on SWE-bench Verified). Agentic coding consumes about 2.00M tokens per task, roughly $1{,}700\times$ and $600\times$ as many as code reasoning and code chat, and costs about \$2.68 per task, compared with \$0.016 and \$0.023. Its input/output ratio (106) is two orders of magnitude higher than in the other two settings. The three settings use different task sets, so the comparison illustrates the scale of the gap rather than isolating the effect of the agentic loop.

C.2 Token–Cost Dynamics Across Phases and Rounds

To understand how different token types contribute to cost, we analyze token usage at both the phase and round levels. We use Claude Sonnet 4.5 as a case study, report per-model breakdowns for the eight older models, and replicate the analysis with Sonnet 5 under mini-SWE-agent. We decompose per-round cost into four token types: non-cached input, output, cache creation, and cache read tokens, each billed at different rates (Appendix A.2). Cache tokens are either written once for reuse or read in later rounds at lower cost. The total cost is the sum of these components.

Phase-level token usage dynamics We divide each trajectory into five phases: *Setup*, *Explore*, *Fix*, *Validate*, and *Closeout* (Table 4). For each phase, we compute average token counts and cost across 500 instances.

Phase	Description	Proportion
Setup	Task planning, environment setup, initial reproduction	9.98%
Explore	Code search, file inspection, root-cause analysis	30.37%
Fix	Code edits, debugging iterations, patch refinement	33.53%
Validate	Testing, regression checks, verification	16.59%
Closeout	Final checks, cleanup, summary output	9.53%

Table 4: Semantic phase decomposition of agent trajectories (Sonnet 4.5, OpenHands).

Figure 8 summarizes how token usage and cost evolve across the problem-solving process. In terms of token counts (Figure 8a), cache read input tokens dominate across phases, reflecting context reuse. The *Fix* phase has the highest token volume due to iterative debugging. Non-cached input and cache creation follow similar trends, indicating newly introduced context is cached. Output tokens are relatively higher in *Setup*, reflecting planning-heavy generation.

Cost dynamics largely mirror the token-count patterns (Figure 8b), with cache read input tokens remaining the dominant contributor due to continued context accumulation. However, differences in token pricing introduce noticeable deviations. In particular, output tokens contribute disproportionately to cost relative to their volume—for example, accounting for nearly a quarter of total cost in the *Setup* phase and exceeding the cost of both non-cached input and cache creation tokens despite a much smaller token proportion.

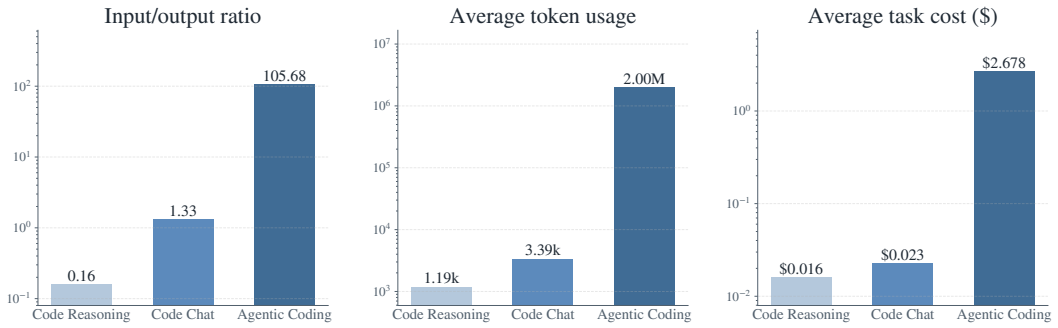
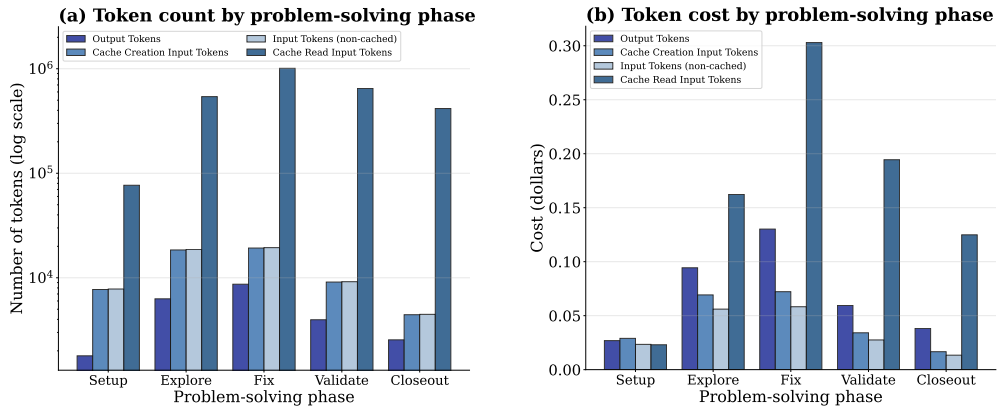


Figure 7: **Token usage and cost across three coding settings.** Code reasoning and code chat use fixed reference summaries; agentic coding averages twelve OpenHands models on SWE-bench Verified (cost averages the models with recorded prices). The settings differ in task distribution.



(a) Count per token type.

(b) Cost per token type.

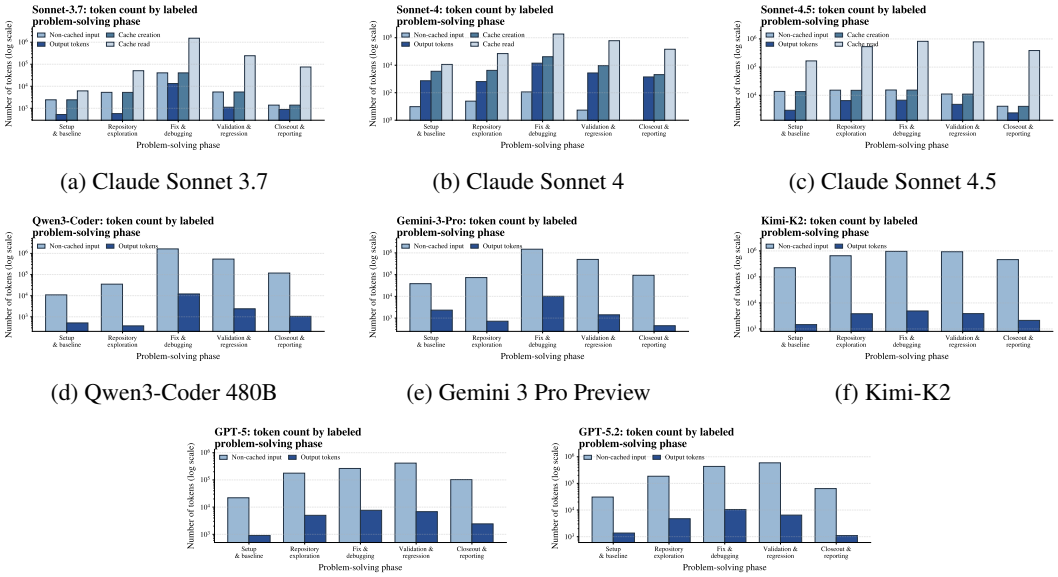
Figure 8: **Phase-level token usage and cost dynamics (Sonnet 4.5, OpenHands).** Phases (*Setup* → *Closeout*) reflect problem-solving stages, with token usage and cost varying across them.

Per-model phase decomposition Figures 9 and 10 extend this analysis to eight models. The qualitative pattern is consistent across models: cache read tokens dominate volume, while output tokens contribute disproportionately to cost.

Replication with Sonnet 5 under mini-SWE-agent The same pattern holds for a newer model under a different harness (Figure 11). Across all 2,000 Sonnet 5 mini-SWE-agent trajectories, cache reads account for 94.3% of token volume but only 43.3% of cost, while output tokens account for 1.4% of volume but 32.1% of cost; cache writes contribute the remaining 24.6% of cost. Costs are computed from the recorded rates (per million tokens: $2 non-cached input, $2.5 cache write, $0.2 cache read, $10 output).

Round-level cost breakdown To complement the phase-level aggregate view, we zoom into a representative trajectory. Figure 12 presents the exact per-round cost decomposition across token categories. The cost of *non-cached input tokens* closely tracks that of *cache creation tokens*, consistent with their similar per-token rates and the mechanism whereby newly introduced context is quickly persisted for reuse.

