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

Anonymous ACL submission

Abstract

AI agents offer substantial opportunities to increase human productivity across many tasks. However, their use in complex workflows also drives rapid growth in LLM token consumption.[1] When agents are deployed on tasks that can require millions of tokens, two questions naturally arise: how do agents consume LLM tokens, and can we predict token usage before task execution? In this paper, we use the OpenHands coding agent as a case study and present the first empirical analysis of agent token consumption patterns using agent trajectories. We further explore the possibility of predicting token costs before task execution. We find that (1) Agent token consumption has inherent randomness even when executing the same tasks; (2) Unlike chat and reasoning tasks, input tokens dominate overall consumption and cost, even with token caching; (3) Predicting output-token amounts and the range of total consumption can achieve weak-to-moderate correlation, offering limited but nontrivial predictive signal. Surprisingly, we also find that (4) Higher token usage does not lead to higher accuracy, as tasks and runs that cost more tokens are usually associated with lower accuracy. We believe our study provides new and useful insights that can impact decisions around token consumptions for the growing number of AI agentic tasks.

1 Introduction

AI agents are being rapidly adopted across domains, from productivity and customer support to data analysis and software development(Liu et al., 2023b). Among these, *coding agents* are among the most widely used because they can read repositories, reason about issues, call tools, and propose patches with minimal human supervision (OpenAI, 2025; Liu et al., 2023a,b; Yang et al., 2024; Jimenez et al., 2024a). However, 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 may still pay 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 would better understand potential costs and choose models or strategies accordingly; providers could design clearer pricing tiers, enforce budget caps, and trigger early alerts.

In this paper, we present what is to our knowledge the first study on agent token consumption modeling and prediction. To understand the overall pattern of token usage in agentic coding workflows, we use the OpenHands agent (Wang et al., 2025) as a case study, and conduct an empirical analysis on SWE-bench-verified (Jimenez et al., 2024b) using trajectories generated by a wide range of different models. We investigate three questions: (1) Do coding agents consume different amounts of tokens across tasks and runs? (2) How does task difficulty relate to token consumption? (3) Which token types (input vs. output) drive costs in sequential agent executions, and how are they distributed over the trajectory?

Our analysis reveals four key findings. First, more complex tasks tend to consume more tokens on average, but usage varies substantially across runs; some runs use up to $10\times$ more tokens than others for the same task. Second, unlike typical chat and reasoning settings, *input tokens* dominate the overall bill in agentic coding, even when token caching is enabled. A recent study, AgentTaxo (Wang et al.), reports a similar pattern in multi-agent systems, where input tokens outweigh output tokens by a factor of 2–3 times due to inter-agent communication, reinforcing that agentic workloads are broadly input-heavy across settings. Third, token consumption is not concentrated in

[1] In this paper. we use "token consumption" to refer to both input and output tokens used by LLM agents

a single step: reading long contexts (files, diffs, and retrieved artifacts) and repeated tool-mediated expansions together contribute a large fraction of the input token budget. Finally, fourth, we find that tasks and runs with more token usage are associated with lower accuracy. These findings highlight both the heavy-tailed nature of token usage and the central role of context ingestion in agentic tasks.

Building on these observations, we study whether token consumption can be estimated *before* execution using the coding agent itself. We formalize the pre-execution estimation tasks in which the agent is asked to predict input and output token usage given all its available tools and the coding environment. Rather than relying on static predictors or handcrafted features, we leverage the agent's own planning, tool-usage reasoning, and repository exploration capabilities to produce cost estimates prior to execution. This formulation allows the agent to reason about expected workload using the same mechanisms it would employ during task solving.

Our experiments show that richer agent-side reasoning generally improves estimation quality. In particular, explicitly reasoning about anticipated tool usage and repository structure leads to more informative cost estimates. Estimating token consumption on a log scale yields better calibration over broad cost ranges, while *output-token* usage is consistently easier to estimate than *input-token* usage, reflecting the substantial variability introduced by retrieval and context construction. Although accurately predicting token usage remains challenging, agent self-prediction provides a practical, coarse-grained signal of relative cost. These results suggest that agent-driven estimation can support early *budget alerts* and approximate *cost ranges* before launching expensive runs, improving transparency without overpromising point accuracy.

Overall, this work contributes:

- the first empirical characterization of token consumption in agentic coding on SWE-bench, showing large run-to-run variance and the dominance of input tokens in overall costs;
- preliminary experiments on pre-execution token-cost prediction with comprehensive model settings;
- evidence that predicting output-token amounts and consumption ranges (especially in log space) is feasible but challenging, offering actionable signals for designing new pricing strategies for agentic systems.

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

2 Dataset

We collect agent trajectories on the SWE-bench Verified dataset (Jimenez et al., 2024b; Chowdhury et al., 2024) using the *OpenHands* framework (Wang et al., 2025), which we choose for its state-of-the-art performance among open-weight agents and its transparent, fully auditable execution pipeline. SWE-bench Verified is selected because it is the only large-scale software engineering benchmark rigorously validated by human annotators to exclude problematic or ambiguous cases.

Each problem is evaluated with four independent runs across a diverse set of agentic model backbones, including Claude Sonnet 3.7, Sonnet 4, Sonnet 4.5, GPT-5, GPT-5.2, Qwen3-Coder-480B-A35B-Instruct, Kimi-K2, and Gemini 3 Pro (Preview). These models were selected to span a range of architectures, training paradigms, and deployment settings, while offering strong coding capabilities and reliable execution stability. In our main analysis, we aggregate results across all models to identify behavioral and cost-related patterns that are consistent beyond any single backbone, thereby strengthening the generality of the observed phenomena. The collected data include full execution trajectories, inference logs, intermediate outputs, evaluation results, and metadata, enabling a comprehensive analysis of agent behavior and cost dynamics.

In this work, we focus specifically on token consumption throughout the end-to-end problem-solving process: given an initial task description, the LLM agent autonomously interacts with the environment to finish the task without any human intervention. For each problem instance, the agent proceeds in multiple rounds: in each round, the LLM generates a response based on the current prompt, followed by a tool call and execution. In particular, the full conversation history, including all previous prompts and completions, is carried forward unchanged into subsequent rounds.

To enable a more detailed analysis of LLM behavior and its corresponding token consumption

during problem-solving, we extract a set of fine-grained metrics from the LLM completion history. These metrics are obtained by parsing the structured JSON outputs of the agent and leveraging the usage information, which records all LLM interactions at each round. Together, the extracted metrics capture both the functional behavior of the agent, such as tool usage and file access patterns, as well as the underlying token-level dynamics. For all token-related metrics, we report values averaged over the four independent runs per problem.

3 An Empirical Investigation of Agent Token Consumption

3.1 Cost Variability and Inefficient Agent Behavior

Variations across different problems and different runs We begin by analyzing the variation in token consumption and tool usage across different problems (averaged over four independent runs) and across different runs for the same problem. For both total prompt tokens and total completion tokens, we observe substantial variation across problems, indicating a high degree of instability in token usage depending on the specific problem instance.

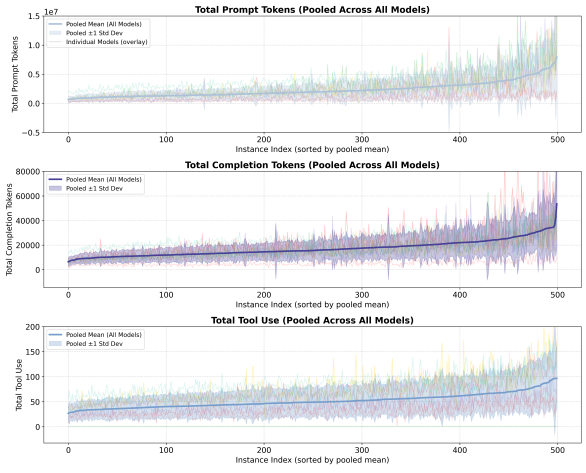


Figure 1: **Distribution of token and tool usage across SWE-bench instances.** Each curve shows the mean and standard deviation of total prompt tokens, completion tokens, and tool usage.

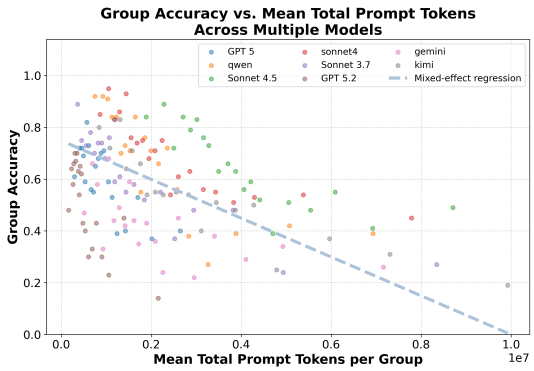
To further investigate this phenomenon, we aggregate the trajectories from all models, and sort all problems by their average total token cost (from low to high) and visualize the variation in both token usage and tool usage across the four runs for each problem (Figure 1). We observe that problems with higher overall token costs tend to exhibit substantially larger cross-run variance, indicating that the agent's behavior becomes increasingly unstable on more complex or longer tasks that require more tokens to solve. A similar pattern holds for tool usage: although the trend is less pronounced than for token consumption, notable variability still exists both across different problems and across runs of the same problem.

Accuracy and the Inverse Test-time Scaling Paradox We analyze how accuracy varies with token and tool usage across problems and cost levels using a regression-based aggregation across all models, which allows us to control for model-specific effects while capturing global trends.

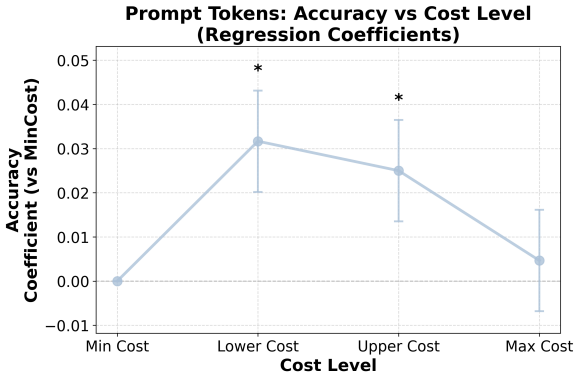
Figure 2(a) show the resulting relationships between accuracy and resource usage. Across models, accuracy tends to decrease as average token or tool consumption increases. Problems that demand more resources are often associated with runs in which the agent explores multiple unproductive trajectories, leading to longer contexts and reduced precision in reasoning. We observe the same trend when using completion tokens instead of prompt tokens, as shown in Appendix A.