Cache read input tokens accumulate over rounds and provide a relatively stable baseline cost; however, total per-round cost is not monotonic, consistent with their weak correlation with overall cost. Instead, cost spikes are driven by discrete actions that introduce new tokens, such as repository exploration, file creation, test execution, and final summarization.

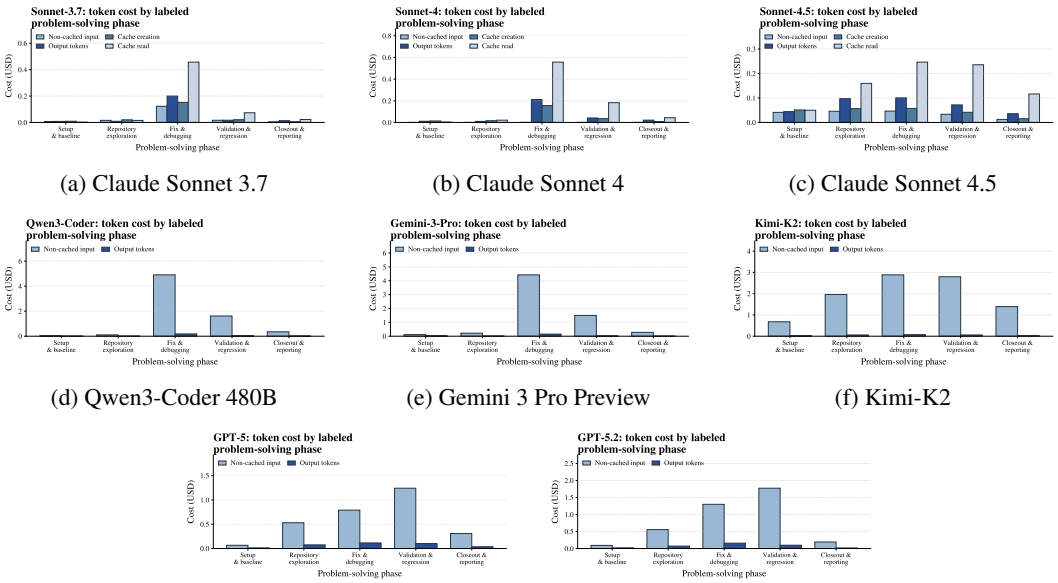


(a) Claude Sonnet 3.7 (b) Claude Sonnet 4 (c) Claude Sonnet 4.5

(d) Qwen3-Coder 480B (e) Gemini 3 Pro Preview (f) Kimi-K2

(g) GPT-5 (h) GPT-5.2

Figure 9: **Per-model phase-level token counts**, decomposed by token type.



(a) Claude Sonnet 3.7 (b) Claude Sonnet 4 (c) Claude Sonnet 4.5

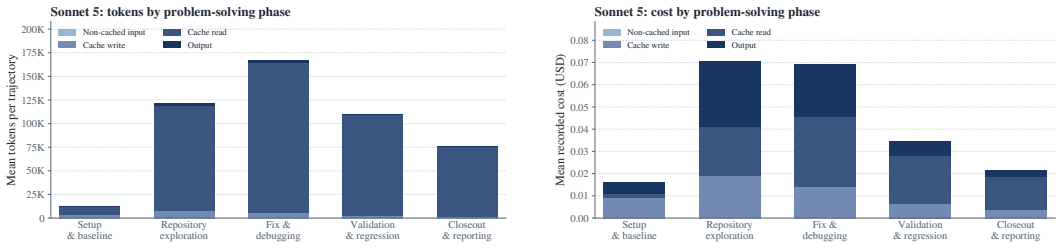
(d) Qwen3-Coder 480B (e) Gemini 3 Pro Preview (f) Kimi-K2

(g) GPT-5 (h) GPT-5.2

Figure 10: **Per-model phase-level token cost**, decomposed by token type.

These patterns align with the functional roles of different phases. In *Setup*, output tokens are primarily driven by planning and reasoning. During *Explore*, increased input usage reflects repository inspection and code reading. In later stages (*Fix*, *Validate*, *Closeout*), output tokens are often associated with script generation and edits, while input tokens arise from reading test results or execution outputs.

Trajectory behavior analysis We analyze representative rounds from a trajectory to understand how the agent balances tool use, code generation, and verification (Table 5). In three rounds (10, 23, 28), token usage is primarily driven by non-cached input arising from tool interactions in preceding steps, such as viewing files, executing tests, and cleaning up artifacts, which generate large amounts of content (e.g., test logs or file contents) fed into the model as input. In three other rounds (1, 17,



(a) Token volume by phase.

(b) Cost by phase.

Figure 11: **Phase-level token volume and cost for Sonnet 5 under mini-SWE-agent.** Cache reads dominate volume, while output tokens contribute disproportionately to cost, as for Sonnet 4.5 under OpenHands.

31), token usage is primarily driven by output tokens from reasoning and code generation: planning (`think`), creating test scripts, writing patches, and producing a final summary.

C.3 Per-Model Token Variability

Figure 13 shows per-instance token usage for each model individually. The heavy-tailed, high-variance pattern reported in the aggregate holds for every model, although its magnitude differs; GPT-6 Astra and Sonnet 4.5 vary least across runs (Figure 1b).

C.4 Accuracy and Repeated Actions by Cost Rank

Across-problem accuracy Figure 14a shows the across-problem relationship summarized in Section 4: grouping each model's tasks by mean input tokens, accuracy decreases as consumption increases. As the between–within regression shows, this slope largely reflects task difficulty rather than an effect of spending more tokens on the same task.

Per-model accuracy contrasts Figure 15 shows, for each model, the accuracy difference between its most expensive run of a task (*MaxCost*) and its second-cheapest run (*LowerCost*). The direction differs across models. Sonnet 3.7 (-4.6 pp, 95% CI $[-8.2, -1.0]$) and Sonnet 4 (-4.6 pp, $[-7.6, -1.8]$) are less accurate at their most expensive runs, whereas Gemini 3 Pro ($+7.0$ pp, $[+4.0, +10.2]$) and Kimi K3 ($+3.6$ pp, $[+1.6, +5.8]$) are more accurate; after Holm correction across the eleven four-run models, Sonnet 4, Gemini 3 Pro, and Kimi K3 remain significant. Pooled, the seven older four-run models show a small decline (-1.4 pp, $[-2.7, -0.1]$), whereas the four most recent models show no such loss, and the nine models below 90% accuracy no clear change (-1.0 pp, $[-2.0, +0.04]$).

Output tokens Ranking runs by output tokens instead of input tokens gives a similar pattern (Figures 14b and 14c). Across the nine eligible models, *MinCost*, *LowerCost*, *UpperCost*, and *MaxCost* have mean accuracies of 64.0%, 68.6%, 67.9%, and 67.7%; *MaxCost* minus *UpperCost* is -0.24 pp (95% CI $[-1.18, 0.69]$). The within-task output-token coefficient in the between–within regression is small and positive ($\beta = 0.048$, $p < 0.001$).

Repeated actions Higher-cost runs contain more repeated file operations (Figure 16). Across the eleven four-run models, *MaxCost* runs average 1.34 more repeated file views and 2.81 more repeated file edits than *MinCost* runs, and the pattern is the same when runs are ranked by output tokens. Because higher-cost runs are also longer, these counts partly reflect trajectory length; as a share of all tool calls, repeated calls rise only modestly with cost (Table 10).

C.5 Per-Model Difficulty Alignment

Table 6 reports, for each model, the Kendall τ_b between developer-estimated difficulty and the model's mean token consumption per task. Alignment is positive but weak for every model and harness (τ_b between 0.23 and 0.39). It is somewhat stronger for the four newest models (0.35–0.39) than for

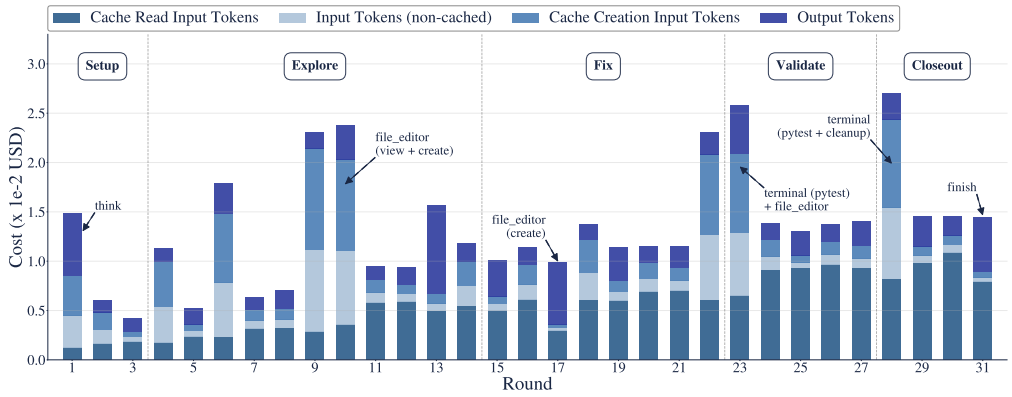


Figure 12: **Round-level token cost breakdown** for a representative Sonnet 4.5 trajectory. Per-round costs are decomposed into non-cached input, cache creation (5-minute persistence), cache reads, and output tokens. The dominant cost source shifts from cache creation in early phases to output tokens in later phases.