To further examine this pattern, Figure 2(b) stratify accuracy by within-problem cost levels. For each problem, the four runs are ranked by total cost and grouped into *MinCost*, *LowerCost*, *UpperCost*, and *MaxCost*. Accuracy increases modestly from *MinCost* to *LowerCost*, but then declines for higher-cost bins, with the clearest drop observed at *MaxCost*. This non-monotonic trend is consistent with an *inverse test-time scaling* phenomenon (Snell et al., 2024; Wu et al., 2025), where additional computation reflects inefficient reasoning cycles and context bloat rather than improved problem solving.

Motivated by these observations, we further examine the behavioral patterns underlying high-cost failures by analyzing repeated file *view* and *modify* actions across cost levels. As shown in Figure 3, the frequency of repeated viewing and editing increases sharply with cost. This indicates that many expensive but failed runs are characterized by redundant back-and-forth file access and re-editing, reflecting inefficient search dynamics that inflate context length and token usage without proportional progress. While not all high-cost runs are dominated by redundancy, this pattern provides a concrete behavioral explanation for the inverse accuracy–cost relationship observed above.

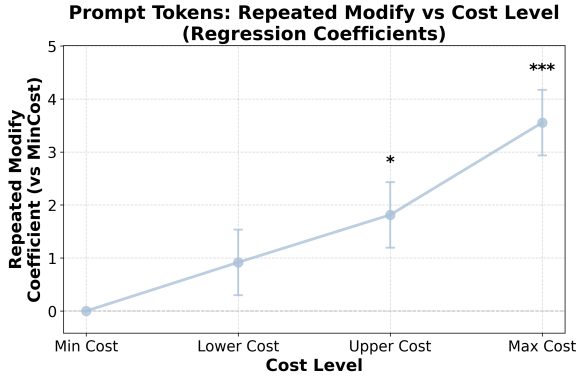


((a)) Group accuracy vs. mean prompt tokens.

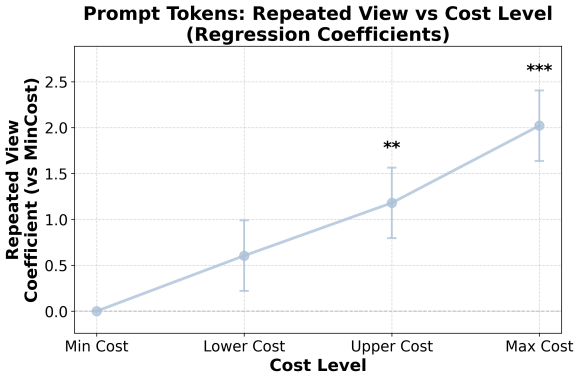


((b)) Prompt tokens: regression coefficients by cost level.

Figure 2: **Accuracy as a function of prompt token usage and cost.** Left: Group-level accuracy plotted against mean prompt token usage across models, with the dashed line indicating a mixed-effects regression trend controlling for model identity. Right: Estimated accuracy coefficients relative to the minimum-cost setting across cost strata, with error bars denoting 95% confidence intervals. Accuracy improvements peak at intermediate cost levels and diminish or reverse at higher costs, indicating a non-monotonic accuracy–cost relationship. Completion-token analyses are deferred to Appendix A.



((a)) Prompt tokens: repeated *modify* actions by cost level.



((b)) Prompt tokens: repeated *view* actions by cost level.

Figure 3: **Repeated file operations increase sharply with cost.** Mixed-effects regression coefficients (relative to the *MinCost* setting) show that higher-cost runs are characterized by a disproportionate rise in repeated file *modify* and *view* actions. This pattern indicates that excessive cost is driven primarily by redundant back-and-forth behavior rather than increased task coverage. Analogous trends for completion tokens are shown in Appendix A. Error bars denote 95% confidence intervals.

Human vs. Model Perception of Difficulty We analyze how problem difficulty affects token consumption and tool usage. The difficulty levels follow the SWE-bench Verified dataset (Jimenez et al., 2024b; Chowdhury et al., 2024), which categorizes problems based on the estimated time required by professional developers to resolve them (e.g., "<15 min", "15 min – 1 hour", "1–4 hours", ">4 hours"). Because there are only three instances in the >4 hours category, we merge it with the 1–4 hours group and report them together as >1 hours.

Aggregating runs across all models, Figure 4 shows the distribution of prompt tokens, completion tokens, and tool usages across difficulty levels. While overall resource consumption tends to rise with problem difficulty, the relationship is far from linear. Notably, **11.51%** of tasks labeled as "<15-minute" required more total tokens than the average ">1-hour" instance, and **27.38%** of ">1-hour" tasks consumed fewer tokens than the "<15-minute" group.