Round	Main cost	Tool usage	Action summary
1	Output	`think`	Plans the fix
10	Non-cached input	`file_editor`	Reads test file, creates reproducer
17	Output	`file_editor`	Writes debugging script
23	Non-cached input	`terminal`, `file_editor`	Runs tests, creates check script
28	Non-cached input	`terminal`	Runs tests, cleans up files
31	Output	`finish`	Summarizes the fix

Table 5: Representative trajectory steps illustrating input- and output-dominated rounds.

the eight older models (0.23–0.32), and similar under both harnesses. Per-model values are lower than the pooled value because averaging over twelve models smooths run-level noise in each task's consumption.

Model	OpenHands	mini-SWE-agent
GPT-5	0.24	–
GPT-5.2	0.28	–
Sonnet 3.7	0.28	–
Sonnet 4	0.25	–
Sonnet 4.5	0.30	–
Qwen3-Coder	0.32	–
Gemini 3 Pro	0.23	–
Kimi K2	0.24	–
GPT-6 Astra	0.35	0.35
Sonnet 5	0.35	0.38
Opus 5	0.38	0.34
Kimi K3	0.39	0.37
Pooled (all models in harness)	0.36 [0.30, 0.42]	0.39 [0.33, 0.45]
Pooled (four newest models)	0.41	0.39

Table 6: **Kendall τ_b between developer-estimated difficulty and mean token consumption.** Pooled values use task means averaged over models, with 95% task-bootstrap intervals.

D MINI-SWE-AGENT COMPARISON

This appendix supplements Section 5.1. The four most recent models (Sonnet 5, Opus 5, GPT-6 Astra, and Kimi K3) are run on the same 500 SWE-bench Verified tasks, four runs each, under both

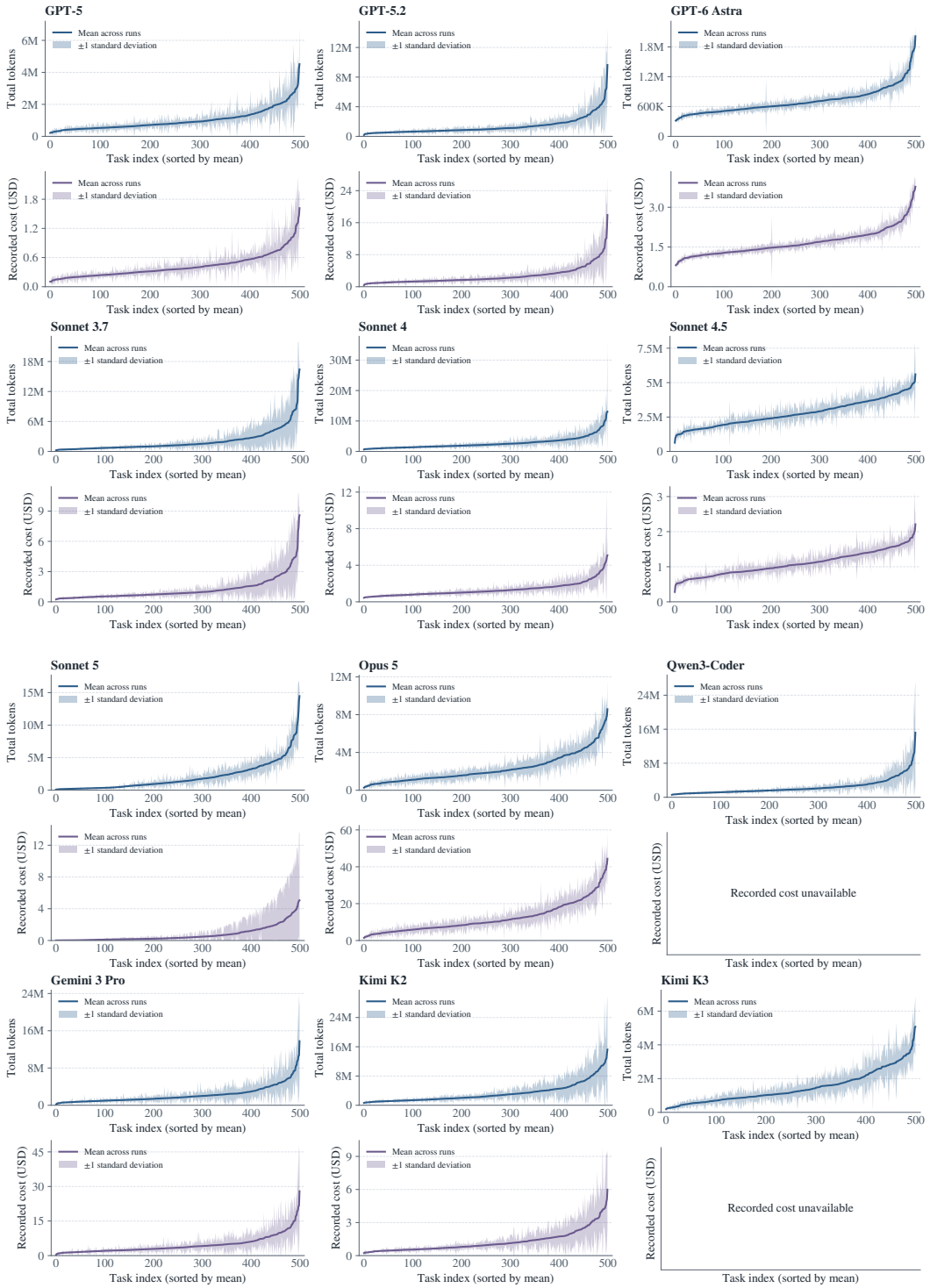
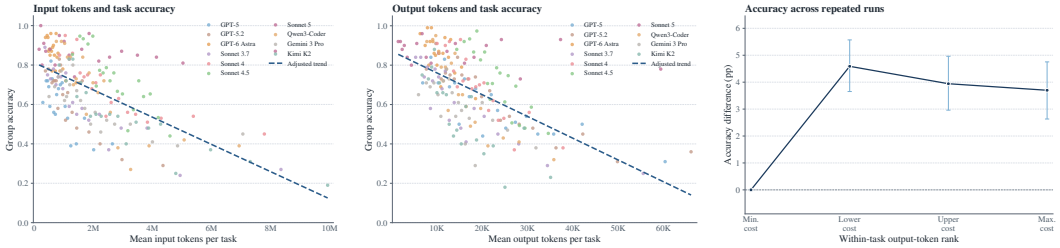


Figure 13: **Per-model token variability.** Per-task mean ± 1 SD across runs, with tasks sorted by mean, for each of the twelve OpenHands models.



(a) Across problems, input tokens. (b) Across problems, output tokens. (c) Within-task output-token rank.

Figure 14: **Accuracy and token usage, supplementing Figure 1c.** (a, b) Group accuracy versus mean input or output tokens (groups of 25 tasks per model; ten models below 90% accuracy), with a model-adjusted trend. (c) Accuracy of within-task output-token ranks relative to *MinCost*, with 95% CIs.

harnesses. Definitions follow Appendix A.3. Difficulty alignment under mini-SWE-agent is reported in Table 6, and the Sonnet 5 phase decomposition in Appendix C.2.

Per-model results Table 7 compares the two harnesses model by model. mini-SWE-agent uses fewer tokens for every model, with reductions from 65% (Kimi K3) to 90% (Opus 5), and every model resolves slightly more tasks. For every model, the token reduction exceeds the reduction in LLM calls (35–70%), consistent with cumulative input growing faster than linearly in the number of calls (Section 5.1).

Table 7: **Per-model comparison of OpenHands (OH) and mini-SWE-agent (mini)** on the same 500 tasks, four runs each. Tokens are mean total tokens per task; turns are LLM calls per task.

	Tokens (K)			Turns			Resolved (%)	
Model	OH	mini	Change	OH	mini	Change	OH	mini
GPT-6 Astra	707.2	107.4	−85%	20.8	10.5	−49%	85.4	87.5
Sonnet 5	1985.0	486.9	−75%	43.3	24.9	−42%	86.7	88.1
Opus 5	2266.8	229.1	−90%	48.9	14.5	−70%	95.2	95.7
Kimi K3	1454.8	515.1	−65%	42.2	27.3	−35%	93.3	94.2
Mean	1603.4	334.6	−79%	38.8	19.3	−50%	90.1	91.3

Consumption variability Token usage under mini-SWE-agent is as variable as under OpenHands, at a lower absolute scale (Figure 17). Mean task consumption ranges from 29.8K to 2.31M tokens, and input tokens account for 97.5% of all tokens. On the same problem, the most expensive run consumes 1.84× as many tokens as the cheapest (median over all runs), and 2.39× on average among successful runs of the 436 tasks solved by every model.

E Cross-Environment Details

This appendix supplements Section 5.2. The three anchor models (GPT-6 Astra, Sonnet 5, and Kimi K3) are evaluated on both SWE-bench Verified harnesses, τ^2-airline, τ^2-retail, and ClawEval. Settings and definitions are given in Appendix A.

E.1 Per-Model Summary

Table 8 reports accuracy, mean token consumption, and turns for each model and setting. Accuracy counts every selected slot, including execution failures; token means use runs with recorded usage.

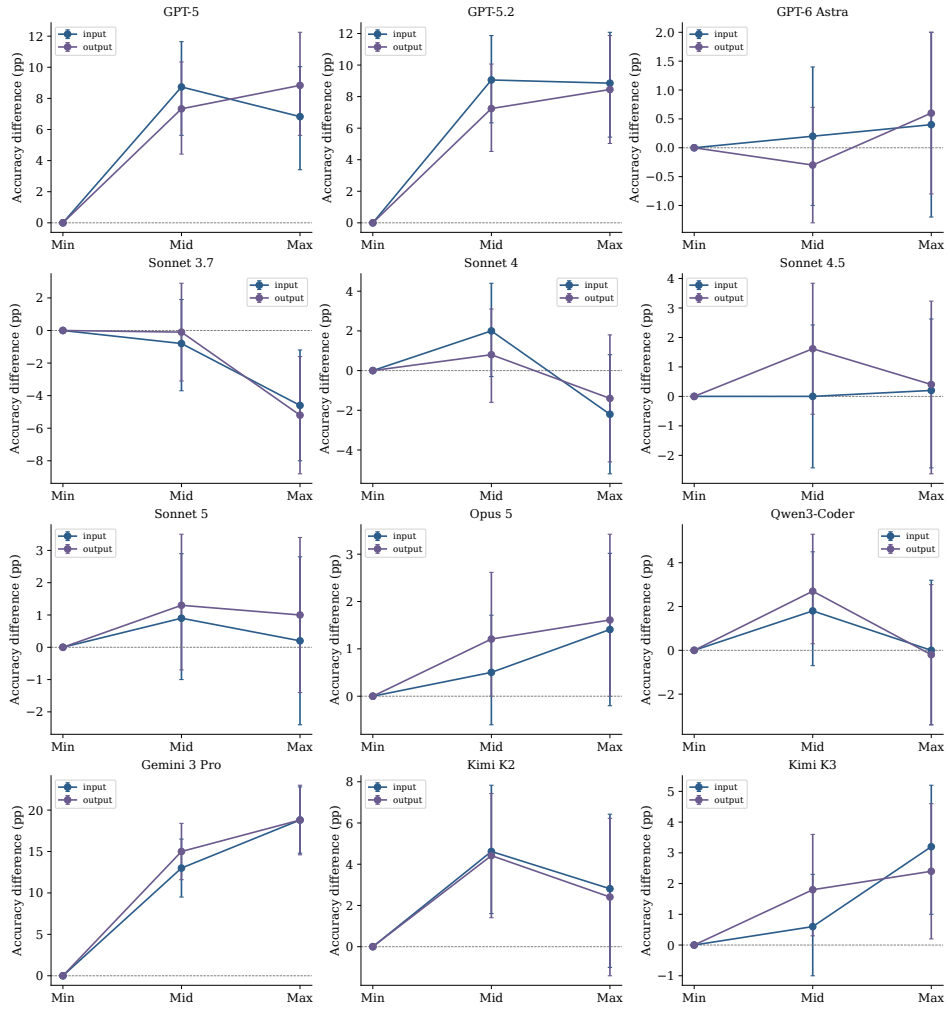
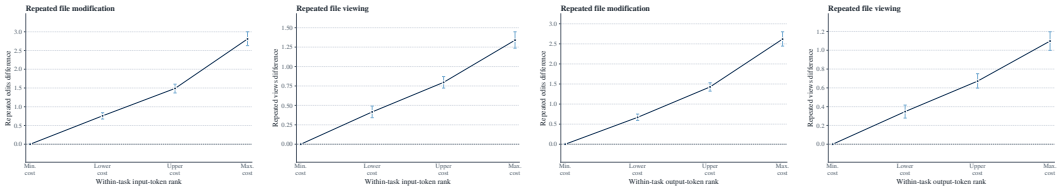


Figure 15: **Per-model accuracy contrasts**, *MaxCost* minus *LowerCost*, ranked by input tokens (left) and total tokens (right), with 95% task-bootstrap intervals. Purple marks models above 90% accuracy; gray marks Sonnet 4.5, whose contrast uses its third versus second run.

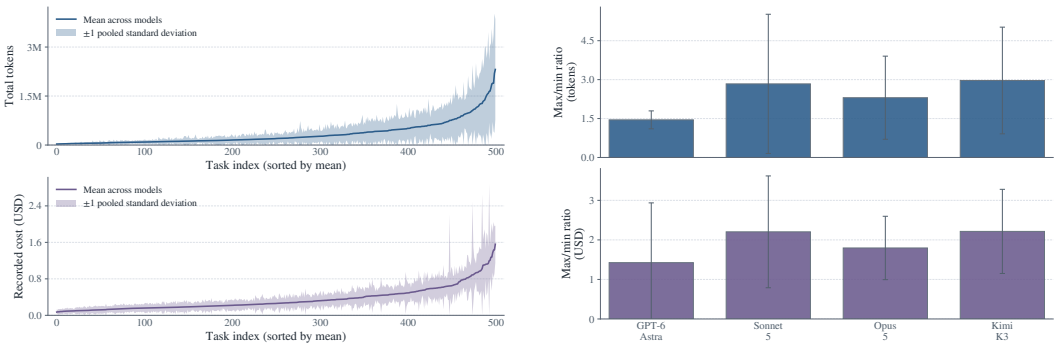


(a) Edits, input rank. (b) Views, input rank. (c) Edits, output rank. (d) Views, output rank.

Figure 16: **Repeated file operations by within-task cost rank** (eleven four-run OpenHands models). Differences are relative to *MinCost*, with 95% task-bootstrap intervals.

E.2 Confidence Intervals for Table 1

Tables 9 and 10 report 95% task-bootstrap intervals for the statistics in Table 1 (10,000 draws; Appendix A.3). The static-prompt share multiplies the size of the exported static text (system messages for SWE-bench, the domain policy for τ^2: 1,622 tokens in airline and 1,401 in retail) by the number of LLM calls and divides by total input tokens. It excludes tool schemas and any instructions placed in user messages, so it is a lower bound; ClawEval does not export its system prompt.



(a) Task-level variability.

(b) Run-to-run variability.

Figure 17: **Token variability under mini-SWE-agent.** (a) Per-instance mean ± 1 SD across runs and four models, sorted by mean. (b) Per-model max/min token ratio among successful runs of the same problem (436 problems solved by every model), with error bars showing ± 1 SD.

Table 8: **Per-model summary across settings.** Tokens are mean total tokens per run; turns are LLM calls per run.