These outliers highlight that human-estimated difficulty does not always align with the model's internal notion of complexity. Tasks that seem easy to humans may still demand extensive reasoning, exploration, or tool interaction from the model, 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 model resource expenditure.

3.2 Token–Cost Dynamics Across Phases and Rounds

To understand how different token types contribute to overall cost, we analyze the relationship between token usage and cost both at an aggregate phase level and within a representative round-level case study.

Charging scheme The total cost for each round of interaction returned from Claude API is composed by separately accounting for four categories of tokens: (i) non-cached prompt tokens, (ii) completion tokens, (iii) cache creation tokens, and (iv) cache read tokens. Non-cached prompt tokens correspond to input text that is processed without leveraging the cache, whereas cache tokens are either created (written once to enable future reuse) or read (retrieved in later rounds at a lower marginal cost). Each category is billed at a distinct rate, with cache creation depending on the persistence setting (here we use the 5-minute write rate). The total per-round cost is simply the sum of these components.[2]

Phase-level Token Usage Dynamics We divide each problem-solving trajectory into five chronological phases—*Early*, *Early-Mid*, *Mid*, *Later-Mid*, and *Later*—based on the total number of interaction rounds for each problem. Within each phase, we aggregate data across 500 problem instances and compute per-round statistics, including the correlation between token types and cost, as well as the average proportion of each token type in both token count and cost.

Building on this setup, we analyze how different token types contribute to cost and usage throughout the agent's problem-solving trajectory. The overall token composition remains stable across phases (Figure 5(a)), with cache-read tokens accounting for most token usage. This stability suggests that although the total number of tokens fluctuates, the fundamental interaction structure remains consistent throughout the process. Notably, the token count proportions of non-cached prompt tokens and cache creation tokens remain nearly identical across all five phases, reflecting that whenever new (non-cached) prompt tokens appear, they are subsequently cached for reuse in subsequent rounds.

[2]Pricing details are available at https://docs.claude.com/en/docs/about-claude/pricing. The cost calculation is shown in Appendix B.

The proportion of cache-read tokens in total cost increases steadily across phases (Figure 5(b)), yet their correlation with total cost remains consistently low (Figure 5(c)). This indicates that cache reads are economical operations: as their cost share rises, it reflects efficient reuse of cached information rather than increased expenditure. In other words, greater reliance on cache reads corresponds to cheaper progress rather than inflated computation.

Finally, as shown in Figure 5, cache creation and non-cached prompt tokens dominate the early stages, when the agent constructs its working context. Completion tokens become increasingly influential from the mid to later phases, showing the strongest correlation with total cost and reflecting the token-intensive nature of reasoning and code generation.

Round-level cost breakdown. To complement the phase-level aggregate view, we now zoom into a representative case study. Figure 6 presents the exact per-round cost decomposition across the four token categories. We observe that the cost of *non-cached prompt tokens* closely mirrors that of *cache creation tokens*, which is expected given their similar per-token rates. Moreover, whenever a segment of non-cached prompt tokens appears, it is typically followed by corresponding cache creation, reflecting the mechanism by which uncached input is soon persisted for reuse. The *cache read tokens* accumulate steadily over rounds and constitute a large fraction of the total cost; however, the total round cost does not monotonically increase, explaining the low correlation between cache reads and per-round cost observed in Figure 5(c). Consistent with the trends highlighted earlier, the dominant cost drivers vary across phases: in the *Early* phase, cache creation dominates total cost, while in the *Later* phases, completion tokens become the primary contributor as generation expands.

Trajectory behavior analysis We selected six representative rounds and inspected their detailed trajectories to understand how the agent balances tool use, code generation, and verification. A compact summary of the representative steps is shown in Table 1.

Rounds dominated by non-cached prompt tokens. In three rounds, token usage was dominated by uncached input from tool calls. The agent relied heavily on repository exploration and targeted searches (e.g., using `grep` or signature inspection) to locate

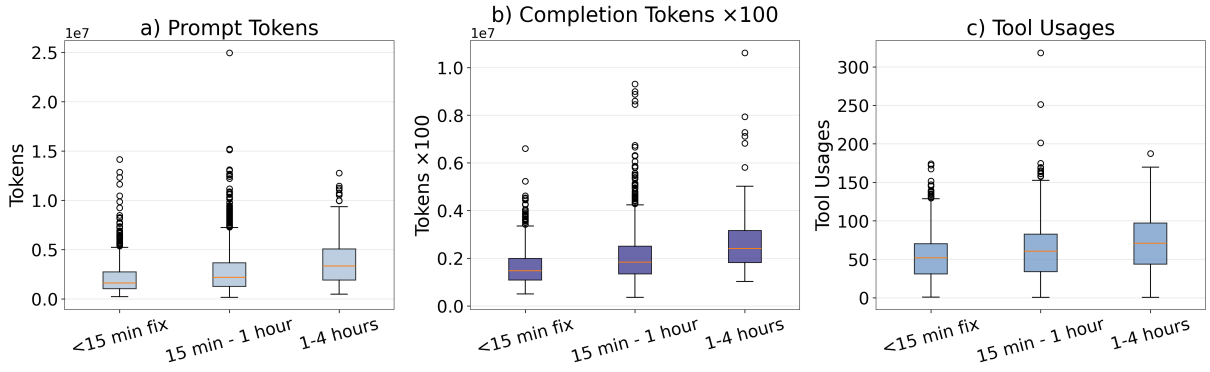
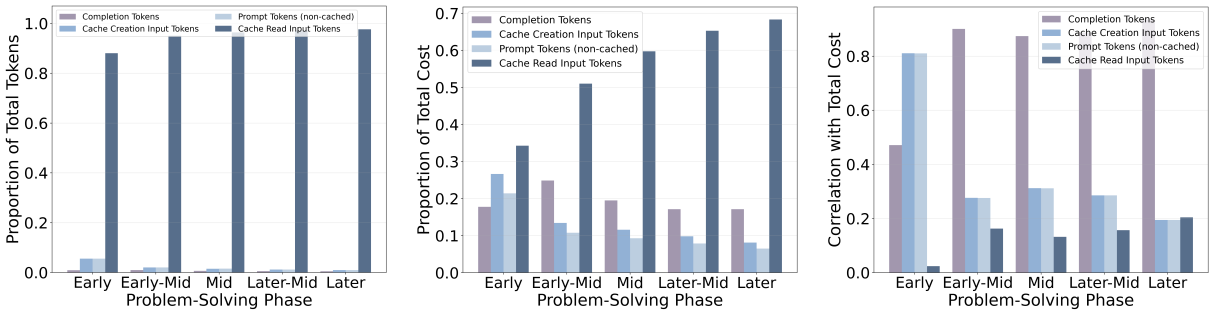


Figure 4: **Token and tool usage across difficulty levels.** Harder tasks generally consume more resources, though some easy ones still show high token usage, revealing large behavioral variance.



((a)) Proportion of total tokens. ((b)) Proportion of total cost. ((c)) Correlation with total cost.

Figure 5: **Token usage and cost composition across problem-solving phases.** Each phase (*Early* → *Later*) represents an equal partition of the agent's trajectory. (a) Token composition remains stable across phases, with non-cached prompt and cache-read tokens dominating total token usage. (b) Cost composition shows that cache creation tokens dominate in the early stages, while cache-read costs become the primary contributor in later phases. (c) Correlation analysis shows that non-cached prompt and cache creation tokens dominate early phases, while completion tokens are most correlated with total cost later.

relevant code regions and align them with stack traces. These actions generated large uncached input dumps, while the completions were relatively short, focusing on synthesizing evidence rather than producing long code edits.

Rounds dominated by completion tokens. In the other three rounds, completion tokens dominated due to long generated outputs. The agent created external reproducers, drafted minimal harnesses, implemented guarded fixes (e.g., for None return annotations), and validated results with environment overrides. These trajectories involved extensive explanatory text and multi-line code edits, with most tokens coming from generated content rather than tool outputs.

4 Predicting Token Consumption

In this section, we investigate approaches for predicting the token consumption of the OpenHands agent. Prediction quality is measured using the Pearson correlation coefficient.

4.1 Self-Prediction by the Agent

We experiment with using the coding agent itself as the predictor of token consumption. This design is motivated by the intuition that the agent responsible for executing a task has the most direct access to its own workflow, including intermediate planning, tool usage, and exploration behavior. In this setting, we modify the instructions of the OpenHands coding agent so that its objective is to estimate token usage rather than to solve the coding problem. The agent retains its full tool-calling and interaction capabilities, but is guided by an updated system prompt to predict the token cost required for a given task. This setup allows the agent to actively explore the codebase, generate intermediate plans, and reason about the expected workload before execution.

Prediction Setting We focus on a fine-grained self-prediction setting, in which the agent is ex-

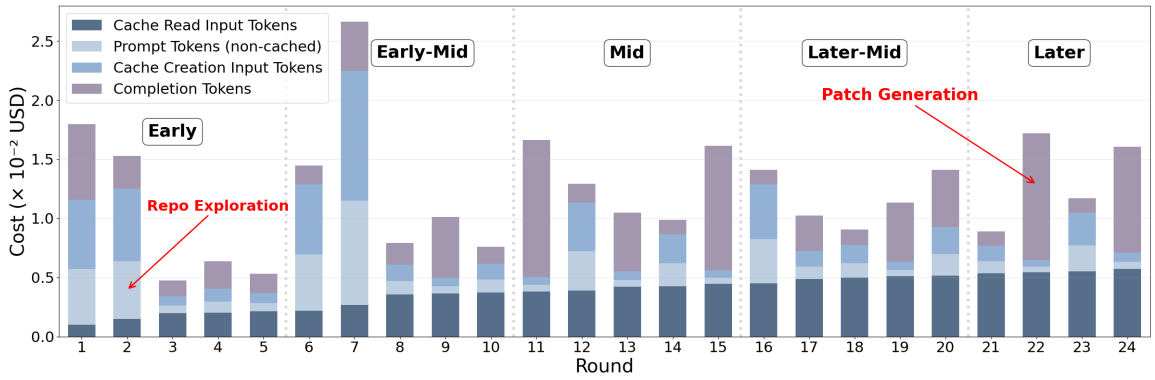


Figure 6: **Round-level token cost breakdown.** Exact per-round costs decomposed into non-cached prompt tokens, cache creation (5m persistence), cache reads, and completion tokens. The dominant cost source shifts from cache creation in early phases to completion tokens in later phases.