Setting	Model	Runs	Accuracy (%)	Tokens (K)	Turns
SWE–OpenHands	GPT-6 Astra	2000	85.4	707.2	20.8
	Sonnet 5	2000	86.7	1985.0	43.3
	Kimi K3	2000	93.3	1454.8	42.2
	Opus 5	2000	95.2	2266.8	48.9
SWE–mini	GPT-6 Astra	2000	87.5	107.4	10.5
	Sonnet 5	2000	88.1	486.9	24.9
	Kimi K3	2000	94.2	515.1	27.3
	Opus 5	2000	95.7	229.1	14.5
τ^2-airline	GPT-6 Astra	200	78.0	82.6	13.4
	Sonnet 5	200	84.5	88.3	8.8
	Kimi K3	200	82.5	85.6	12.1
τ^2-retail	GPT-6 Astra	400	73.0	69.8	12.8
	Sonnet 5	400	84.0	106.3	11.4
	Kimi K3	400	80.0	86.8	12.9
ClawEval	GPT-6 Astra	300	75.7	126.6	5.2
	Sonnet 5	300	73.0	209.1	6.4
	Kimi K3	300	76.3	130.7	5.2

E.3 Failed versus Successful Runs

Table 11 compares the turns of failed and successful runs. Pooled over tasks, failed runs appear longer in airline ($+5.2$ turns), but this reflects which tasks fail: once runs are compared within the same model and task, the difference disappears. Within tasks, failed runs are not longer in any setting and are shorter on SWE-bench Verified under both harnesses and in retail; per model, the matched difference is negative for all three anchors under both SWE-bench harnesses (significant for five of the six model–harness pairs). In airline and ClawEval, each model has only 7–14 tasks with mixed outcomes, so per-model estimates are imprecise. This does not imply that more tokens improve accuracy: the difference is concentrated among the cheapest runs (under mini-SWE-agent, *MinCost* runs resolve 86.3% of tasks versus 87.8–88.8% for the other ranks), whereas the most expensive run is not reliably more accurate than the median run (Table 10).

E.4 Trajectory Length and Variability

Table 12 reports the Spearman correlation between a task's mean number of turns and its run-to-run variability (log-SD over the first three runs), computed at the task level after averaging over the

Table 9: **Level statistics with 95% intervals.**

Setting	Input share (%)	Median log-SD (3 runs)	Turns	Static prompt share (%)
SWE–OpenHands	98.92 [98.90, 98.94]	0.228 [0.216, 0.239]	35.4 [34.1, 36.7]	8.29 [8.12, 8.48]
SWE–mini	97.50 [97.38, 97.60]	0.256 [0.246, 0.269]	20.9 [20.1, 21.8]	0.12 [0.11, 0.12]
τ^2-airline	96.64 [96.38, 96.86]	0.138 [0.123, 0.168]	11.4 [10.2, 12.7]	22.56 [20.80, 24.67]
τ^2-retail	97.72 [97.62, 97.81]	0.098 [0.089, 0.109]	12.4 [11.8, 13.0]	21.03 [20.50, 21.59]
ClawEval	96.82 [94.36, 97.80]	0.106 [0.074, 0.138]	5.6 [4.6, 6.7]	–

Table 10: **Within-task contrasts and difficulty alignment with 95% intervals.** †: interval excludes zero. Accuracy and repeated-share contrasts are in percentage points; the turn contrast is failed minus successful runs of the same model on the same task.

Setting	Accuracy, Max − Lower	Repeated share, Max − Min	Difficulty τ_b	Turns, failed − successful
SWE–OpenHands	$+0.5$ $[-0.8, +1.8]$	$+0.53$ $[+0.37, +0.68]^\dagger$	0.164 [0.109, 0.218]†	-5.25 $[-8.08, -2.56]^\dagger$
SWE–mini	$+1.0$ $[-0.3, +2.4]$	$+0.31$ $[+0.21, +0.41]^\dagger$	0.205 [0.150, 0.256]†	-4.28 $[-6.24, -2.21]^\dagger$
τ^2-airline	$+1.4$ $[-4.7, +7.0]$	$+1.51$ $[+0.77, +2.32]^\dagger$	0.400 [0.276, 0.509]†	-0.03 $[-1.75, +1.66]$
τ^2-retail	$+5.4$ $[+1.0, +10.0]^\dagger$	$+0.41$ $[+0.15, +0.72]^\dagger$	0.145 [0.023, 0.260]†	-1.32 $[-1.86, -0.79]^\dagger$
ClawEval	-1.7 $[-4.7, +1.3]$	$+0.41$ $[-0.02, +0.96]$	-0.045 $[-0.200, 0.111]$	-0.16 $[-1.06, +0.70]$

three anchor models and as the average of per-model correlations. To avoid estimating turns and variability from the same runs, the last column uses turns from the fourth run to predict the variability of the first three (not possible for ClawEval, which has three runs). The association is clearest under mini-SWE-agent and in ClawEval, weak under OpenHands (significant only in the within-model average), and uncertain in τ^2, so trajectory length explains variability only partly and to different degrees across environments.

E.5 Agent–User Coupling in τ^2

In τ^2, the simulated user also consumes tokens, and its behavior can lengthen or shorten a conversation. Figure 18 shows the correlation between the agent's and the simulated user's log token counts across runs of the same task, after subtracting each model–task mean. The correlation is positive for every model and domain (airline: GPT-6 Astra 0.71, Sonnet 5 0.84, Kimi K3 0.48; retail: 0.79, 0.90, 0.77): runs in which the user consumes more tokens are also runs in which the agent does. Since both sides accumulate tokens over the same conversation, the direction of influence cannot be separated. The simulated user is Sonnet 4.5 for the GPT-6 Astra and Sonnet 5 agents and Sonnet 5 for the Kimi K3 agent, so user behavior is not identical across agents.

F Stability of Model Consumption

This appendix supplements Section 6. Definitions of the variance decomposition, relative consumption, and components are given in Appendix A.3. All comparisons use each harness's recorded model configuration (Appendix A.1).

F.1 Variance Decomposition

Table 13 decomposes the variance of log total tokens across the twelve OpenHands models. Model identity, the model–task interaction, and run-to-run variation each account for roughly a quarter of the variance, and task identity for slightly more. Restricting every model to its first three runs gives similar shares (model 23.0%, task 29.9%, model–task 24.5%, runs 22.6%).

F.2 Pairwise Model Comparisons

Table 14 compares every pair of models directly on the same tasks, without reference to the anchor baseline. GPT-6 Astra consumes fewer tokens than every other model in every setting, except against Kimi K3 in airline and ClawEval, where the difference is not significant after correction. The order of Sonnet 5 and Kimi K3 depends on the setting: they are indistinguishable under OpenHands, Kimi K3 is more expensive under mini-SWE-agent, and Sonnet 5 is more expensive in τ^2 and ClawEval.

Table 11: **Turns of failed minus successful runs.** "Pooled" compares all failed and successful runs; "matched" compares runs of the same model on the same task, using model–task pairs with at least one success and one failure.

Setting	Pooled	Matched [95% CI]	Mixed pairs
SWE–OpenHands	$+0.06$	-5.25 $[-8.08, -2.56]$	139
SWE–mini	$+0.94$	-4.28 $[-6.24, -2.21]$	138
τ^2-airline	$+5.16$	-0.03 $[-1.75, +1.66]$	34
τ^2-retail	-0.58	-1.32 $[-1.86, -0.79]$	91
ClawEval	-0.20	-0.16 $[-1.06, +0.70]$	37

Table 12: **Spearman correlation between mean turns and run-to-run variability**, with 95% task-bootstrap intervals. †: interval excludes zero.

Setting	Tasks	Task-level ρ	Mean within-model ρ	Run-4 turns, task-level ρ
SWE–OpenHands	500	$+0.02$ $[-0.07, +0.11]$	$+0.07$ $[+0.02, +0.12]^\dagger$	$+0.02$ $[-0.07, +0.11]$
SWE–mini	500	$+0.21$ $[+0.12, +0.29]^\dagger$	$+0.20$ $[+0.15, +0.25]^\dagger$	$+0.17$ $[+0.09, +0.26]^\dagger$
τ^2-airline	50	$+0.30$ $[-0.00, +0.57]$	$+0.11$ $[-0.09, +0.31]$	$+0.23$ $[-0.08, +0.52]$
τ^2-retail	99	$+0.19$ $[-0.02, +0.38]$	$+0.15$ $[+0.04, +0.26]^\dagger$	$+0.21$ $[+0.01, +0.40]^\dagger$
ClawEval	100	$+0.41$ $[+0.20, +0.59]^\dagger$	$+0.45$ $[+0.32, +0.57]^\dagger$	–

Opus 5 is the most expensive model under OpenHands but cheaper than both Sonnet 5 and Kimi K3 under mini-SWE-agent.