Round	Tool Usage	Action summary
1	`execute_bash` (ls/grep)	Locates decorator, call sites, and links trace.
2	`execute_bash` (repo search)	Explores files broadly, traces errors.
7	`execute_bash` (signature search)	Dumps large files while checking signatures.
11	`str_replace_editor`	Writes reproducer, explains failure, suggests fix.
15	`str_replace_editor`	Proposes patch for `None` return issue.
22	`str_replace_editor`	Adds guard for `None` and verifies fix.

Table 1: Representative trajectory steps across analyzed rounds.

plicitly instructed to decompose the task and produce detailed estimates of token usage (see Appendix C.2 for the full prompt). In practice, we observed that several models struggled to reliably follow the prediction instructions, often deviating from the required format or failing to produce usable estimates. We therefore restrict our analysis to Sonnet-4.5 and GPT-5.2, which demonstrated more consistent instruction-following behavior and were able to reliably complete the self-prediction process. Due to budget limitations, we conducted three independent runs using the same set of 500 samples.

Results and Observations Table 2 summarizes the performance of agent self-prediction across models and token types. Using the coding agent as the predictor yields non-trivial but modest correlations with ground-truth token usage for both Sonnet-4.5 and GPT-5.2. GPT-5.2 consistently achieves stronger alignment with ground truth than Sonnet-4.5, particularly for output-token prediction (0.213 vs. 0.123). Overall, these results indicate that self-prediction captures coarse trends in token usage but remains noisy at the instance level.

Self-Prediction Cost vs. Task Cost and Prediction Error We further analyze whether the cost incurred by self-prediction is informative of the actual task cost and whether it correlates with prediction error. As shown in Table 2, self-prediction cost exhibits a weak but positive correlation with task cost for both models, with GPT-5.2 showing a stronger association (0.246) than Sonnet-4.5 (0.119). This suggests that self-prediction cost partially reflects underlying task difficulty, though the relationship remains limited. Importantly, correlations between task cost and absolute prediction error are generally small and inconsistent across settings, and in some cases negative (e.g., Sonnet-4.5 output tokens). These results indicate that higher self-prediction cost does not reliably correspond to improved or degraded prediction accuracy. Taken together, self-prediction provides a low-cost, coarse-grained signal of relative task difficulty without introducing systematic bias into token estimation quality.

4.2 Challenges

While it is natural to let a coding agent predict its own token consumption—since it is arguably most familiar with its own decision-making—we find that this approach, despite yielding higher correlation with actual usage, suffers from several challenges. In both settings we have explored, self-prediction tends to consistently overestimate the true cost. We also considered providing few-shot examples to give the agent a better sense of

Model	Token type	Corr w/ GT	Corr (AbsErr, task cost)	Corr (pred cost, task cost)
Sonnet-4.5	Input tokens	0.1355	0.1155	0.1185
	Output tokens	0.1229	-0.4563	
GPT-5.2	Input tokens	0.1796	0.2243	0.2461
	Output tokens	0.2130	0.0822	

Table 2: Prediction performance across settings for self-prediction by the same agent.

the token cost per round. However, it is difficult to construct representative high-quality examples: available samples are heterogeneous and often fail to capture general patterns, and directly including multi-round few-shot traces would quickly exceed the context window of current LLMs. Therefore, few-shot settings require further study. Moreover, compared to the cost of a single LLM call, users may be unwilling to tolerate the latency and overhead introduced by multi-round estimation. As a result, self-prediction, although appealing in principle, remains challenging in practice.

5 Discussion

In this paper, we present the first empirical analysis of agent token consumption and explore different methods to predict it. In this section, we highlight the main limitations of our study and discuss the implications of our findings for the design and pricing of agent-based systems.

Limitations A key limitation of our study lies in the range of agentic model backbones evaluated. Although we analyze multiple models, including Sonnet 3.7, Sonnet 4, Sonnet 4.5, GPT-5, GPT-5.2, Qwen Coder, Kimi-K2, and Gemini 3 Pro (Preview), this still represents only a limited subset of the broader agentic model landscape. Collecting full execution trajectories for additional models is extremely costly in both time and compute, which constrains the breadth of our evaluation. While the qualitative patterns we observe are consistent across the models tested, validating these findings on a wider range of architectures and agent designs would further strengthen their generality. To facilitate such extensions, we release our experimental pipeline to enable future work to replicate and expand our analysis.