F.3 Component Values

Table 15 reports the unnormalized components behind Figure 4b. On SWE-bench Verified, GPT-6 Astra takes about half as many turns as Sonnet 5 and Kimi K3 but reads 2–4$\times$ more observation tokens per turn. In τ^2, turn counts are similar across models, and differences in per-turn sizes are smaller. ClawEval trajectories are short (about five turns) but have large observations per turn.

F.4 Tokens versus Recorded Dollars

Recorded dollar costs are available for GPT-6 Astra, Sonnet 5, and Opus 5 under both SWE-bench harnesses. Figure 19 applies the relative-consumption measure to recorded dollars for these three models. Dollar rankings differ from token rankings: Sonnet 5 consumes more tokens than GPT-6 Astra in both harnesses but costs less, because prices, cache discounts, and the share of cached input differ across providers.

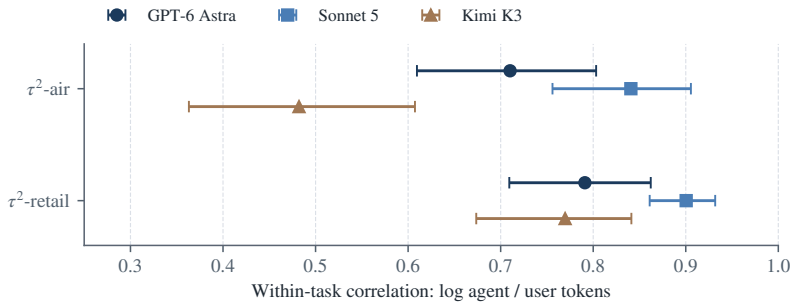


Figure 18: **Within-task correlation between agent and simulated-user log token counts in τ^2**, with 95% task-bootstrap intervals.

Table 13: **Decomposition of log-token variance across twelve OpenHands models**, with 95% task-bootstrap intervals.

Component	Variance	Share (%)
Model	0.180	22.5 [21.4, 23.8]
Task	0.238	29.8 [27.4, 32.0]
Model × task	0.180	22.5 [21.5, 23.6]
Runs within model–task	0.202	25.2 [23.9, 26.6]

Table 14: **Pairwise token ratios on the same tasks** (geometric mean ratio; below one means the first model consumes fewer tokens), with 95% intervals and Holm-adjusted p-values across all 21 comparisons.

Setting	Model pair	Ratio [95% CI]	Adj. p
SWE–OpenHands	GPT-6 Astra / Sonnet 5	0.62 [0.57, 0.67]	$< .001$
	GPT-6 Astra / Kimi K3	0.59 [0.57, 0.62]	$< .001$
	GPT-6 Astra / Opus 5	0.38 [0.36, 0.39]	$< .001$
	Sonnet 5 / Kimi K3	0.96 [0.90, 1.02]	0.332
	Sonnet 5 / Opus 5	0.61 [0.57, 0.65]	$< .001$
	Kimi K3 / Opus 5	0.64 [0.61, 0.66]	$< .001$
SWE–mini	GPT-6 Astra / Sonnet 5	0.42 [0.39, 0.45]	$< .001$
	GPT-6 Astra / Kimi K3	0.27 [0.26, 0.28]	$< .001$
	GPT-6 Astra / Opus 5	0.75 [0.71, 0.80]	$< .001$
	Sonnet 5 / Kimi K3	0.64 [0.60, 0.68]	$< .001$
	Sonnet 5 / Opus 5	1.78 [1.66, 1.91]	$< .001$
	Kimi K3 / Opus 5	2.78 [2.62, 2.95]	$< .001$
τ^2-airline	GPT-6 Astra / Sonnet 5	0.74 [0.65, 0.85]	$< .001$
	GPT-6 Astra / Kimi K3	0.86 [0.76, 0.98]	0.063
	Sonnet 5 / Kimi K3	1.16 [1.06, 1.28]	0.014
τ^2-retail	GPT-6 Astra / Sonnet 5	0.63 [0.58, 0.67]	$< .001$
	GPT-6 Astra / Kimi K3	0.79 [0.74, 0.84]	$< .001$
	Sonnet 5 / Kimi K3	1.26 [1.21, 1.31]	$< .001$
ClawEval	GPT-6 Astra / Sonnet 5	0.77 [0.67, 0.88]	0.001
	GPT-6 Astra / Kimi K3	0.93 [0.80, 1.08]	0.332
	Sonnet 5 / Kimi K3	1.20 [1.05, 1.39]	0.039

Table 15: **Mean components per run.** Observation tokens are text-size estimates; output tokens are provider-reported.

Setting	Model	Turns	Obs. tokens / turn	Output tokens / turn
SWE–OpenHands	GPT-6 Astra	20.8	1163	514
	Sonnet 5	43.3	280	285
	Kimi K3	42.2	295	331
	Opus 5	48.9	256	605
SWE–mini	GPT-6 Astra	10.5	829	387
	Sonnet 5	24.9	282	232
	Kimi K3	27.3	355	372
	Opus 5	14.5	342	341
τ^2-airline	GPT-6 Astra	13.4	139	256
	Sonnet 5	8.8	217	304
	Kimi K3	12.1	158	176
τ^2-retail	GPT-6 Astra	12.8	170	166
	Sonnet 5	11.4	237	189
	Kimi K3	12.9	193	118
ClawEval	GPT-6 Astra	5.2	10804	1099
	Sonnet 5	6.4	10882	1474
	Kimi K3	5.2	11320	1477

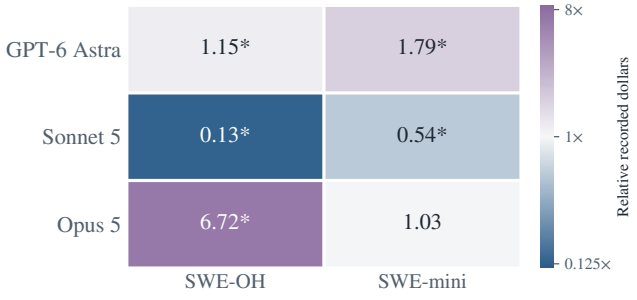


Figure 19: **Relative recorded cost** for the three models with recorded prices under both SWE-bench harnesses (geometric ratio to their mean on the same tasks); $*$: Holm-adjusted $p < .05$.

G SELF-PREDICTION DETAILS

Self-prediction (Section 7) uses all twelve models, three prediction runs each, on all 500 SWE-bench Verified tasks under OpenHands. The agent receives the system prompt in Appendix G.1 and the user instruction with an in-context example in Appendix G.2.

G.1 SYSTEM PROMPT

System Role You are an **OpenHands agent**, a helpful AI assistant that can interact with a computer to estimate token costs for software engineering tasks.
Role Definition Your primary role in this configuration is **token estimation** for SWE-bench–style software engineering tasks. You should thoroughly understand the task and repository using normal tools (e.g., bash, IPython, browsing, file viewing) as needed for analysis. However, instead of implementing a fix, you must **output a final token cost estimate** for completing the task with this agent. If the user asks a question (e.g., "why is X happening"), answer directly.
Token Estimation Objective Your goal is to predict how many LLM tokens will be consumed in total if this agent were to complete the task end-to-end. You must provide:

- `predicted_input_tokens`: Estimated total tokens sent to the LLM
- `predicted_output_tokens`: Estimated total tokens generated by the LLM
- `predicted_total_tokens`: Sum of input and output tokens
- `confidence`: A number in $[0, 1]$
- `breakdown_by_phase`: A JSON object breaking down input/output tokens across phases (e.g., repo_cloning, initial_reading, test_setup, debugging, coding_iterations, verification, review_cleanup)

Critical Constraint: Token Estimation Only IMPORTANT:

- You are a **token estimation agent**, not a problem-solving agent.
- Your only goal is to estimate token costs, not to fix bugs or implement features.
- You must call the `finish` tool with a JSON estimate.
- Never output actual code changes.
- If you find yourself writing code to fix the problem, stop and call `finish`.

Examples of what you should do:

- Explore the codebase to understand the problem.
- Run tests to observe current failures.
- Analyze code complexity and required changes.
- Estimate token usage for the complete solution.
- Call `finish` with a JSON estimate.

Examples of what you must not do:

- Write actual code fixes.
- Modify source files.
- Create test files.
- Commit changes.
- Call `finish` with code patches.

Efficiency Guidelines

- Each action you take is somewhat expensive.
- Wherever possible, combine multiple actions into a single command.
- Use efficient tools such as `find`, `grep`, and `git` with appropriate filters to minimize unnecessary operations.

File System Guidelines

- Do not assume user-provided paths are relative to the current working directory.
- Use efficient tools (e.g., `grep`, `find`, `cat`) to inspect files.
- You may read files for analysis, but must not modify source files.
- Delete any temporary files created strictly for exploration or testing.

Estimation Workflow

1. **Exploration:** Explore relevant files and understand the task context.
2. **Analysis:** Consider multiple solution approaches and estimate token costs for each phase.
3. **Testing (for estimation purposes):**
   - You may run existing tests to understand failures and complexity.
   - Do not create new tests or modify existing tests.
   - If tests cannot be run, estimate based on code analysis.
4. **Estimation:**
   - Synthesize findings into a token usage plan.
   - Base estimates on repository size, test runs, iterations, and tool usage.
   - Consider all phases: repo cloning, initial reading, test setup, debugging, coding iterations, verification, and review cleanup.
   - Delete any temporary exploration files.
5. **Output:** Produce the estimate as a JSON object and call `finish`.

Security

- Only use `GITHUB_TOKEN` or other credentials when explicitly requested and expected.
- Use APIs for GitHub or other platforms unless browsing is required.

Environment Setup

- If an application is missing, install it and rerun the command.
- If dependencies are missing:
  1. Look for dependency files (e.g., `requirements.txt`, `pyproject.toml`, `package.json`).
  2. Install dependencies using those files if present.
  3. Install individual packages only if no dependency files exist.

Troubleshooting

- If issues arise during exploration or testing:
  1. Document the issue in your reasoning.
  2. Factor added complexity into token estimates.
  3. Continue estimation with available information.

Token Estimation Output Format

```
{
  "predicted_input_tokens": <integer>,
  "predicted_output_tokens": <integer>,
  "predicted_total_tokens": <integer>,
  "confidence": <float between 0 and 1>,
  "breakdown_by_phase": {
    "repo_cloning": { ... },
    "initial_reading": { ... },
    "test_setup": { ... },
    "debugging": { ... },
    "coding_iterations": { ... },
    "verification": { ... },
    "review_cleanup": { ... }
  }
}
```

G.2 In-context Example and User Instruction

Running Example (Token Cost Estimation for a Software Engineering Task)

START OF EXAMPLE

USER: I've uploaded a python code repository in the directory `example_repo`. You are a TOKEN ESTIMATION agent. Estimate the token cost to fix the following issue description:

<issue_description>
Fix a bug where the version comparison function fails for development versions (e.g., `"1.2.3.dev0"`) when used across multiple modules in a large codebase with complex dependencies.
</issue_description>

ASSISTANT: I'll explore the codebase to understand the problem and estimate token costs.

<function=execute_bash>
<parameter=command>
```
cd example_repo && find . -name "*.py" -type f | head -50
```
</parameter>
</function>

USER: EXECUTION RESULT of [execute_bash]:

```
./src/core/version.py
./src/core/utils.py
./src/api/endpoints.py
./src/api/middleware.py
./src/processing/transformers.py
./src/processing/validators.py
./src/integration/external_api.py
./src/integration/adapters.py
./tests/unit/test_version.py
./tests/unit/test_integration.py
./tests/integration/test_api.py
./tests/functional/test_workflows.py
... (45 more files)
```

ASSISTANT: This is a larger codebase. Let me explore the structure and find where version comparison is used:

<function=execute_bash>
<parameter=command>

```
cd example_repo && grep -r "compare_versions\|LooseVersion"
↪  --include="*.py" | head -30
```

</parameter>
</function>

USER: EXECUTION RESULT of [execute_bash]:

```
./src/core/version.py:from distutils.version import LooseVersion
./src/core/version.py:def compare_versions(v1, v2):
./src/api/endpoints.py:from src.core.version import compare_versions
./src/processing/transformers.py:from src.core.version import
↪  compare_versions
./src/integration/external_api.py:from src.core.version import
↪  compare_versions
./src/integration/adapters.py:from src.core.version import
↪  compare_versions
... (12 more usages)
```

ASSISTANT: The version comparison function is used in many places. Let me read the main version module and understand the dependencies:

<function=execute_bash>
<parameter=command>

```
cat example_repo/src/core/version.py
```

</parameter>
</function>

USER: EXECUTION RESULT of [execute_bash]:

```
from distutils.version import LooseVersion

def compare_versions(v1, v2):
    return LooseVersion(v1) >= LooseVersion(v2)
```

ASSISTANT: Now let me check how it's used in the integration modules to understand the impact:

<function=execute_bash>
<parameter=command>

```
cat example_repo/src/integration/external_api.py | head -100
```

</parameter>
</function>

USER: EXECUTION RESULT of [execute_bash]:

```
import requests
from src.core.version import compare_versions
# ... (large file with 500+ lines using compare_versions in multiple
↪  places)
```

ASSISTANT: Let me check the test files to understand the expected behavior and test coverage:

<function=execute_bash>
<parameter=command>

```
cat example_repo/tests/unit/test_version.py
```

</parameter>
</function>

USER: EXECUTION RESULT of [execute_bash]:

```
import pytest
from src.core.version import compare_versions

def test_compare_versions():
    assert compare_versions("1.2.3", "1.2.2") == True
    assert compare_versions("1.2.3.dev0", "1.2.2") == True  # This
    ↪  currently fails

def test_compare_versions_edge_cases():
    # ... (multiple edge case tests)
```

ASSISTANT: Let me run the existing tests to see the current failures:

<function=execute_bash>
<parameter=command>

```
cd example_repo && python -m pytest tests/unit/test_version.py -v
```

</parameter>
</function>

USER: EXECUTION RESULT of [execute_bash]:

```
FAILED tests/unit/test_version.py::test_compare_versions - ValueError:
↪  invalid version number '1.2.3.dev0'
FAILED tests/unit/test_version.py::test_compare_versions_edge_cases -
↪  ValueError: invalid version number '1.2.3.dev0'
```

ASSISTANT: Let me also check integration tests that might be affected:

<function=execute_bash>
<parameter=command>

```
cd example_repo && python -m pytest tests/integration/ -v --tb=short
↪  2>&1 | head -50
```

</parameter>
</function>

USER: EXECUTION RESULT of [execute_bash]:

```
FAILED tests/integration/test_api.py::test_version_checking -
↪  ValueError: invalid version number '1.2.3.dev0'
FAILED tests/integration/test_api.py::test_external_api_integration -
↪  ValueError: invalid version number '1.2.3.dev0'
... (8 more failures)
```

ASSISTANT: Based on my exploration, I can now estimate the token costs. This is a complex issue that requires:

1. Understanding the problem across a large codebase with 50+ files
2. Reading and analyzing multiple files that use the version comparison function
3. Finding an alternative solution that maintains backward compatibility
4. Updating all affected modules and ensuring no regressions
5. Running comprehensive test suites (unit, integration, functional)

6. Multiple debugging iterations to handle edge cases
7. Verifying the fix works across all integration points

<function=finish>
<parameter=message>

```
{"predicted_input_tokens": 1250000, "predicted_output_tokens": 18500,
↪  "predicted_total_tokens": 1268500, "confidence": 0.82,
↪  "breakdown_by_phase": {"repo_cloning": {"input_tokens": 5000,
↪  "output_tokens": 800, "total_tokens": 5800, "description":
↪  "Exploring large codebase structure, finding all Python files,
↪  understanding directory layout"}, "initial_reading":
↪  {"input_tokens": 450000, "output_tokens": 3500, "total_tokens":
↪  453500, "description": "Reading issue description, searching for
↪  version comparison usage across codebase, reading version.py (~2000
↪  tokens), reading 8+ files that import it (~15000 tokens each),
↪  reading test files (~8000 tokens), understanding dependencies and
↪  integration points"}, "test_setup": {"input_tokens": 120000,
↪  "output_tokens": 2500, "total_tokens": 122500, "description":
↪  "Running unit tests, integration tests, functional tests, analyzing
↪  failure patterns, understanding test coverage requirements"},
↪  "debugging": {"input_tokens": 280000, "output_tokens": 4200,
↪  "total_tokens": 284200, "description": "Analyzing solution options
↪  (packaging.version vs custom parser), checking library
↪  availability, understanding backward compatibility requirements,
↪  evaluating impact on all integration points"}, "coding_iterations":
↪  {"input_tokens": 320000, "output_tokens": 4800, "total_tokens":
↪  324800, "description": "Implementing fix in version.py, updating
↪  all affected modules, handling edge cases, multiple test-run-fix
↪  cycles, ensuring no regressions in existing functionality"},
↪  "verification": {"input_tokens": 65000, "output_tokens": 2200,
↪  "total_tokens": 67200, "description": "Running full test suite
↪  (unit + integration + functional), verifying edge cases, checking
↪  integration points, validating backward compatibility"},
↪  "review_cleanup": {"input_tokens": 5000, "output_tokens": 500,
↪  "total_tokens": 5500, "description": "Final code review,
↪  documentation updates, ensuring code quality standards"}}}
```

</parameter>
</function>

END OF EXAMPLE

New Task Description Do **not** assume the environment is the same as in the example above.