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. This makes accurate prediction of token consumption important for designing sustainable pricing strategies. Our findings show that token usage, especially input tokens, is highly variable and difficult to predict due to the stochastic nature of agent trajectories. As a result, purely upfront pricing remains challenging, and consumption-based pricing may remain the most practical option until more reliable pre-execution cost estimation methods are developed.

User Transparency Reliable token predictions are also important for user transparency and trust. Ideally, an agentic system would be able to inform users about the likely cost of a task before execution, allowing them to make informed decisions. However, current language models struggle to provide accurate point estimates, limiting the feasibility of presenting exact costs in advance. That said, our results suggest the potential of using agent itself for cost prediction, which could help systems identify tasks with potentially high cost. Such signals may enable providers to issue early warnings, request explicit user approval, or offer alternative execution modes before proceeding, even when precise predictions are not possible.

6 Conclusion

With the rapid growth of agent token consumption in various settings, predicting future token usage before task execution becomes an important task for reasonable and transparent pricing of AI agents. In this paper, we propose the first study aimed at understanding and predicting agent token consumption on agentic coding tasks. Through a series of empirical analyses of agent trajectories, we reveal important findings about the pattern of agent token usage in agentic coding tasks. Furthermore, we explore a range of token usage prediction methods and highlight the challenges of predicting agent token consumption before task execution.

References

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. 2024. Introducing SWE-bench verified.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024a. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

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

Kinde. 2024. Ai token pricing optimization: Dynamic cost management for llm-powered saas.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023a. Repobench: Benchmarking repository-level code auto-completion systems. *Preprint*, arXiv: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, and 3 others. 2023b. Agentbench: Evaluating llms as agents. *Preprint*, arXiv:2308.03688.

OpenAI. 2025. Introducing codex.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *Preprint*, arXiv:2408.03314.

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*.

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, and 5 others. 2025. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2025. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *Preprint*, arXiv:2408.00724.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Preprint*, arXiv:2405.15793.

A Completion Token Analyses

This appendix presents complementary analyses using *completion tokens* in place of prompt tokens. Across all settings, the completion-token results closely mirror the trends reported in the main text: accuracy decreases as completion-token cost increases, and higher-cost runs are associated with a sharp rise in repeated file *view* and *modify* actions. These findings reinforce the conclusion that excessive computation is primarily driven by redundant agent behavior rather than productive progress, and that the inverse accuracy–cost relationship is not specific to prompt tokens.

B Cost Calculation Details

B.1 Explicit Caching Models (Claude Models)

$$\text{Prompt}_{\text{non-cached}} = \text{Prompt}_{\text{total}} - \text{CacheRead}. \quad (1)$$

$$\begin{aligned}\text{Cost}_{\text{round}} = {} & (\text{Prompt}_{\text{non-cached}} \times r_{\text{in}}) \\ & + (\text{Completion} \times r_{\text{out}}) \\ & + (\text{CacheCreation} \times r_{\text{cache\_create}}) \\ & + (\text{CacheRead} \times r_{\text{cache\_read}}). \quad (2)\end{aligned}$$

where r_{in} is the base input rate, r_{out} is the output rate, $r_{\text{cache\_create}}$ is the cache creation rate (5-minute writes in our setting), and $r_{\text{cache\_read}}$ is the cache read rate.

B.2 Implicit Cache (GPT5 and alike)

For GPT-5 models, we use OpenAI's implicit caching mechanism. The API reports cached prompt tokens automatically, without explicit cache creation. At the time of our experiments, the official pricing is: *Input: $1.250 / 1M tokens*, *Cached input: $0.125 / 1M tokens*, and *Output: $10.000 / 1M tokens*.[3]

The non-cached prompt is

$$\text{Prompt}_{\text{non-cached}} = \text{Prompt}_{\text{total}} - \text{CacheRead}_{\text{implicit}}. \quad (3)$$

The total cost is

$$\begin{aligned}\text{Cost}_{\text{round}} = {} & (\text{Prompt}_{\text{non-cached}} \times r_{\text{in}}) \\ & + (\text{CacheRead}_{\text{implicit}} \times 0.2\, r_{\text{in}}) \\ & + (\text{Completion} \times r_{\text{out}}). \quad (4)\end{aligned}$$

[3] See the official pricing documentation: https://openai.com/index/introducing-gpt-5/

C Prompt for Self-Prediction by the Same Agent

C.1 Zero-Shot Setting

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 with actual code changes.

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.