Uploaded files:

`/workspace/{{ workspace_dir_name }}`

I've uploaded a python code repository in the directory {{ workspace_dir_name }}. You are a TOKEN ESTIMATION agent. Estimate the token cost to fix the following issue description:

<issue_description>
{{ instance.problem_statement }}
</issue_description>

Important Constraints

- You are a TOKEN ESTIMATION agent, NOT a problem-solving agent.

- Your ONLY goal is to estimate token costs, NOT to fix bugs or implement features.
- You MUST call the finish tool with a JSON estimate, NEVER with actual code changes.

Granularity Rules

- Unit: `token` (integer).
- Avoid coarse rounding (e.g., 100s/1000s). Use a step size of 1.

Phases for Estimation Your task is to estimate how many LLM tokens would be consumed to solve this problem if a coding agent were to complete it end-to-end.

Phase 1. Exploration: Explore the codebase to understand the problem

1. Read the problem description and understand what needs to be fixed
2. Explore relevant files and directories to understand the codebase structure
3. Search for key functions, classes, or variables related to the issue
4. Identify the root cause and complexity of the problem

Phase 2. Analysis: Analyze the complexity and required changes

1. Assess the scope of changes needed (number of files, lines of code)
2. Consider the debugging iterations likely needed
3. Evaluate the testing complexity and iterations
4. Estimate the number of tool calls and reasoning steps

Phase 3. Token Estimation: Calculate token usage for the complete solution

1. Estimate input tokens for:
   - Repository exploration and file reading
   - Code analysis and debugging
   - Implementation iterations
   - Testing and verification
2. Estimate output tokens for:
   - Reasoning and analysis responses
   - Code generation and explanations
   - Debugging responses
   - Test results interpretation
3. Calculate total tokens and confidence level

Phase 4. Finish: Provide final token estimate

1. Call the finish tool with a JSON object containing:
   - `predicted_input_tokens`
   - `predicted_output_tokens`
   - `predicted_total_tokens`
   - `confidence` (0–1)
   - `breakdown_by_phase`

Reminder Remember: You are estimating COSTS, not implementing SOLUTIONS. Do not write actual code fixes or modify any files. Your final deliverable is a JSON token estimate, not a working solution.

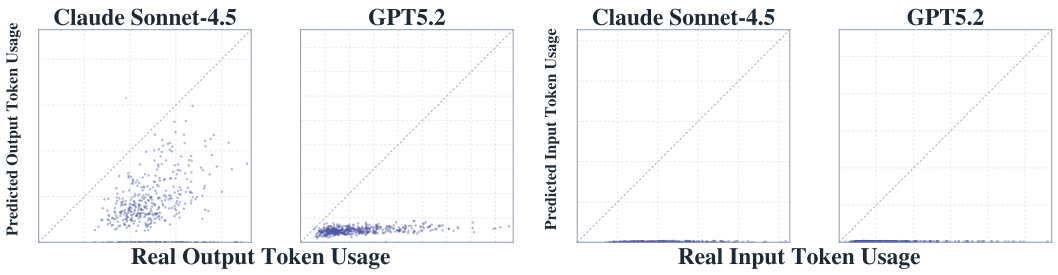
END OF NEW TASK DESCRIPTION

Output Constraint (Format) Please follow the format strictly. Please emit one and only one function call per message.

G.3 Self-Prediction Without the In-Context Example

To examine whether the observed underestimation is induced by the in-context demonstration used in our main setup, we conducted additional runs without providing any example. In practice, most models failed to consistently follow the instruction to perform token estimation without such a demonstration; we therefore report results for Sonnet 4.5 and GPT-5.2, which remained instruction-compliant. As shown in Figure 20, underestimation persists and becomes more severe, particularly for input tokens.

Table 16 further shows that correlation with real token usage degrades substantially without the in-context example. These results indicate that the downward bias is not caused by example-induced anchoring; instead, the demonstration improves calibration, while the underlying difficulty of anticipating long-horizon token growth remains.



(a) Output tokens.

(b) Input tokens.

Figure 20: **Self-prediction without the in-context demonstration** (Sonnet 4.5 and GPT-5.2). The dashed diagonal indicates perfect calibration. Underestimation remains pronounced.

Model	Token	Corr. with actual	Corr. (abs. error, task cost)	Corr. (prediction cost, task cost)
Sonnet 4.5	Input	0.136	0.116	0.119
Sonnet 4.5	Output	0.123	−0.456	0.119
GPT-5.2	Input	0.180	0.224	0.246
GPT-5.2	Output	0.213	0.082	0.246

Table 16: **Self-prediction without the in-context example.** Correlation with actual usage is lower than with the demonstration (Figure 5).

H Discussion: Limitations, Pricing, and Transparency

In this section, we discuss the limitations of our study and the implications of our findings for the design and pricing of agent-based systems.

Limitations Our analysis is observational. It describes how token consumption varies across models, harnesses, tasks, and runs, but does not assign token budgets or vary one design choice at a time, so its associations do not identify causal effects. The harness comparison and the cross-environment analysis compare setups that also differ in prompts and model configurations (Appendix A.1). The cross-environment and stability analyses cover three anchor models. Finally, frontier models have been extensively optimized for SWE-bench, so absolute resolve rates are likely optimistic, and exposure to its tasks during training may also help models anticipate a task's scope when predicting their consumption; our claims concern consumption patterns rather than accuracy levels.

Challenges for self-prediction Self-prediction shows promise, but three challenges remain before it can be deployed. First, predictions underestimate input tokens, the dominant share of cost, so they need calibration on past usage before being quoted as prices or compared against budgets. Second, self-prediction adds latency and overhead, especially for models that explore extensively before estimating, which may be unacceptable in interactive or time-sensitive settings. Third, integrating prediction and execution within a single run is difficult: both require substantial context construction, and combining them often confuses the agent, degrading instruction-following and estimate reliability. Making self-prediction calibrated, efficient, and integrated with execution remains an open challenge.

Budget alerts Our measurements suggest three considerations for budget-aware deployment. First, the strongest models' predictions rank tasks reasonably well and could help flag unusually expensive ones before execution (Section 7); calibrated on historical prediction–actual pairs, they could also be compared against a budget. Second, because runs of the same task routinely differ by a factor of two (Section 4), a budget should allow for this spread rather than treat a single expected cost as a hard limit. Third, a task's most expensive runs are not reliably more accurate than its typical runs (Section 4), and failed runs are not longer than successful runs of the same task (Appendix E.3). Cumulative cost is therefore a poor signal of impending failure, and a fixed token cap would stop successful runs as well as failing ones.

Agent pricing One of the central challenges for providers of agentic systems is how to price them. Traditional AI products such as ChatGPT often rely on subscription-based models, since typical users consume only a limited number of tokens. By contrast, agentic tasks can require very large amounts of tokens, even for seemingly simple problems, due to multi-step reasoning and tool use. Our findings show that token usage, especially input tokens, is highly variable and difficult to predict due to the stochastic nature of agent trajectories, and that it depends on the harness as well as the model: the same models consume 79% fewer tokens under mini-SWE-agent than under OpenHands at similar accuracy (Section 5.1). As a result, purely upfront pricing remains challenging, and consumption-based pricing may remain the most practical option until pre-execution estimates are better calibrated. Complementary approaches such as budget-aware tool-use policies (Liu et al., 2025) may help mitigate cost volatility by enforcing runtime token constraints.

User transparency Reliable token predictions are also important for user transparency and trust. Ideally, an agentic system would inform users about the likely cost of a task before execution, allowing them to make informed decisions. The strongest models can already rank tasks by expected cost reasonably well (Section 7). Calibrated on past usage, such estimates could let providers show cost ranges before execution, issue early warnings, request explicit user approval, or offer cheaper execution modes.