Follow these phases to estimate token costs:

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

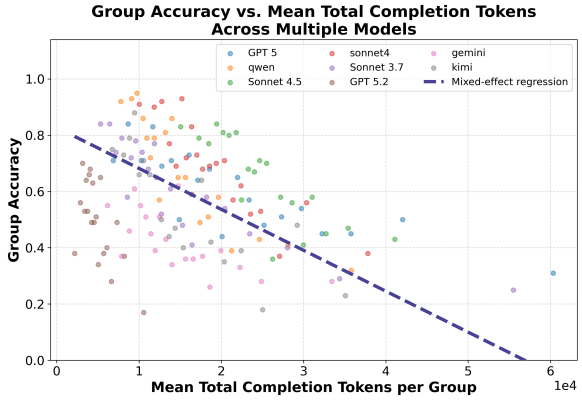
- 1.1 Read the problem description and understand what needs to be fixed
- 1.2 Explore relevant files and directories to understand the codebase structure
- 1.3 Search for key functions, classes, or variables related to the issue
- 1.4 Identify the root cause and complexity of the problem

Phase 2. ANALYSIS: Analyze the complexity and required changes

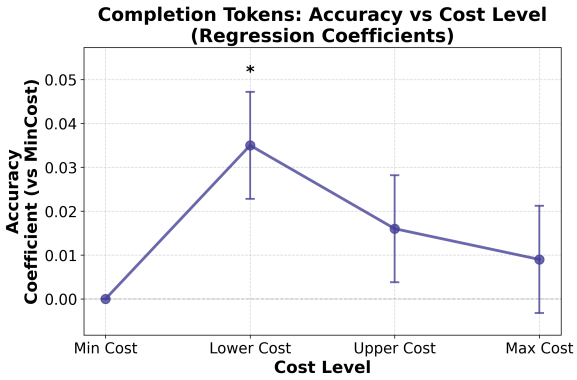
- 2.1 Assess the scope of changes needed (number of files, lines of code)
- 2.2 Consider the debugging iterations likely needed
- 2.3 Evaluate the testing complexity and iterations
- 2.4 Estimate the number of tool calls and reasoning steps

Phase 3. TOKEN ESTIMATION: Calculate token usage for the complete solution

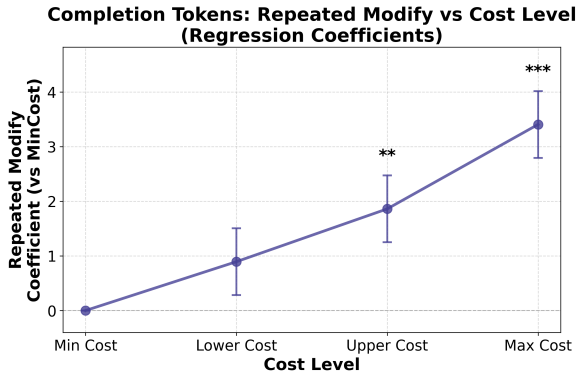
- 3.1 Estimate input tokens for:
  - Repository exploration and file reading
  - Code analysis and debugging
  - Implementation iterations



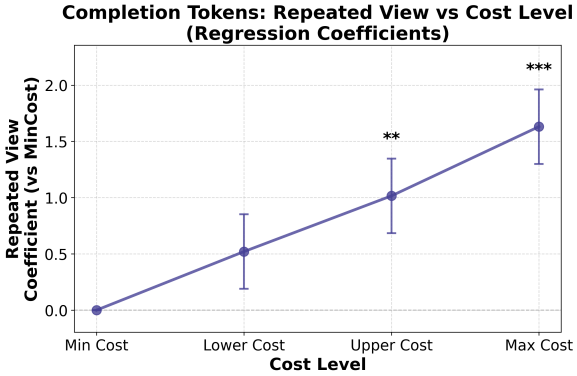
((a)) Group accuracy vs. mean completion tokens across models.



((b)) Completion tokens: regression coefficients by cost level.



((c)) Completion tokens: repeated *modify* actions by cost level.



((d)) Completion tokens: repeated *view* actions by cost level.

Figure 7: **Completion-token analyses.** Top: Group-level accuracy plotted against mean completion-token usage across models, with mixed-effects regression trends controlling for model identity. Bottom: Mixed-effects regression coefficients (relative to the *MinCost* setting) for repeated file *modify* and *view* actions across cost levels. The completion-token results mirror the prompt-token trends reported in the main text, exhibiting decreasing accuracy with higher cost and a sharp rise in redundant file operations at higher cost levels. Error bars denote 95% confidence intervals.

- Testing and verification

- 3.2 Estimate output tokens for:
  - Reasoning and analysis responses
  - Code generation and explanations
  - Debugging responses
  - Test results interpretation

- 3.3 Calculate total tokens and confidence level

Phase 4. FINISH: Provide final token estimate

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

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.

C.2 Zero-Shot + Fine-grained Breakdown Setting

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 with actual code changes.

Fine-Grained Requirements:

- Round all estimates to the nearest **10 tokens** (e.g., 30, 40, 70). Do NOT round to hundreds.
- Break down into **atomic actions**: reading issue description, listing files, inspecting one file, analyzing one function/class, one debugging iteration, one test run.
- Do NOT output ranges (e.g., "200–300"). Always a single number.
- First give sub-action estimates, then sum into totals per phase, then overall totals.
- Final JSON must contain fine-grained estimates inside `breakdown_by_phase`.

(Other parts are the same as in the zero-shot setting and are omitted.)

D The Use of Large Language Models (LLMs)

We used different LLMs to prediction token consumption in our experiments. In addition, large language models (e.g., ChatGPT) were used only as general writing aids to polish language, improve clarity, and format prompts. All experimental design, data analysis, and conclusions were made and verified by the authors.